Univerzitet „UNION” u Beogradu
Računarski fakultet

Nikola Todorović

Primena tehnika veštačke inteligencije u prepoznavanju oblika na slikama

Diplomski rad

Beograd, 2007. godine

Računarski fakultet u Beogradu
Univerzitet „UNION”

Diplomski rad

Primena tehnika veštačke inteligencije u prepoznavanju oblika na slikama

Kandidat: **Nikola Todorović**

Članovi komisije: **Dr. Dragan Urošević, docent**
Dr. Dragan Šaletić, docent

Beograd, 2007. godine

Apstrakt

Nema sumnje da živimo u svetu gde računari imaju jako bitnu ulogu. Omogućavaju nam pristup informacijama, naši poslovi, edukacija, komunikacija, pa čak i životi zavise od njih. Još uvek je rano reći da računari mogu da rezonuju i donose odluke poput čoveka, ali je evidentno da se svakog dana vrši napredak u tom smeru. U poslednje vreme se pojavljuju roboti koji umeju da pričaju, hodaju poput čoveka, pa čak i plešu u ritmu muzike. Tu su zatim programi koji precizno prognoziraju vremenske prilike, vrše razne procene i druge vrste proračuna.

Da bi se moglo reći da jedan kompjuterski program rešava određeni problem jednako dobro kao i čovek, uveden je poseban vid testiranja, poznat pod nazivom Turingov test, prema poznatom naučniku Alenu Turingu. Turingov test se sastoji iz problema koji rešavaju nezavisno računar i čovek. Ukoliko nije moguće razgraničiti rešenje kompjutera od rešenja čoveka bez znanja koje je čije, kaže se da je računarski program prošao Turingov test. Jedan od vidova testiranja kojim će se upravo baviti ovaj rad jeste CAPTCHA (Completely Automated Public Turing test to tell Computer from Humans). To je test koji je između ostalog moguće isprobati na Internetu.

Kao posebnu vrstu ovog testiranja, izdvojićemo jednu sa kojom se susrećemo jako često, prilikom otvaranja naloga, preuzimanja fajlova, pisanja komentara. Naime, često se, kao vid zaštite protiv automatskog pristupa programa, spama ili nekontrolisanog odvlačenja velike količine podataka, koristi slika sa nekoliko karaktera – uglavnom slova, brojeva i znakova interpunkcije. Ti karakteri su na slici uvek napisani tako da je otežano njihovo čitanje i prepoznavanje. Okrenuti su pod različitim uglovima, sadrže smetnje u vidu drugih linija koje ih precrtavaju, iskrivljeni su, obojeni različitim bojama... U ovom radu će se primenjivati razne tehnike sa ciljem prepoznavanje teksta koji je napisan na ovakvim slikama.

Glavni cilj ovog istraživanja i eksperimenta, osim još jednog u niz pokušaja da se računari približavaju ljudima po načinu odlučivanja, je i prikazivanje slabosti ove vrste zaštita. Poenta je u tome napraviti rešenje za neke posebne vrste sličica koje se pojavljuju na pojedinačnim Internet lokacijama i koje međusobno liče. Uostalom, realnije je da spamer (eng. *spammer*) kreira program kojim bi napao' jedan određeni sajt, a ne da napravi program koji bi mogao da spamuje ceo internet (što bi on naravno voleo). Osim toga, kao jedan novi vid spama, pojavljuje se i mail sa sličicama na kojima je ispisan tekst, koji programi za zaštitu od neželjene pošte ne mogu da pročitaju. Očigledno je da bi moderni antispam softver morao da započne sa zaštitom koja bi se oslanjala na upravo ovakav program.

Ključne reči: računarstvo, veštačka inteligencija, genetski algoritmi, tehnike pridruživanja, klasifikator, programiranje, CAPTCHA, spam, Java

Sadržaj

1 Uvodni deo

1.1 Inteligencija

Veštačka inteligencija je oblast u računarstvu čiji je primarni cilj stvaranje mašina koje su u stanju da se ponašaju na način koji čovek smatra inteligentnim. Bez obzira što je to godinama mogao biti samo san, taj san se sve više približava stvarnosti. Uz pomoć sve bržih računara sa sve većom memorijom, kvalitetnim pratećim hardverom i programerskim tehnikama veštačke inteligencije, mašine se sve više približavaju čoveku. Kompjuterski programi su u stanju da uče, donose odluke, rezonuju na osnovu situacije. Sve je više robota koji mogu da se kreću, koji imaju optičke, zvučne ili dodirne senzore i tako oponašaju ljudske radnje i radnje životinja. Premda je glavna primena ovakvih mašina u industriji zabave, sve je više robota koji služe kao pomoć u kući, na poslu, ali i u medicini. Zbog preciznosti i sićušnosti, roboti se koriste prilikom izvršavanja nekih komplikovanih operacija (http://en.wikipedia.org/wiki/Robotic_surgery). Takođe, danas računarski programi sa uspehom predviđaju stanje na berzi i meteorološke uslove, što su stvari koje čovek ne može baš sa velikim uspehom da predvidi. Najbolji šahovski programi se ravnopravno nadmeću sa vrhunskim velemajstorima. Američka vojna agencija DARPA već duže vreme vodi projekte za automatski navođena vozila bez ljudske pomoći, tako da su organizovali trke automatskih vozila kroz pustinju, a novembra ove godine organizuju i trku kroz naseljeno područje. Na osnovu svih ovih primera i s obzirom na brz razvoj informatike i računarskih nauka, moguće je da ideje iz filmova naučne fantastike uskuro ne budu tako fantastične. Bez obzira što sada deluje tako daleko, ali roboti će svakako sve više ulaziti u naše živote.

Profesor Alen Turing se smatra za oca veštačke inteligencije jer je bio prvi koji je još 1950. godine krenuo da se bavi veštačkom inteligencijom. On je osmislio proceduru koja se danas naziva Turingov test i koja se koristi kao provera da li je mašina zaista inteligentna. Sudija koji je čovek komunicira sa računarom i čovekom. Ukoliko nije u stanju da sa sigurnošću razdvoji koji od dva komunikaciona kanala vodi do čoveka a koji do računarskog programa, kaže se da je program prošao test. S obzirom na slabe mogućnosti oponašanja ljudskog glasa, ta komunikacija je obično u tekstualnom obliku.

Poslednjih 50-tak godina, otkada se uopšte razmišlja o inteligentnim mašinama, iznedrilo je veliki broj programirskih tehnika veštačke inteligencije. Ove tehnike se suštinski razlikuju od standardnih algoritama. Konvencionalni algoritmi imaju uglavnom dozu određenosti, izvesnosti, tačno se zna redosled izvršavanja operacija, koji je unapred definisan i striktan. Standardni algoritmi mogu jako dobro da rešavaju određene probleme, ali oni su ograničeni na taj skup problema. Algoritmi veštačke inteligencije nemaju toliki stepen efikasnosti, ali su zato mnogo primenjiviji na dosta veći broj problema. Osobine zajedničke većini tih tehnika su neizvesnost i stohastičnost. Sposobne su da „uče'' na iskustvu i primenjuju stečena znanja.

Kao neke od najpoznatijih tehnika možemo navesti Neuralne mreže, Rasplinutu (*eng.*

fuzzy) logiku, Evolucione algoritme. Dosta su popularni i simulirano kaljenje, tabu pretraga, slučajno pretraživanje, simpleks metode i druge. Može se lako zapaziti da su te metode često osmišljene na način koji podseća na određene principe iz realnog života. To nam govore i sami nazivi tih metoda. Neuralne mreže sačinjene su tako da asociraju na nervni sistem. Jedna neuronska mreža u računarskom smislu se sastoji od skupa perceptrona (pandan nervnim ćelijama) koji su povezani međusobno sinapsama. Oni primaju signale i selektivno, u zavisnosti od ostalih signala koje dobiju od drugih neurona odlučuju da li će se signali proslediti dalje drugim perceptronima. Tako neuralne mreže mogu da se koriste kod metoda mašinskog učenja i da pokazuju jako dobre rezultate. Često se primenjuju u prepoznavanju modela ili oblika (*eng. pattern*). Evoluciona izračunvanja pronalaze uzor u prirodi i oponašaju način na koji se jedna vrsta prilagođava uslovima okoline. Tu se koriste termini poput populacije, razmnožavanja, mutacije. Odlične rezultate prikazuju u oblastima optimizacije, pretrage, minimizacije funkcija. Rasplinuta logika je posebna vrsta logike, koja je slična ljudskom načinu razmišljanja. Za razliku od klasične Bulove logike, gde iskaz može biti ili tačan ili netačan, u rasplinutoj logici može uzeti bilo kakvu vrednost iz opsega [0,1]. Kao što u realnom svetu možemo reći da je čovek visok ili nizak, ali ne postoji striktna granica koja razdvaja ta dva suprotna pojma. Zato se uvode koeficijenti, i na osnovu toga, razvijena je čitava logika, zajedno sa osobinama koje prate konvencionalnu logiku primenjenim na ovu vrstu logike. Ova tehnika je jako dobra u primeni kod teorije odlučivanja, ekspertskih sistema, regulacije i klasifikacije.

Kao što se može videti, svaka od ovih metoda daje dobre rezultate u nekim vrstama problema. To je formalizovano u teoremi „Nema besplatnog ručka" (*eng. No Free Lunch*) . Formulisali su je Wolpert i Macready. Ona govori o tome da u problemima optimizacije i pretrage ne postoji najbolji algoritam, onaj koji se uvek može koristiti i koji garantuje najbolje rezultate. Za jedan problem će se najbolje pokazati genetski algoritmi, za drugi simulirano kaljenje, za treći neuronske mreže. To nam govori da treba pažljivo pristupati svakom problemu i dobro razmisliti na koji način će se „napasti".

1.2 Spam

Spam, u početku termin korišćen za neželjenu poštu, sada se odnosi na uznemiravanje na Internetu u najširem smislu reči. Pored elektronske pošte, kao glavnog i najrasprostranjenijeg oblika, u tu grupu spadaju i spam preko programa za ćaskanje (*eng. chat*), spam vesti, spam za internet pretraživače, spam u Internet dnevnicima (blogs), forumima, pa i spam preko SMS-a. Mada spam nije tako opasan kao što su virusi i drugi maliciozni programi, sve veće prisustvo spama u svakoj oblasti Interneta svakako svrstava spam u ozbiljnije probleme. Inicijatori spam-a često nemaju neki poseban cilj – osim da uznemiravaju i ometaju druge ljude, i često je takva vrsta spama neškodljiva, osim što oduzima vremena i ume da nervira ljude. Ipak, s obzirom da spam sve više uzima maha, može se slobodno reći da prerasta u biznis i da spameri (inicijatori) sve više primenjuju lukave tehnike kako bi postigli razne ciljeve.

Jedan od glavnih ciljeva spama je reklamiranje. U neželjenim elektronskim porukama mogu se često naći ponude za kupovinu, ulaganje. Prema Internet stranici [6], podela

neželjene pošte prema tome šta se njom reklamira je kao na tabeli 1.1.

Proizvodi	25%
Finansijski	20%
Za odrasle	19%
Prevara	9%
Zdravlje	7%
Za Internet	7%
Opuštanje	6%
Duhovni	4%
Ostali	3%

Tabela 1.1 – Raspodela spam poruka

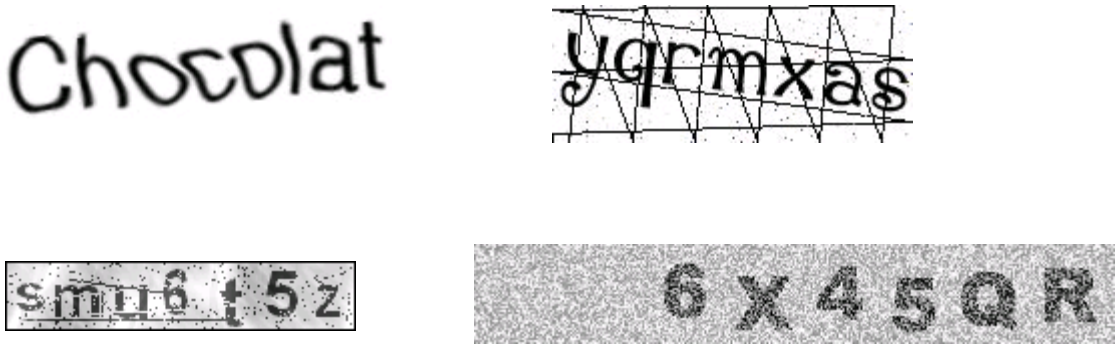
Na porukama preko Internet foruma, često se može videti poplava poruka koje reklamiraju neke sajtove, najčešće za odrasle. Tako je vrlo popularni sajt za gledanje video fajlova youtube meta stalnih napada spamera koji reklamiraju pornografske sajtove ili sajtove za nalaženje ljubavnog partnera. S obzirom na karakteristike Interneta koji je dostupan svima i svako može da radi šta želi, spam je logičan sled događaja. Sajtovi koji su megapopularni imaju i najviše problema sa neželjenim poštama, porukama na forumima i slično. Problem kod spama je što najčešće umesto ljudi koji bi trebalo da kucaju gomilu poruka, to čini aplikacija koju su oni kreirali specijalno za tu priliku. Ta aplikacija može da napiše mnogo više poruka od čoveka, za vrlo kratko vreme. Osim reklamiranja, u spam porukama se nalaze i razni politički i verski motivi i poruke.

Jedan od sofisticiranijih vidova spama jeste spam namenjen Internet pretraživačima. Sajtovi koji pretražuju Internet, kao što su Google®, Yahoo®, Microsoft-ov Live Search®, se danas susreću sa novim poteškoćama. Naime, da bi neki pretraživač mogao da zna koji je sajt popularniji, kvalitetniji, poznati PageRank® algoritam za rangiranje i njegove modifikacije se koriste. Taj algoritam se oslanja na broj linkova koji pokazuje na određenu Internet stranicu, smatrajući stranicu na koju pokazuje veći broj linkova popularnijom, što je u prvim godinama Interneta zaista i važilo. Poznajući tu metodologiju, spameri su se dosetili i rešili da naprave gomile stranica koje ukazuju na neku stranicu, ne bi li je podigli na što bolje plasirano mesto kada korisnici vrše pretragu po Internetu. Takva vrsta spama se naziva *spamdexing*, što je reč nastala spajanjem dve reči (eng. *spam* i *indexing*). Sama činjenica da se to masovno radi nam dokazuje da je spam jako ozbiljna stvar i da zaista možemo tvrditi da se spam pretvara u posao. Prema nekim okvirnim procenama polovina Internet sadržaja su pornografske stranice, dok je polovina preostalih stranica spam. Zaista, spam sve više ugrožava normalan razvoj Interneta i pravi probleme.

Naravno, ni velike kompanije ne sede mirno pored tolike gomile neželjenih poruka. Dosta se napora ulaže u razne antispam zaštitne metode. Poznati sajtovi kao što su *craigslist* i *youtube* se uzdaju u svoje korisnike, tako što omogućavaju da se poruke označe kao neželjene. Na taj način, ako dovoljan broj korisnika označi neku poruku kao spam, ta će poruka biti obrisana, a njen inicijator ima velike šanse da mu se isključi nalog. Tu se naravno postavlja pitanje koga ima više i ko ima veći uticaj, spameri ili dobronamerni korisnici. Spam se šalje automatizovano, tako da to daje dobre mogućnosti da se napravi zaštita. Svi današnji serveri elektronske pošte poseduju zaštitne mehanizme, koji rade

bolje ili gore. Da bi se napravio dobar spam filter, analiziraju se reči koje se pojavljuju u tekstu poruke, njihova učestanost, elektronska adresa pošiljaoca, i još dosta drugih podataka. Recimo, veliki serveri imaju i veliki broj korisnika i svakodnevno im pristižu milioni poruka. Njima takođe pomažu dobronamerni korisnici, koji mogu okarakterisati određene poruke kao spam i kada se ta ista poruka pošalje na drugu adresu u okviru istog servera, spam filter će je sam odstraniti. Međutim, elektronska pošta omogućava ne samo slanje teksta, već i fajlova u vidu dodataka. Kao nešto komplikovaniji vid neželjene pošte, pojavljuju se sasvim jednostavne poruke koje liče na obične, i koje u dodatku prosleđuju fajl, obično sliku. Na toj slici se nalazi propagandna poruka. Kako se poruka prema kojoj se spam razotkriva ne nalazi u tekstu već u slici, to biva teže antispam filtru da je klasifikuje kao nepoželjnu poštu. Činjenica je da se takve poruke često provuku do elektronskog sandučeta autoru ovih redova ali i ljudima koje poznaje.

Još jedna izuzetno raširena zaštita od spama je takozvana CAPTCHA zaštita. CAPTCHA® je akronim (**C**ompletely **A**utomated **P**ublic **T**uring test to tell **C**omputers and **H**umans **A**part) koji je zaštitni znak Karnegi Melon univerziteta. Kompjuter je u stanju da generiše Turingov test (o kome je pisano u prethodnom odeljku) i da razlikuje ljude od računara, ali ne i da reši test. U najvećem broju slučajeva, ovaj test se odnosi na slike koje u sebi sadrže tekst koji je na određeni način izmenjen raznim smetnjama, dovoljno da kompjuteru bude jako teško da odredi koji su znakovi na slici, a opet dovoljno lako čoveku koji može bez ikakvih problema da ih raspozna. Čoveku je potrebno nekoliko sekundi da otkuca kji su karakteri na slici, i može da nastavi sa željenim akcijama. Najčešće, takve akcije bi bile registracija na raznim stranicama (za elektronsku poštu, forumi i dr.), zatim slanje poruka, pa prebacivanje i dovlačenje fajlova. Dakle, to su akcije koje su osetljivog karaktera i kojima Internet stranice nikako ne žele automatski pristup. CAPTCHA sličice mogu izgledati kao na slici 1.1.



Slika 1.1 – Različite vrste CAPTCHA slika

CAPTCHA sličice ima gotovo svaki veliki sajt i one u mnogome otežavaju posao spamerima. Najčešće se koriste prilikom prijavljivanja na Internet stranicu, za elektronsku poštu, forume, prililkom slanja poruka.

1.3 Cilj i plan projekta

Ovaj rad ima više poetni i ideja vodilja koje će sada biti navedene. Glavna ideja rada je dati još jedan od dokaza kako kompjuter može biti u stanju da raspoznaje stvari poput čoveka. U ovom specijalnom slučaju, radi se o kompjuterskom viđenju i želja je pokazati da kompjuter može raspoznavati oblike na slikama jako dobro. Sledeći cilj je pokazati kako CAPTCHA zaštita nije dovoljno dobra i kako se može zaobići. Svakako, iluzoran bi bio u ovom trenutku pokušaj da se napravi jedinstveni program koji bi rešavao mnoge ili čak sve vrste CAPTCHA sličica. Za to je potrebna grupa ljudi, i jako puno vremena i truda uloženog u to. Postoje sada razni projekti, kao što su http://sam.zoy.org/pwntcha/ i http://www.cs.sfu.ca/~mori/research/gimpy/, koji se trude da rešavaju CAPTCHA problem, i oni rešavaju jedan po jedan tip slika. Takođe, ako se vratimo na priču o spamu, vrsta neželjene pošte koja zadaje dosta problema je poruka koja ima dodatnu sliku i u njoj skrivenu poruku. Ako bi ta slika bila pregledana programom koji može da pročita šta je na slici, onda bi antispam filter mogao da je okarakteriše kao spam. To bi značilo da se program koji prepoznaje CAPTCHA slike može primeniti u antispam zaštiti.

Projekat se u opštem slučaju sastoji iz više etapa. Najpre je potrebno prikupiti veliki broj CAPTCHA sličica, koje se koriste za obučavanje programa. Radi treniranja programa, potrebno je da čovek izvrši obeležavanje sličica, tj. da označi koji znaci idu uz koju sliku, kako bi se napravila baza podataka uz pomoć koje će se izvršiti trening. Zatim, s obzirom da CAPTCHA slika sadrži dosta veštačkih smetnji koje otežavaju prepoznavanje znakova, program treba da eliminiše te smetnje i izdvoji karaktere koji se pojavljuju. Ti šumovi mogu biti različiti, u vidu šarenih boja, rešetki, čudnih pozadina. Nekad može biti dodatni problem i što se znakovi međusobno prepliću. Završni korak je napraviti program koji može, kada već dobije izdvojeni znak da prepoznaje koji je karakter u pitanju. Nekada je u zavisnosti od specifičnosti koju poseduje neki tip CAPTCHA slika, potreban još neki korak, ali u opštem slučaju, ovo je ono kroz šta treba proći. Za ovaj rad je izabran tip CAPTCHA sličica *b2evo* koje su javno dostupne i mogu se naći na sajtu www.b2evolution.net. Ono što je posebno zanimljivo a tiče se njih, jeste da se koriste na nekim domaćim popularnim Internet stranicama. Neki primeri dati su na slici 1.2.

Slika 1.2 – Izgled CAPTCHA slika koje se obrađuju u ovom radu

2. Tehnologije

2.1 Genetski algoritmi

Evolucione tehnike predstavljaju skup tehnika veštačke inteligencije i zajedničko im je to što koriste motive preuzete iz prirode. Izuzetno su primenljive u problemima optimizacije i pretrage. Dele se na dve grupe tehnika, grupnu inteligenciju i evolucione algoritme. Kod grupne inteligencije obično postoji skup agenata koji čini kolektivno jedan sistem gde na osnovu pojedinačnih rezultata i međusobnih interakcija dolazi do globalno dobrih rešenja. Jedan od najčešćih predstavnika ovih algoritama je optimizacija kolonije mrava. Evolucioni algortmi se prema svojim idejama baziraju na elementima poznatim iz Darvinove teorije evolucije, kao što su prirodna selekcija, ukrštanje i mutacija. Kod ovog tipa algoritama postoji populacija jedinki, od kojih svaka predstavlja rešenje. Najbolje jedinke opstaju i kombinuju se kako bi se smenama generacija došlo do boljih rešenja. Tako ovaj sistem podseća na prirodni razvoj neke vrste i njeno adaptiranje na spoljašnje uslove. Evolucioni algoritmi se dele na četiri osnovne tehnike: evolucione strategije, evoluciono programiranje, genetsko programiranje i genetske algoritme. Ovaj odeljak posvećen je genetskim algoritmima, jer će biti korišćeni u daljem izlaganju, pa bi bilo zgodno imati u vidu način na koji se stiže do rezultata u ovoj podvrsti evolucionih tehnika.

Genetski algoritmi se koriste da bi se pronašao globalni minimum/maksimum neke funkcije ili njegova aproksimacija. Osmislio ih je John Holland, zajedno sa svojim kolegama i studentima na Mičigen univerzitetu u Americi i prva publikacija na tu temu objavljena je 1975. godine [8]. Bazirani su na principima genetike i prirodne selekcije. Populacija, koja se sastoji od velikog broja pojedinačnih rešenja evoluira, i tako se približava najboljem rešenju. Svako pojedinačno rešenje, koje se u okviru ove vrste algoritama naziva jedinkom, predstavlja jednu vrednost iz domena funkcije koja se optimizuje. Pošto je obično ta funkcija višedimenzionalna, to se i ta vrednost sastoji od više parametara. Vrednosti parametara su posebne za svaku jedinku. Funkcija koja se optimizuje se obično naziva funkcija evaluacije ili fitnes funkcija (funkcija prilagođenja), zato što ona služi za ocenjivanje jedinki i govori o tome koliko je koja jedinka kvalitetna.

Kao što je rečeno, svaka jedinka sadrži skup parametara na osnovu kojih se određuje njena sposobnost. Ti parametri, odnosno njihova prezentacija, nazivaju se genom ili hromozomom. Parametre možemo predstaviti na više načina, pa i vrste gena mogu biti različite. Najčešći su geni u vidu niza bitova, mada ima i drugih mogućnosti. U genetskim algoritmima od najvećeg značaja su četiri operacije a to su selekcija, uparivanje (ili nalaženje partnera za razmnožavanje), zatim ukrštanje i mutacija. Najpre će biti objašnjene sve četiri vrste operatora, a potom i razlog zbog kojeg genetski algoritmi zaista funkcionišu i imaju primenu u realnosti. Implementacija jednog genetskog algoritma bi mogla biti predstavljena sledećim koracima:

1. Inicijalizacija populacije, bira se slučajnih N jedinki iz domena funkcije.
2. Vrši se selekcija jedinki na osnovu funkcije evaluacije. Jedinke koje su prošle izbor dobijaju mogućnost da se razmnožavaju.
3. Razmnožavanje se vrši tako što se slučajno izabere par jedinki, i od njih proizilaze nove dve, tako što se ukrštaju njihovi geni.
4. Neke od jedinki u novodobijenoj generaciji mutiraju
5. Ako nije ispunjen dovoljan broj generacija, vraćamo se na korak 2.
6. Program vraća najbolje rešenje, to je jedinka sa optimalnom (najmanjom ili najvećom) vrednošću funkcije

Proces selekcije je neophodan da bi se populacija usmerila ka ekstremima funkcije. Odabirom najkvalitetnijih jedinki, povećava se šansa da će njihovim ukrštanjem takođe nastati kvalitetne jedinke. Postoji više načina da se izvrši selekcija. Najkorišćenije su elitistička i rulet. Prilikom elitističkog izbora samo najbolji preživljavaju. Dakle, odredi se koliko jedinki treba da preživi i svojim genima učestvuje u narednoj generaciji. Taj odnos može biti jedna polovina, mada nije obavezno. Prednost kod ovakvog izbora je što se biraju baš najbolje jedinke i dobre su šanse za brz napredak populacije. Međutim, mana je što se na ovaj način vrlo brzo, posle samo nekoliko generacija može doći do prilično jednolične populacije, koja bi se zaustavila na lokalnom optimumu, umesto da se približava globalnom. Zato treba biti oprezan sa elitističkim modelom. Kod rulet modela, kao što mu naziv govori, dosta zavisi od slučajnosti, tj. verovatnoće. Naime, svaka jedinka ima šanse da preživi, s tim što su šanse proporcionalne njihovim sposobnostima, tj. vrednosti funkcije evaluacije. Ovakav model nema problema sa ,,zakucavanjem" u lokalnim ekstremima ekstremima, mada je zahtevniji jer traži veći broj iteracija ne bi li se došlo do najboljeg rešenja (videti [1]). Verovatno najbolji model je kada se ukombinuju elitistički i rulet modeli. To se radi tako što se prvih nekoliko jedinki (može i samo jedna) ostavi da prežive, a na ostale primeni rulet eliminacija. Metoda turnira je takođe poznata. Slučajno se izabere k jedinki i od njih najbolja prolazi dalje. Takav postupak se ponavlja sve dok se ne popuni zadovoljavajući broj jedinki potrebnih za razmnožavanje. Još jedna zanimljiva metoda je metoda sa pragom (*eng. threshold*). U tom slučaju, za preživljavanje jedinke neophodno je da njena vrednost bude bolja od određenog praga. Tada se može desiti da posle selekcije ostane vrlo malo jedinki, pa je u tom slučaju neophodno dodavati u populaciju nove slučajno generisane jedinke, ne bi li popunile potreban broj. Tako će biti u početku, a u kasnijim iteracijama, biće dovoljno dobrih jedinki koje će biti bolje od praga, tako da se i prag postepeno mora približavati optimumu, kako bi uopšte imalo smisla držati ga.

Nakon selekcije, jedinke koje su preživele treba uparivati kako bi se dobile nove jedinke. Dve jedinke koje nazivamo roditeljima se udruže da bi kreirale dve nove jedinke, koje zovemo decom. Jasno je da je to proces razmnožavanja, i da bi se moglo razumeti kako razmnožavanje funkcioniše, neophodno je prvo shvatiti kako su geni predstavljeni u jedinki. Svakako će novi gen biti na neki način mešavina gena dva roditelja. Između parametara koji oslikavaju jedinku i gena mora da postoji bijekcija, jer na osnovu parametara dobijamo gen, a kasnije na osnovu gena dobijamo parametre. Prvi, i najčešće korišćeni tip gena jeste bitovski gen. Bitovski gen se sastoji od niza nula i jedinica. Da bismo od niza parametara dobili bitovski gen, potrebno je da se vrednosti tih parametara prebace u binarni sistem i poređaju jedna iza druge i dobiće se pravi binarni niz. To bi bio relativno jednostavan način da se iz niza parametara dobije gen. Najbolje se vidi na sledećem primeru. Imamo jedinku od tri parametara (3, 7, 5). Kada ih prevedemo u binarni sistem (tako što za svaki od parametara koristimo 3 bita, dobićemo (**011**, **111**,

101), a sami binarni gen bi bio **011111101**. Autor ovog rada češće primenjuje drugu vrstu prezentacije bitovskih gena, koju možemo nazvati mešanje. Način na koji bi se od ova tri broja kreirao binarni gen je sličan mešanju (*eng. merging*) po tome što se redom stavlja prvi znak prve reči, potom prvi znak druge reči, pa prvi znak treće reči, i tako redom. Tako bi prethodni primer ispao **011110111**. Ovakav način prezentacije je dobar u sprezi sa određenim metodama ukrštanja gena, što proističe iz teoreme o gradivnim blokovima, koja je fundamentalna teorema genetskih algoritama i koja će kasnije biti objašnjena. Kada je binarna reprezentacija gena u pitanju, potrebno je najpre odrediti koliko bitova čini jedan gen. Onda se odredi koliko bitova pridružujemo kom parametru. Obično se znaju donja i gornja granica parametra. Primera radi, neka su one *min* i *max*, a parametar je predstavljen sa k bitova. Sa *k* bitova postojaće mogućnost za 2^k različitih vrednosti. Dati binarni broj određen sa *k* bitova uzeće vrednost b iz skupa {0,1,2,... 2^k -1}, tako da će u tom slučaju, vrednost parametra biti

$$min + \frac{(max - min) \cdot b}{2^k}$$

i na taj način će ravnomerno biti pokrivene vrednosti iz datog intervala. Tako je moguće koristiti i realne brojeve kao parametre. Očigledno, veći broj bitova po parametru omogućava i veću preciznost prilikom traženja rešenja, ali istovremeno i usporava i otežava pretragu zahtevajući veći broj generacija i više jedinki u populaciji, pa samim tim i duže izračunavanje i veću iskorišćenost resursa računara.

Sada se možemo vratiti na proces uparivanja. Od dve jedinke koje imamo u trenutnoj generaciji, potrebno je formirati dve nove koje će ih naslediti u sledećoj. Postupak kojim se to izvodi se naziva ukrštanje ili rekombinacija. Kada se izaberu dve jedinke koje predstavljaju roditelje, određuje se najpre da li će oni uopšte imati decu. To se bira slučajno, obično sa verovatnoćom od 0.5 do 0.8 da će imati decu. Ako nemaju, onda upravo te dve jedinke nastavljaju u narednoj generaciji, a ako imaju onda se vrši ukrštanje gena i proizvode dve potpuno nove jedinke. Kod bitovskih gena, koristi se nekoliko različitih vrsta ukrštanja. Verovatno najkorišćenije je ukrštanje sa jednom tačkom. Pod pretpostavkom da geni imaju istu dužinu (ako nemaju, jedan se može dopuniti nulama) na slučajan način izabere se mesto koje će prepoloviti nizove na dva dela. Izvrši se izmena kao na sledećem primeru.

Prvi gen: 1101001|0100
Drugi gen: 1010010|1000

Novi geni: 1101001|1000
1010010|0100

Uspravna crta označava tačku koja razdvaja dva dela gena.

Često se koristi i uniformno ukrštanje, gde se na svakom mestu u nizu može naslediti bit ili od jednog roditelja ili od drugog, kao na sledećem primeru. Ovaj model može biti dobar kada bitovi nisu preterano nezavisni, jer ne čuva dobar raspored susednih bitova.

Prvi gen: 11010010100
Drugi gen: 10100101000

Novi geni: 11100111000
10010000100

Pored ovih tipova ukrštanja, koristi se i ukrštanje sa više tačaka, kod kog se izabere više mesta za ukrštanje i na svakom od njih se promeni roditelj.

U genetskim algoritmima, kao što je napisano u koraku 4. postupka, nakon uparivanja neke od jedinki mutiraju. Mutacija je mala promena nekog od gena kako bi se postiglo uvođenje novog genetskog materijala u populaciju. Sa verovatnoćom reda veličine 0.05 svaka od jedinki može da se izmeni na nekoj poziciji u svom genu, jednostavnom inverzijom.

Druga vrsta gena koja se koristi u ovom radu jesu geni sa realnim brojevima. Jedna bitna prednost ovih gena u poređenju sa najkorišćenijim bit-genima je što realni brojevi mogu maksimalno da iskoriste preciznost mašine prilikom izračunavanja parametara, i tako dobije preciznije rešenje. Nekada je i logičnije uzeti realne vrednosti kao gene kada imamo realne parametre, umesto komplikovanijeg pretvaranja bit-gena u realne brojeve i nazad. S obzirom na korišćenje računarskih operacija sa pokretnim zarezom, ovi metodi zahtevaju i više računarskih resursa prilikom ukrštanja i mutacije. S obzirom na drugačiji tip gena, i rekombinacija i mutacija će se koristiti na drugačiji način. Počinje se od sličnih ukrštanja kao i kod bitovskih gena, tako što se bira jedna ili više tačaka koje će biti tačke ukrštanja i pre i posle njih će različiti naslednici dobiti gene različitih roditelja.

Prvi gen:	1.23	25.7	|	3.11	4.05	6.1	| 5.4 6.4
Drugi gen	2.45	3.3	|	0.78	5.33	3.2	| 4.9 3.33
Novi geni	1.23	25.7	|	0.78	5.33	3.2	| 5.4 6.4
	2.45	3.3	|	3.11	4.05	6.1	| 4.9 3.33

Primer sa dve tačke ukrštanja

Ono što je kod ovih vrsta ukrštanja očigledan problem je da tu ne dolazi do promene genetskog materijala. Oni realni brojevi koji su generisani na početku algoritma ostaće i do kraja i tako se praktično onemogućuje prava optimizacija parametara (videti [2]). Zbog toga se kod ukrštanja primenjuju ove metode samo malo modifikovane. Uvodi se parametar β, koji se slučajno bira između 0 i 1. Tako, ako prvi gen ima parametre a1, a2, a3, ... an, a drugi b1, b2, b3, ..., bn i ako se koristi ukrštanje sa jednom tačkom ukrštanja, koja je na mestu k, novi geni ce se dobiti linearnom kombinacijom prva dva

Prvi gen	$a_1 \;\; a_2 \;\; a_3 \ldots a_k \;\; a_{k+1} \ldots a_n$
Drugi gen	$b_1 \;\; b_2 \;\; b_3 \ldots b_k \;\; b_{k+1} \ldots b_n$

Novi geni:

$$a_1 \cdot \beta + b_1 \cdot (1-\beta) \quad a_2 \cdot \beta + b_2 \cdot (1-\beta) \ldots a_k \cdot \beta + b_k \cdot (1-\beta) \quad a_{k+1} \cdot (1-\beta) + b_{k+1} \cdot \beta \ldots a_n \cdot (1-\beta) + b_n \cdot \beta$$

i

$$a_1 \cdot (1-\beta) + b_1 \cdot \beta \quad a_2 \cdot (1-\beta) + b_2 \cdot \beta \ldots a_k \cdot (1-\beta) + b_k \cdot \beta \quad a_{k+1} \cdot \beta + b_{k+1} \cdot (1-\beta) \ldots a_n \cdot \beta + b_n \cdot (1-\beta)$$

Svakako, kod ove vrste gena, i mutacija će biti drugačija. Jedna varijanta mutacije bi mogla biti da se slučajno izabere element niza (ili više njih) i zameni novim slučajno izabranim brojem iz odgovarajućeg opsega. Takva mutacija može biti dobra, međutim ona donosi apsolutno novi element u niz koji nema nikakve veze sa elementom koji je bio na njegovom mestu ranije. Stoga, još jedna varijanta mutacije je da se umesto broja koji

mutira upiše novi broj koji je prema Gausovoj raspodeli u njegovoj okolini. To je jako zanimljiv način mutiranja kod koga treba biti oprezan oko parametra raspodele σ koji je standardna devijacija normalne raspodele. Ako bude preveliki, onda novi broj može izaći izvan svojih granica, a ako je premali onda ni mutacija neće imati efekta, jer će novi broj biti previše bliz starom. Stoga, ukoliko se dobro ne razmisli oko izbora tog parametra, verovatno je bolje koristiti klasičnu zamenu.

Od značajnijih vrsta gena još treba pomenuti i gene sa permutacijama, gene sastavljene od znakova i adaptivne gene.

Genetski algoritmi su, prema prikazu nekoliko vrsta selekcija, mutacija i ukrštanja relativno jednostavni za implementaciju, ne zahtevaju komplikovane algoritme i strukture podataka. Pa ipak, daju dobre rezultate u velikom broju problema sa optimizacijama i pretragama. Kao što se može zaključiti iz svih dosadašnjih primera, kao i varijanti selekcije, mutacije i ukrštanja, praktično svaki element genetskog algoritma uključuje i funkciju verovatnoće, tj. generator slučajnih brojeva. Sa tog aspekta, genetski algoritmi su stohastički po svojoj prirodi i ne mogu sa sigurnošću da garantuju uspeh. Bez obzira na to, postoji hipoteza o gradivnim blokovima čiji odgovor daje šema teorema, kako se još naziva. Ona objašnjava zbog čega su genetski algoritmi dobri kod optimizacija i pretraga i zašto uopšte sistem operatora kao što su selekcija, uparivanje, ukrštanje i mutacija funkcioniše i daje dobre rezultate. Šema ili blok je izraz koji je Holand definisao i označava niz ili string znakova iz skupa {0,1,*}, gde je * džoker znak i može označavati i 0 i 1. Ako se gen, odnosno hromozom sastoji od nula i jedinica i ima dužinu 8, jedna šema ili blok u okviru tog hromozoma bi bio niz 01****11. Za gene dužine *N*, ukupan broj različitih gena je 2^N a šema 3^N. Jedna šema, sa *K* zvezdica pokriva 2^K gena. Tako, upravo navedena šema 01****11 bi pokrivala 2^4 =16 gena. To su:

01**0000**11	01**1000**11
01**0001**11	01**1001**11
01**0010**11	01**1010**11
01**0011**11	01**1011**11
01**0100**11	01**1100**11
01**0101**11	01**1101**11
01**0110**11	01**1110**11
01**0111**11	01**1111**11

Gde su debljim slovima označeni znakovi koje pokriva džoker znak. Definišu se dve osobine šeme *S*, to su njena dužina i red. Red šeme se označava sa o(*S*) i to je broj fiksiranih znakova (različitih od zvezdice) u šemi, u datom primeru je 4. Što je veći red šeme, to je ona određenija. Dužina, koju označavamo sa δ(*S*) je broj tačaka ukrštanja između prvog i poslednjeg fiksiranog znaka i u ovom slučaju je 7. Na osnovu definicije, zaključuje se da je to broj tačaka kada bi se u toku ukrštanja mogla rasturiti šema prilikom ukrštanja sa jednom tačkom. Prema teoremi o šemama, šeme sa manjim dužinama, a boljom funkcijom evaluacije imaju šanse da budu rasprostranjene u većem broju jedinki i tako dalje.

S obzirom da u prirodi opstaju jedinke koje najbolje prilagode svoj genom, slična je situacija i ovde. Vrši se optimizacija parametara i preživljavaju jedinke čiji parametri daju najbolje rezultate. Slično kao u realnom svetu, gde sposobniji imaju veće šanse za opstanak, i u genetskim algoritmima jedinke koje funkcija evaluacije bolje oceni imaju veće šanse da budu i u narednoj generaciji populacije.

2.2 Tehnike udruživanja

Udruživanje (eng. boosting) je relativno nova metoda, ili bolje rečeno tip metoda. *Boost* je homonim i na engleskom može da znači udruživanje ili pojačavanje, tako da ove metode možemo zvati metodama udruživanja ili metodama pojačavanja. Ova familija metoda se pojavila u poslednjih petnaestak godina i predstavili su je istraživači sa američkih univerziteta, *Robert Schapire* sa Prinstona i *Yoav Freund* sa UCSD (*University of California, San Diego*). Osnovna svrhva ovih algoritama pojačavanja je rešavanje problema klasifikacije, kada je potrebno odrediti da li neki objekat pripada određenoj grupi objekata. Međutim, uz malu doradu, ovi algoritmi se u opštem slučaju mogu primeniti i za uopštene probleme mašinskog učenja, optimizacije, pa i odlučivanja. Ovi algoritmi su zapravo odgovor na *Boosting* hipotezu, koju je 1988. postavio *Michael Kerns*, a to je pitanje može li se više jednostavnih klasifikatora udružiti da zajedno formiraju jednog dobrog klasifikatora. Otuda i logika da se metoda zove metoda udruživanja. Odgovor na ovu hipotezu je potvrdan i sada već postoji dosta metoda, kao što su *LPBoost*, *TotalBoost*, *BrownBoost*, *MadaBoost*, *LogitBoost*, a verovatno najpoznatiji i najkorišćeniji među njima je *AdaBoost*. *AdaBoost*, je jako značajan zato što je to prvi algoritam koji je bio adaptivan, tj. mogao je da menja svoje ponašanje u zavisnosti od ponašanja prostih klasifikatora. Algoritmi udruživanja mogu jako dobro da se primene u sprezi sa svim algoritmima učenja/treniranja, jer poboljšaju njihov efekat. Kod problema klasifikacije, potrebno je da algoritam učenja daje dobre rezultate u tek nešto više od polovine slučajeva i boosting algoritam će to moći dobro da iskoristi. S obzirom da se algoritam udruživanja koristi kao meta-algoritam, radi pojačavanja nekog već postojećeg algoritma, verovatno ne bi bilo pogrešno ni da ga zovemo algoritam pojačavanja. Ipak, u ovom radu, koristiće se izraz algoritam udruživanja, jer se on odnosi na pravu suštinu i na pitanje koje je Kerns postavio, a to je da se udruženim snagama dobija bolji rezultat. Takođe, udruživanje samo po sebi podrazumeva pojačanje, tako da i zbog toga, algoritmi udruživanja zvuči kao logičniji naziv.

Kao što smo rekli, kod tehnika udruživanja, potrebno je da postoji izvestan algoritam učenja koji je u sprezi sa algoritmom udruživanja. Učenje treba da obezbedi takozvane jednostavne klasifikatore, od kojih će svaki predstavljati na neki način određenu karakteristiku koja je pronađena u datoj grupi objekata. Kao i kod svakog treniranja, potrebno je pripremiti referentni skup podataka prema kome se vrši trening. Da bi treniranje bilo uspešno potrebno je u tom skupu imati veliki broj objekata koji pripadaju toj grupi (pozitivni primeri), ali možda još i veći broj objekata koji nisu iz te grupe (negativni). Svim primerima za početak pridajemo podjednaku važnost, koju ćemo nazvati težina primera, a koja će se kasnije menjati. Udruženo obučavanje počinje tako što osnovni algoritam za obučavanje kreira jednostavan klasifikator, koji je najbolji koji možemo dobiti za date primere. Onim primerima koje je jednostavni klasifikator dobro ocenio ćemo smanjiti važnost, dok oni primeri koji su pogrešno ocenjeni, dobiće povećanje težine, pa će u narednim rundama više vredeti, što nam govori da osnovni algoritam obučavanja treba da proizvede klasifikator koji će se bolje ponašati upravo sa tim primerima. Na taj način, algoritam udruživanja forsira osnovni algoritam obučavanja da se fokusira više na primere koje je loše klasifikovao u prethodnim rundama (videti 3). Dakle, *ada boost* je postupak kod koga se sukcesivno primenjuje osnovni algoritam učenja, i kod

koga trening novog klasifikatora u svakoj rundi zavisi od uspešnosti prethodnih klasifikatora. Da bi detalji bili jasniji, sledi formalniji opis postupka.

Jednostavan klasifikator je funkcija h() koja za dati ulaz x, daje izlaz h(x) $\in$ {-1,1} i čiji je procenat uspešnosti nešto bolji od običnog nagađanja. Primenjujući osnovni algoritam za učenje sukcesivno, dobićemo niz jednostavnih klasifikatora, h_1, h_2, ... h_m, koji će zajedno biti deo kvalitetnog klasifikatora. Svaki klasifikator h_j će imati svoj koeficijent verodostojnosti αj pa će se konačna odluka donositi putem težinskog glasanja (*eng majority voting*)

$$h_{strong} = \text{sign}(\sum a_j \cdot h_j(x)).$$

Težine svakog od jednostavnih klasifikatora se određuju algoritmom udruživanja. Ako imamo *n* primera, neka su oni x_1, x_2, ... x_n, i njihovi očekivani izlazi y_1, y_2, ... y_n. Važnost svakog od njih ćemo modelovati realnim brojem w_i, i u početku će svi biti jednaki, $1/n$. Dakle, u samom algoritmu imamo dve vrste težina, jedno su težine koje pripisujemo klasifikatorima, a drugo su težine povezane sa primerima, i govore nam koliko je koji primer bitan u datom trenutku procesa. Pošto treba istrenirati *m* klasifikatora, postupak udruživanja će imati *m* rundi (ciklusa). U svakoj rundi se najpre istrenira jednostavni klasifikator na osnovu težina primera. Potom se izračuna kumulativna greška klasifikatora kao suma

$error_j = \sum w_i$, za one i gde je $y_i \neq h_j(x_i)$

Težina klasifikatora hj se računa kao

$$\alpha j = \log\left(\frac{1-error_j}{error_j}\right)$$

Treba primetiti da će $error_j$ biti manje od 1/2, što proističe direktno iz osobine jednostavnih klasifikatora da su uspešniji od 50%. Pošto je $error_j$ < 1/2, to govori da će αj biti veće što je $error_j$ manje, što je i logično jer to govori da će u krajnjem težinskom glasanju više vredeti glas klasifikatora koji su imali manju grešku. Nakon izračunavanja težine klasifikatora, potrebno je ažurirati i težine primera. Kao što smo rekli, primerima koji su dobro klasifikovani težine se smanjuju, dok se loše klasifikovanim povećavaju.

$$w_i = w_i \cdot \exp(-\alpha_j \cdot y_i \cdot h_j(x_i))$$

Da bi niz *w* ostao raspodela, izvrši se normalizacija i na taj način se završi jedna runda. Ukupan broj rundi može biti fiksan, a može i trajati dok se ne uspostavi zadovoljavajuće nizak nivo greške rezultujućeg udruženog klasifikatora. Kao izlaz algoritma vraća se niz klasifikatora h_j, kao i njihove težine α_j.

Algoritam se može predstaviti sledećim pseudokodom:

1. Postavljanje težina w_i na inicijalnu vrednost $w_i = 1/N$, za svako i = 1,*N.*
2. j =0
3. Nalaženje najboljeg jednostavnog klasifikatora h_j prema težinama w_i
4. Računanje greške $error_j$ klasifikatora h_j prema formuli
   $$error_j = \sum w_i$$
   za one i za koje je klasifikator h_j loše okarakterisao
5. $\alpha j = \log\left(\frac{1-error_j}{error_j}\right)$ je relevantnost klasifikatora h_j
6. Ažuriranje vrednosti w_i
   $w_i = w_i \cdot \exp(-a_j)$ ako je test i dobro okarakterisan

$w_i = w_i \cdot \exp(a_j)$ ako je loše klasifikovan

7. Ako tačnost nije zadovoljavajuća, povećati j za 1 i vratiti se na korak broj 3
8. konačni klasifikator su dva niza h i a, klasifikatori i njihove važnosti

2.3 Mašinsko viđenje i OCR

Mašinsko (ili računarsko) viđenje je oblast nauke koja se bavi mašinama koje mogu "gledati". Svakako, mašine ne mogu gledati na način na koji čovek vidi, ali mašinsko gledanje se odnosi na obradu vizualnih signala koji pristižu na ulaz sa video kamere ili foto senzora. Digitalna slika se pretvara u simbolički opis koji je uređaju razumljiv i koji ovaj može upotrebiti. Sa stanovišta povezanosti sa ostalim naukama, može se reći da je mašinsko viđenje usko povezano sa nekolicinom drugih nauka, šta više, mašinsko viđenje predstavlja kombinaciju nauka kao što su: Veštačka inteligencija, Mašinsko učenje, Robotika, Obrada signala, Optika, Geometrija...

Napretkom tehnologije i dobijanjem sve kvalitetnijeg hardvera u smislu kamera, računarskih procesora i memorije, kao i razvojem tehnika veštačke inteligencije postavljeni omogućen je značan pomak u ovoj oblasti, pa ipak, računarsko viđenje se smatra za dosta mladu oblast. Ozbiljnija istraživanja na ovu temu krenula su kasnih 70-tih godina prošlog veka kada je počeo da se pojavljuje hardver sa mogućnostima snimanja i obrade vizuelnih podataka. U ovom trenutku ne postoji opšte poznata generalna strategija koja govori o tome kako se kompjutersko viđenje treba konstruisati u opštem slučaju. Postoje rešenja vrlo specifičnih problema sa svojim posebnim karakteristikama, koji se teško mogu generalizovati. U većini slučajeva, mašine koje imaju ugrađen sistem za gledanje su u stanju da urade određeni zadatak za koji su preprogramirane, ali u poslednje vreme nisu retki ni slučajevi gde mašina može da nauči nešto.

Glavne primene mašinskog viđenja u ovom trenutku su u oblasti medicine. Obično se obrađuju slike dobijene rendgenski, tomografijomi, mikroskopski, ultrazvukom i na osnovu njih se može izvršiti dijagnoza. Može se utvrditi postojanje tumora, arteroskleroze i drugih malignih promena. Moguće je vršiti merenja, kao što su dimenzije organa, krvotoka itd. Na taj način se vrši napredak medicine, tako što se kompjuterski dobija dosta novih podataka. Uz primene u medicini, česte su i vojne aplikacije, pogotovo kod navođenja raketa. Kao najnovija primena, pojavljuju se vozila sa samostalnim upravljanjem. Jedno od njih je i *Rover* koji trenutno istražuje Mars.

OCR je skraćenica (*eng. Optical Character Recognition*) koja se odnosi na automatsko prevođenje slika teksta štampanog ili pisanog rukom, u tekstualni format upotrebljiv na računaru. OCR je jedn od podoblasti mašinskog viđenja. Iako se OCR kao originalni termin koristio da označi prepoznavanje znakova optičkim tehnikama kao što su sočiva i ogledala, a ne digitalnim putem (preko skenera i digitalnom obradom slike), danas je malo ili gotovo da uopšte nema primena čisto optičkog prepoznavanja. Tako je termin OCR danas u širokoj upotrebi i odnosi se upravo na digitalno prepoznavanje znakova.

To je verovatno najlakši način da se starije knjige koje nisu pisane na računaru prevedu u standardni tekstualni format. Pogodnosti tekstualnog formata su vrlo značajne, jer

omogućuju da se pronađe određena reč u dokumentu, zatim pretragu dokumenata po ključnim rečima, kako na Internetu, tako i na kućnom računaru, pa prebacivanje određenog dela teksta sa ciljem citiranja, itd. Dok se OCR programi nisu pojavili, knjige su mogle biti prebacivanje na računar ili prekucavanjem, što bi zahtevalo dosta vremena, ili samo skeniranjem, ali na taj način ne bi se dobio tekstualni sadržaj dokumenta, već samo slika. Upotrebom kvalitetnog OCR softvera, cela procedura je svedena na skeniranje, a na kraju se dobije dokument u tekstualnom formatu. S obzirom da ni OCR programi ni proces skeniranja nisu savršeni, često je potrebno vršiti sitne ispravke u dobijenom tekstu. OCR programi se ne koriste samo za prebacivanje teksta sa papira na računar, već imaju i druge primene. Jedna od njih je prepoznavanje rukopisa na uređajima kao što su PDA (*eng. Personal Digital Asssistant*) ili *Tablet* PC jer je u tim uređajima jedan od bitnijih ulaznih uređaja ekran sa senzorima po kome se može pisati olovkom.

Trenutno postoji dosta OCR programa. Najpoznatiji su *ABBYY FineReader*, *GOCR*, *Novo Dynamics*, *Ocrad*, *Ocropus*, *OmniPage*... Što se njihove uspešnosti tiče, ona zavisi od toga kakav je tip teksta u dokumentu. Problem sa štampanim slovima engleskog alfabeta se smatra rešenim, i OCR programi daju uspešnost od oko 99%. Premda postoje programi koji daju i više od 99% tačnosti čak i tada je potrebno da čovek izvrši korekcije kako bi se dobio tačan tekst. Mada je u početku bilo problema napraviti program koji će raspoznavati različite fontove, sada je to uobičajena mogućnost svakog OCR programa. Neki programi čak uspevaju i da naprave prilično vernu rekonstrukciju stranice koja sadrži i slike, tabele, što OCR programima daje novu dimenziju. Pa ipak, tekst pisan rukom, uključujući i pisana i štampana slova, i dalje je predmet istraživanja. Kao što je rečeno, PDA i *Tablet* PC uređaji poseduju sistem za unos podataka koje čovek piše rukom, međutim, oni osim same slike, koriste i redosled, smer i brzinu pisanja, jer imaju tu mogućnost. U takvoj situaciji, korisnik može da bude uvežban tako da piše slova na specifičan način, povlači linije u određenom redosledu i očekivanom smeru. Imajući sve te olakšice u vidu, prepoznavanje rukopisa je i dalje problem, koji nije rešen. Na čistom papiru, kod precizno pisanih slova, najveća preciznost može doći do 80%-90%, tako da to ostavlja veliki broj grešaka i potrebu da se one ispravljaju. Takođe, prepoznavanje štampanog teksta pisama različitih od engleskog alfabeta još nije skroz razrešeno, pogotovo kada su u pitanju pisma sa velikim brojem karaktera.

3 Postupci primenjeni u ovom radu

S obzirom na sve navedeno, problem koji se obrađuje ovim radom zalazi u polje veštačke inteligencije, digitalne obrade slike, uklanjanja šuma, mašinskog viđenja i prepoznavanje znakova. U narednom odeljku biće reči o samoj implementaciji, metodama i algoritmima koji su korišćeni, softveru, pomoćnim alatima kreiranim za ovu priliku i rezultatima koji su postignuti. Kao što je već navedeno, projekat se sastoji iz nekoliko etapa. Preuzimanje velikog broja CAPTCHA sličica sa Interneta, zatim kreiranje algoritma kojim se razdvajaju korisne informacije iz slike, odnosno izdvajanje slova iz pozadine i na kraju, kreiranje programa za prepoznavanje pojedinačnih znakova. Kompletan programski kod koji je korišćen u izradi je lično delo autora i pisan je u programskom jeziku Java.

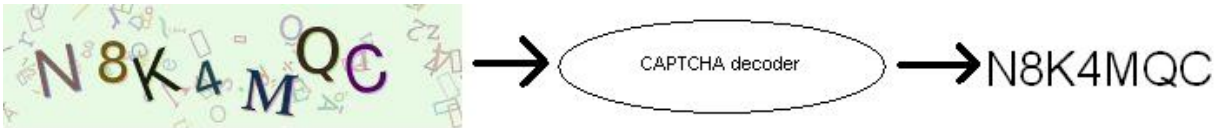
3.1 Prikupljanje slika

Kod svakog algoritma za učenje, neophodno je pripremiti materijal, odnosno primere koji će služiti kao model u postupku obučavanja. U slučaju CAPTCHA sličica, to bi se odnosilo na dovlačenje velikog broja slika preko Interneta. U zavisnosti od situacije, CAPTCHA slike se koriste, kao što je već navedeno, da bi se onemogućila automatska registracija, slanje automatskih poruka, dovlačenje velike količine fajlova. Slike se mogu prikupiti ručno ili automatski. Ručno preuzimanje slika bi podrazumevalo konstantno ponavljanje akcija koje su potrebne, kao što je pokušaj registracije ili slanje poruke na stranicu sa vestima ili forum. Taj postupak može biti dosta mučan kada je potrebno pripremiti veliki broj primera za obučavanje.

Pošto je za ovu priliku korišćen javno dostupan CAPTCHA sistem B2evo, preuzeto je sa njihove stranice oko 5000 slika, od kojih je 2000 bilo korišćeno u obučavanju (učenju).

3.2 Obeležavanje primera

Da bi obučavanje mašine bilo omogućeno na bilo koji način, potrebno je definisati model ponašanja prema kome želimo da naučimo mašinu. To može značiti određivanje ulaznih i izlaznih vrednosti funkcije koju želimo modelovati. U slučaju prepoznavanjaznakova na slikama, par ulazni parametar će biti jedna slika, a izlazni niz slova koji odgovaraju slici 3.1.



Slika 3.1 – CAPTCHA dekoder

Pošto su ulazni parametri već dovučeni, potrebno je generisati izlazne. Izlazni parametri na žalost ne mogu biti automatski generisani, jer bi to već značilo posedovanje CAPTCHA dekodera. U principu, obučavanje je moguće kontrolisati od strane računara, ali to znači da bi računar morao da ima informacije o izlaznim informacijama, što u ovom slučaju nemamo. Jedino što je moguće automatizovati u ovom slučaju, to je automatizacija procesa obeležavanja slika, za šta je kreiran program *TestLabeling*. Za svaku sliku, korisnik može da obeleži željeni rezultat, tj. otkuca slova koja su na slici, i pređe na sledeću sliku u datom folderu. Program *TestLabeling* sve podatke čuva u istom folderu, tako što se za svaku sličicu otvori fajl sa istim nazivom kao i slika i ekstenzijom txt. Tako dobijeni podaci se koriste u obučavanu, i kod izdvajanja znakova od pozadine i kod programa koji prepoznaje znakove.

3.3 Izdvajanje znakova iz pozadine

Prilikom svake obrade signala, vrši se razdvajanje značajnih informacija od nevažnih (šuma). U ovom slučaju, želja nam je da od pozadine odvojimo znake. Linije u pozadini su manji karakteri i druge geometrijske figure, rotirani svaki pod nekim uglom, često isprepleteni među sobom, sve u cilju ometanja dekodiranja CAPTCHA slike. Na prvi pogled, stvar koja olakšava razdvajanje znakova iz slika je intenzitet boje. Na slici koja je ispod, a na kojoj je slovo Y uveličano, se vidi da su boje koje su u slovu dosta jače od ovih koje su pored i da se na taj način može izvršiti filtracija. Pored njega slovo B ukazuje i na poteškoće prilikom eliminisanja pozadine. Pošto je slika u jpg formatu, iako boje daju informaciju, na primeru slova B može se videti da je veliki broj tačaka (piksela) koji sačinjavaju slovo B jako bledo, što najviše proističe iz JPEG kompresije. JPEG kompresija koristi manu ljudskog oka i pravi vrlo fine prelaze između boja susednih tačaka koje nisu uočljive, ali se primećuju kada je slika uveličana i naravno, kada kompjuter obrađuje sliku. Ono što je u ovom slučaju problem, upravo zbog finih prelaza jeste odrediti granicu kada je neki piksel u pozadini, a kada sačinjava karakter, dakle u kom trenutku se postavlja granica između slova i pozadine. Jasno je da se radi o intenzitetu svetlosti, ali ne baš i na koj način. Sledeća stvar koja postavlja novi problem jeste to što znakovi nekada mogu biti spojeni, a neki znakovi se sastoje iz više delova, npr. '?' iz dva ili '%' čak iz tri. Znači, imamo da su u jednoj povezanoj figuri sadržana dva različita znaka, a takođe da dve ili više figura čini jedan znak. Treba najpre naći način kako razrešiti prvi problem i kada, da li u ovom delu, gde se izdvajaju znakovi iz pozadine, ili u delu gde se prepoznaju pojedinačni znakovi, s tim što bi se u tom slučaju moralo vršiti prepoznavanje dva znaka sa jedne slike. Drugi problem (sastavljanje znaka iz više delova) treba rešiti u trenutku samog izdvajanja znakova iz pozadine, da se delovi ne bi smatrali posebnim znakovima. Kao rešenje ovog problema samo se nameće sličnost u boji i blizina samih figura.

Slika 3.2 – Uveličane slike znakova koje pokazuju koliko JPG kompresija pravi fine prelaze između boja

3.4 Opis Algoritma za izdvajanje znakova

U ovom odeljku konkretnije će biti opisan postupak kojim se dolazi do izdvajanja znakova iz pozadine, kao i razdvajanja povezanih znakova. Algoritam se oslanja na genetski algoritam kojim se podešavaju parametri kako bi se postigao najbolji rezultat.

Prvi deo algoritma je filtriranje. Kada se prođe kroz celu matricu tačaka (piksela), za svaku tačku se, prema intenzitetu boje odredi može li ona biti deo nekog znaka ili ne. Samo boje ispod određene osvetljenosti se uzimaju u razmatranje. Da bi se osvetljenost neke boje izračunala, koristi se drugačiji prostor boja od klasičnog RGB (*eng. red, green, blue*), već HSV (hue, saturation, value) gde se value odnosi na osvetljenost boje. Boje sa manjim svetlom su jarke, baš kao što su ove kojima su obojeni znaci. Da bi se ovo izračunalo, potrebno je odrediti prag (*eng. threshold*), tj. minimalnu vrednost intenziteta boje tako da tačka može biti uključena u obradu. Parametar nazivamo `colorIntensityThreshold`. Samo tačke koje imaju intenzitet veći od `colorIntensityThreshold` učestvuju u znacima. Kada se to završi, od tačaka koje su preostale treba izdvojiti nekoliko manjih slika, za svaki karakter po jednu sliku.

Treba, dakle, pronaći koje tačke su povezane i nalaze se u istom znaku. Za to se koristi dobro poznati BFS (*eng. Breadth First Search*, pretraga prvo u širinu) algoritam (videti [5]) kojim se mogu naći povezani čvorovi u grafu. U slučaju sa slikom, čvorovi grafa su tačke koje su prošle filtraciju po intenzitetu, a ivice između dva čvora postoje ako su date dve tačke susedne, bilo po horizontali, vertikali ili dijagonali. Tako je svaki čvor povezan sa najviše 8 susednih čvorova. Za dati novonastali graf, uz pomoć BFS algoritma pronađu se komponente povezanosti koje će u dobrom delu slučajeva same po sebi biti gotovi znaci. Međutim, kao što smo rekli, postoji problem kako razdvojiti dva povezana znaka, jer oni čine jednu komponentu, kao i rešiti situaciju kada više od jedne komponente čini jedan isti znak. Logično je najpre rešiti prvi problem, i kada budemo sigurni da jedna komponenta pripada samo jednom znaku, odlučiti koje komponente spojiti da sačinjavaju isti znak.

Kako razdvojiti dva znaka koja su povezana, u ovom slučaju rešava se tako što se koristi njihova razlika u boji. Kao što se može primetiti iz primera koji su na slici 1.2, znakovi su različitih boja, tako da je to dobar parametar koji može pomoći prilikom razdvajanja povezanih znakova. Naravno, čoveku nije problem da razlikuje boje na slici, ali slika u JPG

formatu u principu pravi dosta problema, pre svega zbog suptilnih prelaza na mestima gde se znakovi spajaju. Zato se uvode novi parametri. Najpre, prilikom obilaska u širinu, uvodi se novi parametar, `ColorDifThreshold`, koji smanjuje broj ivica u grafu. Od sada, da bi dva čvora (tačke) bila povezana, nije više dovoljno da budu samo susedi, već i da im boje budu dovoljno bliske. To ipak nije dovoljno dobar postupak, jer zbog JPG kompresije, često se dešava da ni tačke koje su bliske i nalaze se u istom znaku nisu tako bliske po boji, što se može i primetiti na slici 3.2 b). Zbog toga se umesto razlike u boji dve susedne tačke, uzima razlika između boje nove potencijalne tačke i srednje vrednosti boje svih tačaka koje su do tog trenutka u datoj komponenti. Za razliku boja, koriste se samo parametri *Hue* i *Saturation*, pošto se osvetljenost već koristila kao filter znakova.

Kada su razdvojene grupe različitih znakova, potrebno je spojiti različite grupe koje formiraju isti znak. To je pre svega potrebno u slučajevima kod znakova kao što su '%' i '?', ali ne samo tu. Često se dešava da sam algoritam razdvajanja razdvoji isti karakter zato što je filter eliminisao neke tačke koje bi mogle pripadati određenom znaku, ili ih je na osnovu boje razdvojio. Spajanje se vrši na osnovu 3 parametra, to su `colorDifCoef`, `euclidDistCoef` i `horDistThreshold`. Parametar `colorDifCoef` govori koliko bliskost boja grupa govori o tome da li one čine isti znak. Boja grupe se računa na osnovu srednje vrednosti boje svih znakova. Parametar `euclidDistCoef` govori koliko bliskost grupa po euklidskom rastojanju znači da su oni u istoj grupi, gde se euklidsko rastojanje grupa računa kao euklidsko rastojanje njihovih centara (težišta). Parametar `horDistThreshold` označava minimalno rastojanje po horizontali, ali ne centara grupa, već najbližih tačaka grupa.

Na kraju, uvodi se i sedmi parametar, to je `groupSizeThreshold` koji predstavlja minimalni broj piksela potreban da bi jedna grupa činila znak. Ako je broj tačaka u grupi manji od `groupSizeThreshold` onda je ta grupa odbačena, jer ne može da da dovoljan broj informacija o znaku. To je u praksi verovatno slučajno ostao mali deo nekog znaka koji nije spojen sa njim ili neki deo pozadine koji eventualno nije isfiltriran u početku.

Sve ukupno, nabrojano je 7 parametara, za koje ne znamo tačne vrednosti. Da bi se one izračunale, koristi se genetski algoritam. Kompletni algoritam za razdvajanje predstavljamo kao jednu funkciju koja dobija kao ulaz sliku, a na izlaz vraća izdvojene karaktere. U procesu obučavanja, želimo da broj znakova koje je algoritam vratio kao rezultat bude što bliži očekivanom broju znakova. Dakle, cilj je izračunati 7 navedenih parametara da se smanji razlika između broja znakova koje je algoritam vratio i očekivanog broja znakova. Pored očekivanog broja znakova, cilj je i da znakovi koji su dobijeni imaju prosečan broj tačaka koji je približan 150, što je empirijskim putem određeno. Funkcija evaluacije se računa po sledećoj formuli:

$$eval = \frac{1}{(1 + error + aveErr)}$$

Cilj je maksimizirati ovu funkciju. Parametar *error* je prosečna vrednost odstupanja broja znakova dobijenih algoritmom od očekivanog broja znakova. *aveErr* označava odstupanje prosečnog broja tačaka po znaku od 150 i računa se kao:

$$aveErr = \frac{|avePix - 150|}{150}$$

gde je *avePix*, prosečan broj tačaka po znaku.

Za ovo obučavanje, koristi se 10 pažljivo odabranih slika. Među njima ima i znakova kao što su '?' i '%', ali i dosta situacija preklapanja. Svakako, bilo bi bolje obučavati na još većem broju slika, međutim, obrada 10-tak slika za jedan skup parametara traje i do 0.05s, pa je zbog velikog broja jedinki i generacija u genetskom algoritmu realnije koristiti manji broj slika. Svakako, paralelni genetski algoritmi bi u ovom sluačaju puno pomogli, jer su izračunavanja apsolutno nezavisna. Vršeno je dosta eksperimentisanja, i sa klasičnim, bitovskim genima, i sa kontinualnim. Rezultati se mogu videti u tabeli 3.1.

Vrsta gena	Vrsta selekcije	Vrsta ukrštanja	Vrsta mutacije	Najbolji rezultat
Bit	Elitistički	Jedna tačka	Klasična	0.8562900
Bit	Rulet	Jedna tačka	Klasična	0.8698365
Bit	Elitistički + rulet	Jedna tačka	Klasična	0.8551672
Bit	Elitistički	Uniformno	Klasična	0.8683546
Bit	Rulet	Uniformno	Klasična	0.8765388
Bit	Elitistički + rulet	Uniformno	Klasična	0.8783915
Realni	Elitistički	Jedna tačka poboljšano	Normalna raspodela	0.7821947
Realni	Rulet	Jedna tačka poboljšano	Normalna raspodela	0.7713611
Realni	Elitistički + rulet	Jedna tačka poboljšano	Normalna raspodela	0.7983637

Tabela 3.1 – poređenje tehnika genetskih algortama kod razdvajanja znakova

Parametri koji su isti korišćeni u svakom procesu obučavanja su broj generacija (100), faktor mutacije (0.03), faktor ukrštanja (0.65) jer su pokazani kao dobri. Kod ovog problema, primetno je da se realni geni znatno lošije pokazuju od bitovskih. Kod bitovskih gena je mala razlika, i uniformno ukrštanje daje nešto bolje rezultate. Što se tiče vrsta selekcije, kada su fiksirani vrsta gena i ukrštanja, ne razlikuju se previše, ali ipak malo bolje rezultate od ostalih daje kombinacija elitističke i reulet selekcije. Sve u svemu, najbolji rezultati dobijeni su bit genima, uniformnim ukrštanjem, klasičnom mutacijom i kombinacijom elitističkog i rulet metoda izbora, preko 0.878. Ta vrednost govori da je *error* najviše 0.1, odnosno jedna greška u 10 slika, što je sasvim dobro. Stoga će se prilikom prepoznavanja koristiti upravo parametri dobijeni na taj način.

3.5 Prepoznavanje pojedinačnih znakova

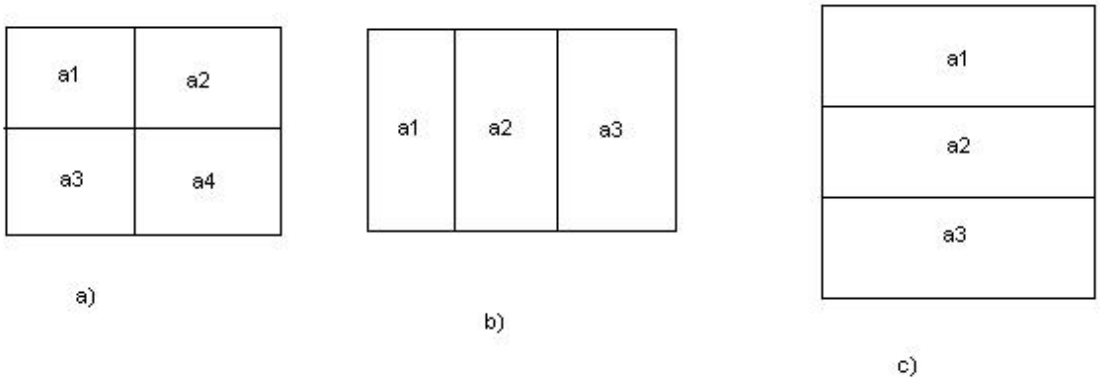
Nakon ekstrakcije znakova iz pozadine, sledeći proces u algoritmu je prepoznavanje pojedinačnih znakova. Prepoznavanje znakova i oblika uopšte se svodi na pronalaženje različitih karakteristika koje su specifične za određeni oblik ili model (*eng. pattern*). Na osnovu toga da li su pronađene te specifičnosti, donosi se odluka da li ili ne dati oblik pripada nekoj grupi oblka. Program koji automatski određuje ovo, naziva se klasifikator. U ovom radu, koristi se *adaboost* sistem klasifikatora, koji je opisan u odeljku 2.2. Za jednostavne klasifikatore, koristi se nekoliko tipova geometrijskih klasifikatora koji će biti opisani u ovom odeljku.

Kao glavna poteškoća kod ove klasifikacije, može se uzeti to da su slova rotirana za određeni ugao. U ovom slučaju, ideja kojom se pristupa ovom problemu jeste takođe rotiranje. Najpre, klasifikator se nauči da prepoznaje znake koji nisu rotirani, odnosno postavljeni su onako kako se štampaju i pišu. Kada postoji klasifikator koji dobro funkcioniše za znake koji nisu rotirani, možemo ga iskoristiti i kod rotiranih znakova. Činjenica je da nije unapred poznato pod kojim uglom je rotiran dati karakter, ali se u ovom slučaju može empirijski zaključiti da je opseg rotiranja od $-\pi/6$ do $\pi/6$. Svaki znak će proći svih 60 stepeni rotiranja, i to kroz 21 ugao sa razmakom od 3 stepena. Za svako od pomenutih rotiranja, proveriće se bliskost sa svakim od mogućih znakova i biće izabran onaj znak sa kojim se pronađe najveća korelacija.

Ostaje pitanje kako se vrši obučavanje sa znacima koji nisu rotirani. Najpre, koristi se program za izdvajanje karaktera iz pozadine, kako bi se prikupio što veći broj primera prema kojima bi se klasifikator trenirao. Radi lakše obrade, za svaki znak je kreiran poseban folder, u koji se smeštaju slike u kojima se on nalazi. Program za izdvajanje znakova iz pozadine ipak ne radi tačno u 100% slučajeva, a što je još i bitnije, znakovi dobijeni ekstrakcijom su rotirani, svaki pod određenim uglom. To zahteva ljudski faktor koji bi ručno morao da izvrši proveru koji od znakova su dobro raspoređeni, što je lakši, ali i ručno rotiranje slika da bi znakovi došli u početni položaj. Da bi taj proces bio što brži, izvesna doza automatizacije je potrebna. Za tu svrhu, kreiran je program `TestRotating`, koji omogućuje brz prelaz sa slike na sliku radi rotiranja. Za rotiranje i promenu veličine slike, upotrebljene su funkcije programskog jezika Java, iz klase `AffineTransform`.

Kod treniranja, uspešnost se može videti u tabeli 3.2 U prvoj koloni tabele prikazani su znaci koji se mogu naći u datim CAPTCHA sličicama. Za obučavanje sa udruživanjem (adaboost) primenjena su dva postupka. Prvi i nešto lakši je u slučaju kada se netačni primeri ne rotiraju, a drugi postaje malo teži jer su oni rotirani pod različitim uglovima i tako otežavaju obučavanje. Drugi primer je dosta realniji, jer se prilikom prepoznavanja treba odstraniti i slučajeve kada su netačni znaci okrenuti pod lošim uglom.

U ovom eksperimentu, korišćene su tri vrste klasifikatora, koje se baziraju na istom principu. Najpre, svaki klasifikator odredi broj koji pridruži slici na osnovu određenog algoritma. Taj broj možemo nazvati vrednost slike. Ako je vrednost slike iznad određenog praga, slika je pozitivno klasifikovana, inače je negativno. Prva pomenuta vrsta je na slici 3.3a) i predstavlja klasifikator sa četiri pravougaonika. Pravougaonik koji je deo slike, sastoji se iz četiri manja. Kada se vrši klasifikacija, vrednost slike se računa tako što se sumiraju vrednosti tačaka (od 0 do 255) u svakom delu pravougaonika, pomnože sa koeficijentom koji odgovara tom manjem pravougaoniku i na kraju sve saberu. Jedan klasifikator definisan je veličinom pravougaonika, pozicijom u slici i koeficijentima a_1, a_2, a_3 i a_4 i pragom. Na taj način se može izgenerisati veliki (gotovo beskonačan broj klasifikatora), s tim što su u ovom radu ograničene vrednosti koeficijenata na skup {-1,1}. Na sličan način se definišu i preostale dve vrste klasifikatora, prva se zove tri vertikale a druga tri horizontale. Predstavljene su na istoj slici, pod b) i c). Vrši se sumiranje vrednosti tačaka u svakom manjem pravougaoniku i dodaju se na ukupan zbir pomnožene sa svojim koeficijentom. U sva tri slučaja, otvara se mogućnost za primenu genetskih algoritama prilikom podešavanja paarametara a_1, a_2, a_3 i a_4, naravno pod uslovom da ovi mogu da uzimaju vrednosti izvan skupa {-1,1}.



Slika 3.3 – prikazi jednostavnih klasifikatora
a) četiri pravougaonika, b) tri vertikale, c) tri horizontale

ZNAK	Uspešnost bez rotacije negativnih primera	Uspešnost sa rotacijom negativnih primera
2	0.996	0.992
3	0.996	0.996
4	0.984	0.996
5	0.988	0.984
6	0.996	0.996
7	0.996	0.996
8	0.992	0.996
9	0.996	0.992
A	0.988	0.996
B	0.98	0.976
C	0.996	1.000
D	0.996	0.968
E	0.992	0.984
F	0.996	0.996
G	0.996	0.996
H	0.984	0.992
J	0.996	0.988
K	0.992	0.996
L	0.996	0.996
M	0.984	0.984
N	0.996	0.988
P	0.996	0.996
Q	0.964	0.996
R	0.988	0.996
S	0.996	0.996
T	0.996	0.996
U	0.992	0.996
V	0.996	0.992
W	0.996	0.988

ZNAK	Uspešnost bez rotacije negativnih primera	Uspešnost sa rotacijom negativnih primera
X	0.992	0.996
Y	0.996	0.996
Z	0.996	0.988
@	0.996	0.992
#	0.996	0.996
$	0.996	0.996
%	0.995	0.993
?	0.996	0.996
&	0.996	0.996
*	0.996	0.996

Tabela 3.2 – Uspešnost prilikom prepoznavanja znakova
Napomena: znakovi koji nisu u tabeli, ne koriste se u CAPTCHA slikama B2evo.

Rezulttati obučavanja su jako dobri, jer pokazuju da jednostavni klasifikatori kada se ukombinuju mogu jako dobro da funkcionišu. Gotovo u svim slučajevima, uspešnost treniranja je oko 99%, što je veoma dobro.

4 Implementacija

Kompletan programski kod korišćen u ovom radu, napisan je posebno za ovu priliku. Sve je otkucano u programskom jeziku Java, u programskom okruženju Eclipse. Ukupan softverski paket se sastoji iz nekoliko programa, od kojih su neki povezani međusobno. Naime, kao potpuno zasebni projekti, izdvajaju se dva pomoćna projekta koji se koriste, to su `AdaBoost` i `GeneticAlgorithms`. Glavni projekat se naziva `Diplomski` i oslanja se na prethodna dva. Projekti `AdaBoost` i `GeneticAlgorithms` pisani su tako da se mogu upotrebljavati i kasnije, a takođe i jednostavno dopunjavati novim idejama i metodama.

4.1 Projekat GeneticAlgorithms

Radi razlikovanja od drugih projekata, prilikom povezivanja, ovaj projekat ima glavni paket pod nazivom `genetic`. Sastoji se od nekoliko potpaketa, prema abecednom redu to su: `common`, `genes` (sa svojim potpaketima `bitgene` i `floatgene`), `mutation`, `reproduction`, `selection` i `test`. Kao što se može primetiti, svaki paket je dobio naziv prema jednom od bitnijih pojmova u genetskim algoritmima i u njemu se nalaze klase i interfejsi korišćeni baš za tu operaciju, osim paketa test, koji se koristi da bi se proverio ispravan rad sistema pre nego što se ovaj primeni u drugim projektima.

Krenimo redom, paket `common` sadrži najopštije klase koje se mogu reći da su zajedničke svakom genetskom algoritmu. Prva je apstraktna klasa `Entity` i označava jednu jedinku. Klasa je aptstraktna jer je implementacija jedinke različita u svakoj primeni. U klasi `Entity`, nalaze se sledeće funkcije koje treba implementirati, a to su:

```
public abstract Entity clone();

public abstract double evaluate();

public abstract Entity[] crossOver(Entity e, double factor);

public abstract Entity mutate(double factor);
```

Uz njih, nalazi se i parametar `gene`.

```
protected Gene gene;
```

Funkcija clone je nasleđena iz osnovne klase `Object`, i treba da ima istu ulogu kao i tamo. Funkcija `evaluate` je zapravo funkcija evaluacije, čiju vrednost treba optimizovati. Funkcija `crossOver` se koristi za ukrštanje, i takođe je apstraktna, pre svega zbog različitih načina ukrštanja različitih tipova gena. Takođe, i funkcija `mutate` je iz istog razloga apstraktna. Pošto svaka jedinka ima svoj gen, parametar `gene` je kao polje zaštićenog (*eng. protected*) tipa da bi se mogao koristiti u nasleđenim klasama. Ono što je

logično, to je da se na osnovu gena vrši dekodiranje parametara i na osnovu dekodiranih informacija vraća vrednost funkcije evaluate.

Klasa `Population` se takođe nalazi u paketu common, i odnosi se na sve operacije koje se događaju sa skupom jedinki koji nazivamo populacija. U konstruktoru objekata ovog tipa se nalaze parametri na osnovu kojih funkcioniše ceo genetski algoritam. To su veličina populacije, konstante mutacije i ukrštanja, komparator koji određuje da li se računa maksimum ili minimum funkcije evaluacije, zatim operatori selekcije i reprodukcije. Broj generacija se ne zadaje eksplicitno, već se za prelaz iz jedne generacije u drugu koristi funkcija `nextGeneration`() koja se može zvati koliko je puta potrebno. To je dobro, jer broj poziva nije fiksiran, već se može prekinuti bilo kada je korisnik zadovoljan pronađenim rešenjem bilo kada je proteklo više vremena nego što se želi. U paketu common se nalaze i klase `EntityMinimumComparator` i `EntityMaximumComparator`, koje implementiraju interfejs `Comparator` i koriste se prilikom selekcije.

U paketu gene, zajedno sa njegovim potpaketima nalazi se implementacija same srži genetskih algoritama, gena i to bitovskih i kontinualnih. Osnova je apstraktna klasna `Gene`, iz koje proističu klase `BitGene` i `FloatGene`. Klasa `Gene` sadrži sledeće apstraktne metode:

```
public abstract Gene mutate(double mutateFactor);

public abstract Gene[] crossOver(Gene otherGene, double crossOverFactor);

public abstract Gene clone();

public abstract void randomize();
```

Ponovo, slično klasi `Entity`, i klasa `Gene` poseduje metode za mutaciju, ukrštanje i kloniranje, uz metod randomizacije koji je potreban u startu, kada se počinje od slučajnih gena. Te metode su apstraktne zbog drugačije implementacije realnih gena i bitovskih ili u opštem slučaju bilo koje druge vrte gena. U ovim paketima se nalaze apstraktne klase za operacije ukrštanja, `BitCrossOverOperator` i `FloatCrossOverOperator`, koje dalje nasleđuju `BitUniformCrossOver` i `OnePointCrossOver` kod bitovskih gena, odnosno `OnePointBlendingCrossOver` kod kontinualnih. Jedina funkcija bitovskih operatora ukrštanja je

```
public abstract BitSet[] crossOver(BitSet a, BitSet b, int size);
```

jer se bitovski geni baziraju na objektima tipa `BitSet`. Kod kontinualnih gena, slična funkcija postoji, i njen potpis je

```
public double[][] crossOver(double[] a, double[] b, double[] min,
                            double[] max);
```

što je i logično s obzirom da parametri imaju gornje i donje granice. Često korišćenje apstraktnih metoda u mnogome olakšava programiranje i omogućava brzo širenje projekta. Tako bi se vrlo brzo mogli dodati novi metodi ukrštanja, kao što je na primer ukrštanje sa više tačaka, ili kompletno nove vrste gena kao što su geni permutacija.

Paket mutation sadrži apstraktnu klasu `MutationOperator` koja ima samo jednu

metodu. Njen potpis je:

```
public abstract <T extends Entity> Vector<T> mutate(Vector<T> entities,
                                                    double mutationFactor);
```

Klase koje nasleđuju klasu `MutationOperator` primaju kroz ovu funkciju niz jedinki i na nekima vrše mutacije, u zavisnosti od verovatnoće i generatora slučajnih brojeva. Kao što se može primetiti, u ovoj funkciji koriste se šabloni, paradigma koja u Java programskom jeziku nije baš na najbolji način implementirana (u poređenju sa jezikom C++). Često se u kompletnom projektu koriste šabloni (*eng. template*) jer su jako dobri za smanjenje koda i dobar stil programiranja. U ovom paketu, klase koje mogu da budu operatori mutacije su `ClassicMutation` i `EliteMutation`. `EliteMutation` se samo razlikuje u tome što čuva najkvalitetniju (prema oceni) jedinku u populaciji i ne dozvoljava da se ona mutira.

U paketima reproduction i selection se analogno nalaze operatori za prirodnu selekciju i izbor parova za razmnožavanje. Što se tiče reproduction paketa, tu je jedino `RandomReproduction` klasa, o kojoj ime dovoljno govori. Što se tiče operatora izbora, tu su tri različita, od kojih je svaki naslednik apstraktne klase `SelectionOperator`. To su klase `EliteSelection`, `EliteRouletteSelection` i `RouletteSelection`. Svaka od njih implementira funkciju

```
public abstract <T extends Entity> Vector<T> select(Vector<T> entities,
                                                    int nRemaining);
```

Iz niza jedinki, vraća se novi niz koji sadrži `nRemaining` jedinki koje se nalaze u inicijalnom nizu. Metoda koja se koristi je elitna, rulet ili kombinacija elitističke i rulet. Vrlo je jednostavno dodati novu metodu selekcije.

U paketu test, koji stoji provere radi, nalaze se klase za nalaženje ekstremnih vrednosti običnih jednodimenzionalnih i višedimenzionalnih funkcija.

4.2 Projekat Boosting

Ovaj projekat je predviđen za implementaciju tehnika sa udruživanjem. U kontekstu ovog rad, iskucan je programski kod algoritma Ada udruživanja, koji je u stanju da se uključi u proces obučavanja bilo kog klasifikacionog problema. U paketu `boosting` se nalazi jedini paket adaboost, koji sadrži sve klase i interfejse potrebne da se koristi ovaj algoritam za obučavanje udruživanjem. Klase koje se koriste u ovom algoritmu su `AdaBoost`, `StrongClassifier`, `TestCase`, `WeakClassifier` i `WeakTrainer`.

Sa objašnjenjem ovog projekta krenućemo od klase `TestCase`. To je ništa drugo nego prazna klasa bez ijedne metode. Klase koje nasleđuju ovu klasu se koriste da bi definisale primere za treniranje, što pozitivne, što negativne. Klasi `AdaBoost`, koja je osnovna klasa u kojoj je algoritam implementiran, prosleđuju se dva niza objekata tipa `TestCase`, jedan za pozitivne test primere, a drugi za negativne. Pored ovih nizova, klasa `AdaBoost` prima kroz konstruktor i objekat apstraktne klase `WeakTrainer`. Ovo su polja klase `AdaBoost`:

```
private T[] pos, neg;

private double[] posVals, negVals;

private WeakTrainer<T> wt;

private Vector<WeakClassifier<T>> classifiers;

private Vector<Double> relevance;
```

I ovde se koriste šabloni, pa je tip `T` izveden iz tipa `TestCase`. Nizovi `pos` i `neg` su upravo nizovi pozitivnih i negativnih trening primera. Nizovi `posVals` i `negVals` su realni brojevi koji označavaju koliko je koji primer bitan. Objekat `wt` tipa `WeakTrainer` se koristi za proizvodnju jednostavnih klasifikatora, koji se smeštaju u niz classifiers a važnost svakog od njih čuva se u nizu relevance.

Nakon konstruktora klase `AdaBoost`, metoda koju treba pozivati je `boostRound`(), i označava proizvodnju novog jednostavnog klasifikatora u zavisnosti od test primera i njihovih važnosti iz nizova `posVals` i `negVals`. Novi jednostavni klasifikator i njegova relevantnost se smeštaju u niz classifiers i relevance. Tako, na osnovu nekoliko rundi (barem jedne) moguće je kreirati objekat klase `StrongClassifier` metodom `getStrongClassifier`(). Metodu `boostRound` treba pozivati dok korisnik nije zadovoljan preciznošću klasifikatora ili dok broj rundi ne prevaziđe određeni, maksimalni broj.

Kako bi uopšte obučavanje bilo moguće, i njegovi rezultati bili vidljivi, koriste se dve dodatne klase, `WeakTrainer`, `StrongClassifier` i interfejs `WeakClassifier`. Objekat tipa `WeakTrainer`, kao što je veći i navedeno, prosleđuje se objektu `AdaBoost` u konstruktoru. On se poziva prilikom svake runde pojačavanja i njegova apstraktna metoda

```
public abstract WeakClassifier<T> train(double[] posVals, double[] negVals);
```

se koristi da bi se proizveo najbolji klasifikator za date vrednosti pozitivnih i negativnih primera. U zavisnosti od problema koji se obrađuje i `train` metoda će biti drugačija, zato ona i jeste apstraktna. Kao što se vidi, `train` metoda proizvodi jednostavni klasifikator koji je u stanju da u slučaju nešto boljem od 50% predvidi da li dati objekat pripada nekoj vrsti objekata. Interfejs `WeakClassifier` ima sledeće metode

```
public boolean classify(T t);

public void setThreshold(double thr);

public double evaluate(T t);

public WeakClassifier<T> clone();
```

Prva metoda, `classify` upravo izvršava klasifikaciju, i vraća rezultat logičkog tipa. Druga i treća metoda su usko povezane jedna sa drugom. Za potrebe ovog rada, koristi se jednostavna vrsta klasifikatora koja sadrži jedan prag (*eng. threshold*). Svaki test je okarakterisan jednom realnom vrednošću svi test primeri koji daju veću vrednost od praga su pozitivno klasifikovani, a svi primeri koji se vrednuju ispod praga su negativno klasifikovani. Tako, da bi se izvršilo obučavanje, pozove se metoda `evaluate` na svim

testovima za obučavanje i onda se na osnovu rezultata kako pozitivnih, tako i negativnih primera odredi najbolja vrednost za prag i ona postavi metodom `setThreshold`. Na taj način, WeakTrainer može da obuči klasifikatore. Poslednja metoda označava kloniranje.

4.3 Prepoznavanje CAPTCHA slika

Za potrebe ovog rada, kreiran je poseban projekat u Eclipse okruženju. Naziv projekta je `Diplomski` i on se oslanja na dva prethodno navedena projekta, `Boosting` i `GeneticAlgorithms`. U projektu se nalaze programi za obučavanje i kod razdvajanja znakova i kod prepoznavanja. Pored njih tu je još nekoliko nezavisnih alata koji služe da bi se proces što više automatizovao. Sve u svemu, nalazi se četiri paketa, krenimo redom.

Prvi paket se naziva `geometry` i u njemu su pomoćne klase za manipulaciju slika kao geometrijskih figura. Dve klase se nalaze u paketu geometry, njihova imena su `ResizeImage` i `Rotation`. Klasa `ResizeImage` ima dva bitna statička metoda, to su

```
public BufferedImage resize(BufferedImage im, int wn, int hn)
```

i

```
public BufferedImage scaleBounds(BufferedImage im)
```

Prvi metod se koristi prilikom promena veličine slike. Naime, radi uniformnosti kod treniranja, sve slike su dovedene da imaju istu veličinu. U tom slučaju se poziva upravo metoda resize. U drugom slučaju metoda `scaleBounds` prima kao parametar jedino sliku i vraća novu, manju sliku, ali tako da nova slika nema više bespotrebnih tačaka bele boje. Primer efekta metode `scaleBounds` može se videti na slici 4.1

Slika 4.1 – skaliranje slike

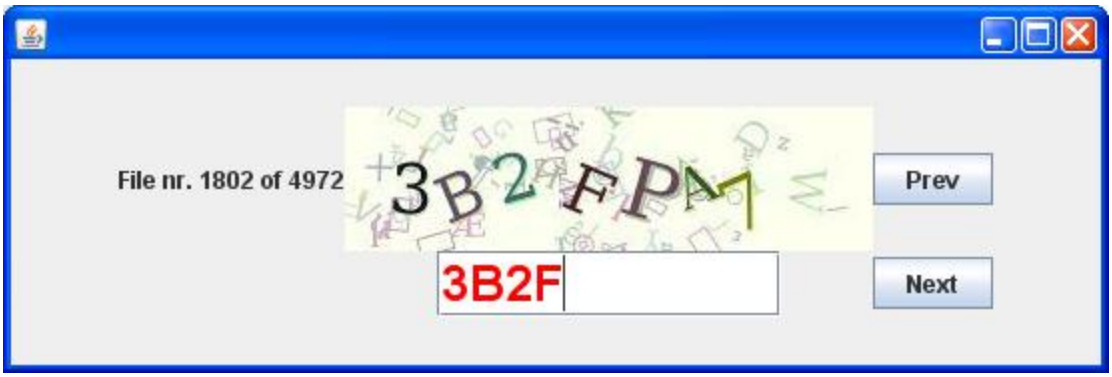
Ova metoda se koristi prilikom izdvajanja znakova iz pozadine, ali i prilikom pripreme testova za prepoznavanje.

Sledeći paket koji se opisuje je `common`. U njemu se nalaze uobičajene stvari, koje se mogu koristiti u bilo kojem programu za prepoznavanje CAPTCHA slika. Tu se nalazi klasa Utils sa svoje četiri metode za prebacivanje iz RGB boje u 32-bitni tip int i obrnuto, za računanje razlike između dve boje i dobijanje 4 bajta od jednog int parametra. Druga klasa je `TestCollecting`, koja je samostalni program i mini alat za dovlačenje velikog broja slika sa stranica na kojima se generišu. Klasa je kreirana tako da implementira interfejs `Runnable`. Sadrži dve bitne metode, to su `downloadWebPage` i `downloadImage`, koje rade upravo onošto im ime govori, dovlače Internet stranice i slike

sa Interneta koje su na tim stranicama, a služe kao CAPTCHA. Da bi se za bilo koji sajt to odradilo, potrebno je navesti kakav je regularan izraz za link koji pokazuje do sličice.

Paket classifiers sadrži tipove jednostavnih klasifikatora korišćenih u ovom radu. To su tri klase, `FourSquareClassifier`, `ThreeHorizontalClassifier` i `ThreeVerticalClassifier`. Svaki vrši klasifikaciju na osnovu vrednosti koju izračunaju i praga koji im je dodeljen.

Paket b2evo sadrži ostale bitne programe vezane za ovu vrstu CAPTCHA sličica. Najpre ćemo pomenuti program TestLabeling, koji je i prvi program iz ovog paketa koji se koristi. Naime, kada se sa Interneta preuzme veliki broj slika, potrebno je svaku obeležiti, tj. u tekstualnom formatu napisati njen sadržaj. Za što efikasnije obeležavanje, koristi se ovaj program. Za dati direktorijum, otvore se sve slike u njemu, i one koje još uvek nisu obeležene se prikazuju jedna po jedna u prozoru aplikacije. Ispod slike se može uneti tekst koji piše na slici. Pritiskom na dugme „next", prelazi se na sledeću sličicu, a uneti podaci se pamte u tekstualnom fajlu koji ima isto ime kao i slika. Na slici 4.2 se vidi kako izgleda aplikacija.



Slika 4.2 – izgled programa TestLabeling za obeležavanje slika

Što se tiče genetskog algoritma razdvajanja, tu je nekoliko klasa. Klasa `GroupSeparation` koja nasleđuje opštu klasu `ImageCharSeparation` i vrši ekstrakciju karaktera iz pozadine na osnovu parametara i algoritma koji su navedeni u odeljku 3.4. Za korišćenje ove klase, potrebno je znati sledeće metode i razumeti kako one funkcionišu.

```
public void store(BufferedImage im);

public void separate();

public int getGroupCount();

public BufferedImage getGroup(int index);
```

Metoda `store` smešta sliku i priprema je za obradu. Nakon nje treba pozvati metodu `separate` koja će odraditi razdvajanje i kada ona završi, mogu se koristiti metode `getGroupCount` da bi se dobio ukupan broj grupa, i metoda `getGroup` koja vraća sliku sa datim indeksom. Za unutrašnje razumevanje date klase, neophodno je poznavati metode:

```
protected void divideIntoGroups()

private void bfs(int x, int y)
```

```
private boolean merging(int g1, int g2)
```

Metoda `divideIntoGroups` najpre vrši filtraciju tačaka, koje treba da budu uključene u znakove, a koje su pozadina. Nakon toga, se više puta poziva `bfs` metoda, sve dok se ne oformira podela u grupe. Posle te podele, proveri se da li je potrebno spajati pojedine grupe, što se radi metodom merging, i ako jeste potrebno, onda se te grupe zaista i spoje i postanu jedna, sve u metodu `divideIntoGroups`. Kompletan izvorni kod ovih metoda može se naći u dodatku na kraju rada.

Kao što je već rečeno, da bi se parametri korišćeni u izdvajanju znakova optimizovali, koristi se genetski algoritam. Da bi se povezao genetski algortitam sa ovim razdvajanjem, uvedene su dodatne klase. To su `GroupSeparationEntity`, `FloatGroupSeparationEntity`, `GroupSeparationTester` i `GeneticGroupSeparationMain`. `FloatGroupSeparationEntity` nasleđuje `GroupSeparationEntity`, dok ova druga nasleđuje klasu `Entity` iz paketa genetskih alogritama. Najbitnija metoda koju ove klase implementiraju je `evaluate`, koja ocenjuje koliko je data jedinka kvalitetna. `GroupSeparationEntity` se oslanja na bitovske gene, doks se izvedena klasa iz nje, `FloatGroupSeparationEntity` oslanja na kontinualne. U svakoj od ovih klasa postoji metoda koja gen koji je pridružen tom entitetu pretvara u korisne informacije, odnosno same parametre. Ta metoda je:

```
protected void transform(Gene g)
```

Kada se poziva funkcija evaluate, najpre se parametri dobijeni iz funkcije transform iskoriste tako da se kreira objekat tipa `GroupSeparation`. Zatim se taj objekat prosledi klasi `GroupSeparationTester` koja svojom funkcijom

```
public double calculateFitness(GroupSeparation sep)
```

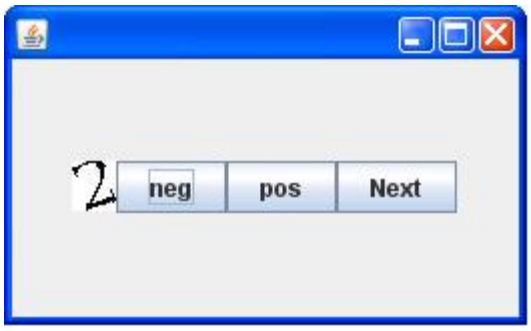
izračunava kvalitet parametara. Funkcija evaluacije je bazirana na izvesnom broju slika koje su u klasi `GroupSeparationTester` otvorene. Ova klasa u konstruktoru prima parametar folderName tipa String i za taj folder se učitaju sve slike koje su u njemu, zajedno sa očekivanim izlazom svake slike.

Klasa `GeneticGroupSeparationMain` je inicijalna kapisla u procesu obučavanja ovog algoritma. Sadrži jedino main funkciju u kojoj se podešavaju razni parametri za genetske algoritme, tip gena, ukrštanja, selekcije, koeficijenti mutacije i ukrštanja... Ova klasa takođe kontroliše ceo postupak obučavanja.

Nakon uspešnog obučavanja algoritma razdvajanja, upravo se taj algoritam koristi kako bi se generisalo što više primera za obučavanje algoritma prepoznavanja. Iz preuzetih slika treba izdvojiti što više znakova koji se kasnije koriste kao primeri prilikom obučavanja. To se postiže programom `SameCharGrouping`, koji koristi već obeležene slike, i kod onih kod kojih se broj grupa razdvojenih programom `GroupSeparation` poklapa sa brojem znakova, te grupe se prebacuju u odgovarajući direktorijum. Kao što je i rečeno, uspeh nije stopostotan, pa je potrebno ručno prebacivanje nekih slika, i odstranjivanje slika koje ne liče dovoljno na znako koji bi trebalo da predstavljaju.

Pošto je napravljena baza sa malim sličicama, od kojih svaka predstavlja po jedan znak,

potrebno ih je i rotirati da bi sve došle u uspravan položaj. Klasa `TestRotating` upravo to radi. Slično aplikaciji `TestLabeling`, i ova aplikacija učitava slike iz pojedinih direktorijuma, prikazuje ih jednu po jednu i ima dugmiće za rotiranje u pozitivnom i negativnom smeru. Slika 4.3 prikazuje kako to izgleda. Pritiskom na dugme next, prelazi se na sledeću sliku u katalogu i pamti se trenutna rotacija.



Slika 4.3 – Izgled programa TestRotating

Kada su slike izrotirane da zauzimaju početni položaj, možemo istrenirati klasifikator putem metode sa pridruživanjem. Tri klase su bitne u ovom postupku, to su `AdaBoostMain`, `B2evoRecognitionTrainer` i `CharacterTest`. Klasa `AdaBoostMain` sadrži main metodu i ona nadgleda obučavanje. Najpre, učitavaju se za svaki znak pozitivni i negativni primeri, koji se predaju objektima klasa `AdaBoost` i `B2EvoRecognitionTrainer`. `B2evoRecognitionTrainer` nasleđuje klasu `WeakTrainer` iz projekta `Boosting` i omogućava treniranje. Koristi klase iz paketa classifiers prilikom treniranja. Klasa `CharacterTest` je izvedena iz klase `TestCase`, takođe iz projekta `Boosting` (koji je opisan u odeljku 4.2). Ona sadrži samu sliku i njene inicijalne dimenzije (pre nego što je izmenjena), koji se koriste u treniranju.

5 Zaključak i moguća unapređenja

Na osnovu kompletnog rada i projekata koji ga sačinjavaju, kada se uzmu u obzir rezultati koji postignuti, može se zaključiti da sistem udruženih jednostavnih klasifikatora zaista dobro funkcioniše i da se može jako dobro primenjivati u problemima prepoznavanja oblika. Uz sve to, treba uzeti u obzir i činjenicu da je primenjeno samo tri vrste jednostavnih klasifikatora i da se to može još dosta poboljšati uvođenjem novih, raznovrsnijih vrsta i trenirati na još većem broju primera, što bi sigurno postiglo odlične rezultate i na mnogo komplikovanijim vrstama objekata.

U ovom radu primenjen je jednostavan vid klasifikatora sa jednim parametrom koji označava prag. Dosta je bolje koristiti još sofisticiranije vrste, koje umesto pragova podele kodomen funkcije evaluacije na određene intervale i pripadnost neke vrednosti određenom intervalu donosi pozitivnu ili negativnu klasifikaciju. Takođe, kada se radi o klasifikatorima, moguće je uvesti još dosta novih, a jedan od bitnijih bi mogao da bude i detekcija uglova i ključnih tačaka (videti [4]).

Radi poboljšanja algoritma otklanjanja šuma, može se koristiti još vrsta genetskih algoritama, eksperimentisati sa novim vrstama selekcije i ukrštanja kako bi se dobijali još bolji rezultati. Da bi se algoritam poboljšao, potrebno je poboljšati i računanje razlike u bojama. Treba primeniti sofisticirano računanje preko potpunih prostora boja kao što je CIELAB (videti [7]).

Paralelizacija je jedna od bitnijih stvari koje može poboljšati procese obučavanja i kod izdvajanja i kod prepoznavanja znakova. Prvi, zato što genetski algoritmi su jako zgodni za paralelno izračunavanje na više računara. Moguće je imati čak i paralelne populacije koje se objedinjuju. U ovom slučaju, više procesora bi još bolje moglo da se iskoristi, pogotovo što evaluacija oduzima 99.9% mašinskog vremena. Više računara bi automatski značilo isto toliko puta više generacija ili primera za testiranje, samim tim i bolje rezultate. Kod obučavanja klasifikatora, takođe bi se paralelizacijom moglo dosta poboljšati, više slabih klasifikatora iskoristiti, i više primera. Obučavanje klasifikatora za svaki znak u ovom trenutku traje nekoliko sati, a u slučaju više računara to bi se isto toliko puta smanjilo, zato što bi vreme potrebno za komunikaciju između računara bilo minimalno u odnosu na vreme izračunavanja.

6 Dodatak A

U ovom odeljku biće predstavljeni zanimljiviji delovi izvornog programskog koda koji je korišćen za realizaciju projekta. Kao što je već rečeno, kompletan kod je pisan u programskom jeziku Java.

6.1 Metode iz paketa GeneticAlgorithms

6.1.1 Uniformno ukrštanje bitova

```
public BitSet[] crossOver(BitSet a, BitSet b, int size) {
          BitSet[] res = new BitSet[2];
          res[0] = new BitSet(size);
          res[1] = new BitSet(size);
          for (int i = 0; i < size; i++)
                    if (r.nextDouble() > 0.5) {
                              res[0].set(i, a.get(i));
                              res[1].set(i, b.get(i));
                    } else {
                              res[0].set(i, b.get(i));
                              res[1].set(i, a.get(i));
                    }
          return res;
}
```

6.1.2 Ukrštanje bitova sa jednom tačkom

```
public BitSet[] crossOver(BitSet a, BitSet b, int size) {
          int n = size;
          int point = r.nextInt(n+1);
          BitSet[] res = new BitSet[2];
          res[0] = new BitSet(n);
          res[1] = new BitSet(n);
          for (int i = 0; i < n; i++)
                    if (i < point) {
                              res[0].set(i, a.get(i));
                              res[1].set(i, b.get(i));
                    } else {
                              res[0].set(i, b.get(i));
                              res[1].set(i, a.get(i));
                    }
          return res;
}
```

6.1.3 Poboljšano ukrštanje kontinualnih gena sa jednom tačkom

```
public double[][] crossOver(double[] a, double[] b, double[] min,
                    double[] max) {
          if (a.length != b.length)
                    return null;
          int n = a.length;
          double[][] res = new double[2][n];
          Random r = new Random();
          int midPoint = r.nextInt(n+1);
          double beta = r.nextDouble();
          for (int i = 0; i < n; i++)
                    if (i < midPoint) {
                              res[0][i] = a[i] * beta + b[i] * (1-beta);
                              res[1][i] = a[i] * (1-beta) + b[i] * beta;
                    } else {
                              res[0][i] = a[i] * (1-beta) + b[i] * beta;
                              res[1][i] = a[i] * beta + b[i] * (1-beta);
                    }
          return res;
}
```

6.1.4 Elitistička metoda izbora

```
public <T extends Entity> Vector<T> select(Vector<T> entities,
                    int nRemaining) {
          Vector<T> res = new Vector<T>();
          for (int i=  0; i < nRemaining; i++)
                    res.add(entities.get(i));
          return res;
}
```

6.1.5 Rulet metoda izbora

```
public <T extends Entity> Vector<T> select(Vector<T> entities,
                    int nSelected) {
          Vector<T> res = new Vector<T>();
          double total = 0;
          double[] sum = new double[entities.size()];
          total = sum[0] = entities.get(0).evaluate();
          for (int i = 1; i < entities.size(); i++) {
                    total+=entities.get(i).evaluate();
                    sum[i] = total;
          }
          Random r = new Random();
          for (int i = 0; i < nSelected; i++) {
                    double randomNumber = r.nextDouble()*total;
                    int index = Arrays.binarySearch(sum, randomNumber);
                    index = -index -1;
                    res.add((T)entities.get(index).clone());
          }
          return res;
}
```

6.1.6 Kombinacija elitističke i rulet metode izbora

```
        public <T extends Entity> Vector<T> select(Vector<T> entities,
                        int nSelected) {
                Vector<T> res = new Vector<T>();
                double total = 0;
                double[] sum = new double[entities.size()];
                total = sum[0] = entities.get(0).evaluate();
                for (int i = 1; i < entities.size(); i++) {
                        total+=entities.get(i).evaluate();
                        sum[i] = total;
                }
                // nalazenje najbolje jedinke
                T best = entities.get(0);
                for (int i = 1; i < entities.size(); i++)
                        if (comp.compare(best, entities.get(i))< 0)
                                best = entities.get(i);
                //izbor ostalih po rulet algoritmu
                Random r = new Random();
                for (int i = 0; i < nSelected; i++) {
                        double randomNumber = r.nextDouble()*total;
                        int index = Arrays.binarySearch(sum, randomNumber);
                        index = -index -1;
                        res.add((T)entities.get(index).clone());
                }
                return res;
        }
```

6.2 Metode iz paketa AdaBoost

6.2.1 Runda udruživanja

```
        public void boostRound() {
                WeakClassifier<T> wc = wt.train(posVals, negVals);
                double error = 0;
                boolean[] corPos = new boolean[pos.length], corNeg = new
boolean[neg.length];
                // calculating error
                for (int i = 0; i < pos.length; i++) {
                        corPos[i] = wc.classify(pos[i]);
                        if (!corPos[i])
                                error += posVals[i];
                }
                for (int i = 0; i < neg.length; i++) {
                        corNeg[i] = !wc.classify(neg[i]);
                        if (!corNeg[i])
                                error +=  negVals[i];
                }
                double rel  = Math.log((1-error)/error);
                double exp = Math.exp(rel);

                // update strong classifier
                classifiers.add(wc);
                relevance.add(rel);

                // updating weights
                // and normalizing the distribution
```

```
                double sum = 0;
                for (int i = 0; i < pos.length; i++   ) {
                        if (corPos[i])
                                posVals[i] /= exp;
                        else
                                posVals[i] *= exp;
                        sum += posVals[i];
                }
                for (int i = 0; i < neg.length; i++) {
                        if (corNeg[i])
                                negVals[i] /= exp;
                        else
                                negVals[i] *= exp;
                        sum += negVals[i];
                }
                for (int i = 0; i< posVals.length; i++)
                        posVals[i]/=sum;
                for (int i = 0; i< negVals.length; i++)
                        negVals[i]/=sum;
        }
```

6.2.2 Klasifikacija kod jakog klasifikatora

```
        public boolean classify(T t) {
                double sum = 0;
                for (int i = 0; i < cls.length; i++) {
                        if (cls[i].classify(t))
                                        sum += relevance[i];
                        else
                                sum -= relevance[i];
                }
                return sum > 0;
        }
```

6.3 Metode korišćene kod prepoznavanja oblika

6.3.1 Smanjivanje slike, zadržavanje samo bitnih podataka

```
        /**
         * crops only the important part of the image
         * @param im
         * @return
         */
        public BufferedImage scaleBounds(BufferedImage im) {
                BufferedImage res;
                int xmin = im.getWidth() + 1, xmax = -1, ymin =
im.getHeight() + 1, ymax = -1;
                Color[][] c = new Color[im.getWidth()][im.getHeight()];
                for (int i = 0; i < im.getWidth(); i++)
                        for (int j = 0; j < im.getHeight(); j++) {
                                c[i][j] = Utils.int2Color(im.getRGB(i,
j));
                                if (c[i][j].getRed() < 250 &&
c[i][j].getBlue() < 250
                                                && c[i][j].getGreen() <
```

```
250) {
                                                    if (i < xmin)
                                                              xmin = i;
                                                    if (i > xmax)
                                                              xmax = i;
                                                    if (j < ymin)
                                                              ymin = j;
                                                    if (j > ymax)
                                                              ymax = j;
                                          }
                                }
                    res = new BufferedImage(xmax - xmin + 1, ymax - ymin + 1,
                                          BufferedImage.TYPE_INT_RGB);
                    for (int i = 0; i < xmax - xmin + 1; i++)
                                for(int j = 0; j < ymax - ymin + 1; j++)
                                          res.setRGB(i, j,
Utils.color2Int(c[i+xmin][j+ymin]));
                    return res;
          }
```

6.3.2 Rotacija slike

```
private BufferedImage invert(BufferedImage im) {
          BufferedImage res = new BufferedImage(im.getWidth(), im.getHeight(),
                                          BufferedImage.TYPE_INT_RGB);
          for (int i = 0; i < res.getWidth(); i++)
                    for (int j = 0; j < res.getHeight(); j++)
                              res.setRGB(i, j, im.getRGB(i, j) ^ 2147483647);
          return res;
}

public BufferedImage rotate(BufferedImage im, double angle) {
          BufferedImage pom = new BufferedImage(im.getHeight() + im.getWidth(),
                                          im.getWidth() + im.getHeight(),
                                          BufferedImage.TYPE_INT_RGB);

          for (int i = 0; i < pom.getWidth(); i++)
                    for (int j = 0; j < pom.getHeight(); j++)
                              pom.setRGB(i, j, Utils.color2Int(new Color(255,
                                                                255, 255)));

          for (int i = 0; i < im.getWidth(); i++)
                    for (int j = 0; j < im.getHeight(); j++)
                              pom.setRGB(im.getHeight() / 2 + i, im.getWidth() /
                                                    2 + j, im.getRGB(i, j));

          pom = invert(pom);
          AffineTransformOp op = new AffineTransformOp(AffineTransform
                                          .getRotateInstance(angle, pom.getWidth()
                                          2, pom.getHeight() / 2), null);

          BufferedImage res = op.filter(pom, null);
          return new ResizeImage().scaleBounds(invert(res));
}
```

6.3.3 Evaluacija kod klasifikatora sa četiri pravougaonika

```
public double evaluate(T t) {
          double[][] mat = t.getIntensityMatrix();
          double val = 0;
          for (int i = 0; i < w; i++)
                    for (int j = 0; j < h; j++)
                              if (i < w / 2)
                                        if (j < h / 2)
                                                  val += mat[x0 + i][y0 + j] * a1;
                                        else
                                                  val += mat[x0 + i][y0 + j] * a2;
                              else if (j < h / 2)
                                        val += mat[x0 + i][y0 + j] * a3;
                              else
                                        val += mat[x0 + i][y0 + j] * a4;
          return val;
}
```

6.3.4 Evaluacija kod klasifikatora sa tri uspravna pravougaonika

```
public double evaluate(T t) {
          double[][] mat = t.getIntensityMatrix();
          double val = 0;
          for (int i = 0; i < w; i++)
                    for (int j= 0; j < h; j++)
                              if (i < w/3)
                                        val += a1 * mat[x+i][y+j];
                              else
                                        if (i < (2*w)/3)
                                                  val += a2 * mat[x+i][y+j];
                                        else
                                                  val += a3 * mat[x+i][y+j];
          return val;
}
```

6.3.5 Evaluacija kod klasifikatora sa tri horizontalna pravougaonika

```
public double evaluate(T t) {
          double[][] mat = t.getIntensityMatrix();
          double val = 0;
          for (int i = 0; i < w; i++)
                    for (int j = 0; j < h; j++)
                              if (j < h / 3)
                                        val += a1 * mat[x + i][y + j];
                              else if (j < (2 * h) / 3)
                                        val += a2 * mat[x + i][y + j];
                              else
                                        val += a3 * mat[x + i][y + j];
          return val;
}
```

6.3.6 Pretraživanje u širinu kod razdvajanja znakova

```
private void bfs(int x, int y) {
          int[] qx = new int[w * h], qy = new int[w * h];
          int[] nx = { 0, 1, 0, -1, 1, 1, -1, -1 }, ny = { 1, 0, -1, 0, -1, 1,
                                        -1, 1 };
          int b = 0, e = 0;
          qx[b] = x;
          qy[b] = y;
          int maxX = -1, minX = w;
          int maxY = -1, minY = h;
          int sumX = x, sumY = y;
          double sumR = redIm[x][y], sumG = greenIm[x][y], sumB = blueIm[x][y];

          float[] hsb = Color.RGBtoHSB((int) sumR, (int) sumG, (int) sumB,null);
          double sumH = hsb[0];
          double sumS = hsb[1];
          int gs = 1;
          group[x][y] = groupCount;
          while (b <= e) {
                    int xx = qx[b], yy = qy[b];
                    if (xx > maxX)
                              maxX = xx;
                    if (xx < minX)
                              minX = xx;
                    if (yy > maxY)
                              maxY = yy;
                    if (yy < minY)
                              minY = yy;

                    for (int i = 0; i < nx.length; i++) {
                              int nextX = xx + nx[i];
                              int nextY = yy + ny[i];
                              if (inside(nextX, nextY, w, h))
                              if (group[nextX][nextY] == UNVISITED) {
                                        double refRed, refGreen, refBlue;
                                        refRed = sumR / gs;
                                        refGreen = sumG / gs;
                                        refBlue = sumB / gs;
                                        float[] refHsb =
                                        Color.RGBtoHSB((int) refRed,
                                        (int) refGreen, (int) refBlue, null);

                                        float[] hsbNext = Color.RGBtoHSB(
                                        (int) redIm[nextX][nextY],
                                        (int) greenIm[nextX][nextY],
                                        (int) blueIm[nextX][nextY],
                                        null);

                                        double hdist =
                                        Math.abs(hsbNext[0] - refHsb[0]);
                                        if (1 - hdist < hdist)
                                                  hdist = 1 - hdist;

                                        if (hsbNext[1] < 0.2 && refHsb[1] < 0.2)
                                                            hdist /= 4;
                                                  if (hsbNext[2] < 0.2 && refHsb[2]
                                                            < 0.2)
                                                            hdist /= 4;

                                                  if (hdist < colorDifThreshold) {
                                                            e++;
```

```
                group[nextX][nextY] = groupCount;
                qx[e] = nextX;
                qy[e] = nextY;
                sumX += nextX;
                sumY += nextY;
                hsb = Color.RGBtoHSB((int) redIm[xx][yy],(int) greenIm[xx][yy],
                                (int) blueIm[xx][yy], null);
                sumH += hsb[0];
                sumS += hsb[1];
                sumR += redIm[xx][yy];
                sumG += greenIm[xx][yy];
                sumB += blueIm[xx][yy];
                gs++;
                                }
                        }
                }
                b++;
                }
                if (gs >= groupSizeThreshold) {
                        groupCount++;
                        this.maxX.add(maxX);
                        this.minX.add(minX);
                        this.maxY.add(maxY);
                        this.minY.add(minY);
                        this.sumX.add(sumX);
                        this.sumY.add(sumY);
                        this.sumHue.add(sumH);
                        this.sumSat.add(sumS);
                        this.sumRed.add(sumR);
                        this.sumGreen.add(sumG);
                        this.sumBlue.add(sumB);
                        this.groupSize.add(gs);
                } else {
                        for (int i = 0; i < b; i++)
                                group[qx[i]][qy[i]] = UNVISITED;
                }
}
```

6.3.7 Metoda podele grupa

```
protected void divideIntoGroups() {
          groupCount = 0;
          maxX = new Vector<Integer>();
          minX = new Vector<Integer>();
          maxY = new Vector<Integer>();
          minY = new Vector<Integer>();
          sumX = new Vector<Integer>();
          sumY = new Vector<Integer>();
          sumHue = new Vector<Double>();
          sumSat = new Vector<Double>();
          sumRed = new Vector<Double>();
          sumGreen = new Vector<Double>();
          sumBlue = new Vector<Double>();
          groupSize = new Vector<Integer>();
          // first, we scan every pixel and see if it is enoguh colorful,
          // otherwise it is background (0)
          for (int i = 0; i < w; i++)
                    for (int j = 0; j < h; j++) {
                              if (intensity[i][j] > colorIntensityThreshold)
                                        group[i][j] = UNVISITED;
                              else
                                        group[i][j] = BACKGROUND;
                    }
          // using bfs algorithm, all adjacent pixels with similar colors are
          // grouped
          long bfsStarted = System.nanoTime();
          for (int i = 0; i < w; i++)
                    for (int j = 0; j < h; j++)
                              if (group[i][j] == UNVISITED) {
                                        bfs(i, j);
                              }
          long bfsEnded = System.nanoTime();
          // closest groups with similar colors are being merged
          // their indices are sorted based on minX (1-based)
          int[] ind = new int[groupCount];
          for (int i = 0; i < groupCount; i++)
                    ind[i] = i;
          for (int i = 0; i < groupCount; i++)
                    for (int j = i + 1; j < groupCount; j++)
                              if (minX.get(ind[j]) < minX.get(ind[i])) {
                                        int pom = ind[i];
                                        ind[i] = ind[j];
                                        ind[j] = pom;
                              }
          // try to join every two close groups
          // array switched shows if some group has merged into another
          // zero means that the group has
          int[] switched = new int[groupCount];
          for (int i = 0; i < groupCount; i++)
                    switched[i] = UNVISITED;
          int mergeCount = 0;
          for (int i = 1; i < groupCount; i++) {
                    int g1 = ind[i - 1], g2 = ind[i];
                    if (merging(g1, g2)) {
                              switched[g1] = g2;
                              mergeCount++;
                              minX.set(g2, Math.min(minX.get(g1), minX.get(g2)));
                              maxX.set(g2, Math.max(maxX.get(g1), maxX.get(g2)));
                              minY.set(g2, Math.min(minY.get(g1), minY.get(g2)));
                              maxY.set(g2, Math.max(maxY.get(g1), maxY.get(g2)));
                              sumX.set(g2, sumX.get(g1) + sumX.get(g2));
```

```
                        sumY.set(g2, sumY.get(g1) + sumY.get(g2));
                        sumHue.set(g2, sumHue.get(g1) + sumHue.get(g2));
                        sumSat.set(g2, sumSat.get(g1) + sumSat.get(g2));
                        sumRed.set(g2, sumRed.get(g1) + sumRed.get(g2));
                        sumGreen.set(g2, sumGreen.get(g1)
                        +sumGreen.get(g2));
                        sumBlue.set(g2, sumBlue.get(g1) + sumBlue.get(g2));
                        groupSize.set(g2, groupSize.get(g1) +
                        groupSize.get(g2));
                }
        }
        for (int i = 0; i < groupCount; i++) {
                int x = i;
                while (switched[x] != UNVISITED) {
                        x = switched[x];
                }
                if (x != i)
                        switched[i] = x;
        }
        Vector<Integer> gr = new Vector<Integer>();
        int[] newGroup = new int[groupCount];
        for (int i = 0; i < groupCount; i++)
                if (switched[i] == UNVISITED) {
                        gr.add(i);
                        newGroup[i] = gr.size() - 1;
                }
        for (int i = 0; i < w; i++)
                for (int j = 0; j < h; j++)
                        if (group[i][j] >= 0) {
                                int grIndex = group[i][j];
                                if (switched[grIndex] == UNVISITED)
                                        group[i][j] = newGroup[grIndex];
                                else
                                        group[i][j] =
                                        newGroup[switched[grIndex]];
                        }
        groupCount = gr.size();
}
```

6.3.8 Metoda spajanja dve grupe

```
private boolean merging(int g1, int g2) {
          double euDist, horDist, colorDist;
          double x1sr = 1. * sumX.get(g1) / groupSize.get(g1);
          double x2sr = 1. * sumX.get(g2) / groupSize.get(g2);
          double y1sr = 1. * sumY.get(g1) / groupSize.get(g1);
          double y2sr = 1. * sumY.get(g2) / groupSize.get(g2);
          euDist = Math.sqrt((x1sr - x2sr) * (x1sr - x2sr) + (y1sr - y2sr)
                                        * (y1sr - y2sr));

          float[] hsb1 = Color.RGBtoHSB(
                              (int) (sumRed.get(g1) / groupSize.get(g1)),
                              (int) (sumGreen.get(g1) / groupSize.get(g1)),
                              (int) (sumBlue.get(g1) / groupSize.get(g1)), null);

          float[] hsb2 = Color.RGBtoHSB(
                              (int) (sumRed.get(g2) / groupSize.get(g2)),
                              (int) (sumGreen.get(g2) / groupSize.get(g2)),
                              (int) (sumBlue.get(g2) / groupSize.get(g2)), null);

          colorDist = Math.abs(hsb1[0] - hsb2[0]);
          if (1 - colorDist < colorDist)
                    colorDist = 1 - colorDist;

          double sat1 = hsb1[1], sat2 = hsb2[1];
          if (sat1 < 0.2 && sat2 < 0.2)
                    colorDist /= 4;
          double l1 = hsb1[2], l2 = hsb2[2];
          if (l1 < 0.2 && l2 < 0.2)
                    colorDist /= 4;

          double overlapping = 0;
          if (maxX.get(g1) > minX.get(g2)) {
                    horDist = 0;
                    overlapping = maxX.get(g1) - minX.get(g2);
                    overlapping = overlapping / (maxX.get(g2) - minX.get(g2));
          } else {
                    horDist = minX.get(g2) - maxX.get(g1);
          }
          if (overlapping > 0.7)
                    euDist = 0;

          if (horDist > horDistThreshold)
                    return false;

          if (euDist > euclidDistCoef)
                    return false;

          if (colorDist > colorDiffCoef)
                    return false;
          return true;
}
```

6.3.9 Evaluacija jedinke

```
public double calculateFitness(GroupSeparation sep) {
          double error = 0;
          double ave = 0;
          for (int i = 0; i < groupCount.length; i++) {
                    sep.storeMatrices(redIm[i].length, redIm[i][0].length,
                                        redIm[i], greenIm[i], blueIm[i],
                                        intensity[i]);
                    sep.separate();
                    int n = sep.getGroupCount();
                    error += Math.abs((n - groupCount[i]));
                    double sumCur = 0;
                    for (int x : sep.getGroupDimensions())
                              sumCur += x;
                    if (sumCur > 0)
                              ave += sumCur / sep.getGroupCount();
          }
          error /= groupCount.length;
          ave /= groupCount.length;
          double aveErr = Math.abs(150 - ave) / 150;
          return (1 / (1 + error + aveErr));
}
```

6.3.10 Obučavanje metode prepoznavanja

```
public WeakClassifier<CharacterTest> train(double[] posVals,
                    double[] negVals) {
          double skrozBest = 0;
          WeakClassifier<CharacterTest> res = null;
          WeakClassifier<CharacterTest> cl;
          reset();
          int number = 0;
          while ((cl = getNext()) != null) {
                    number++;
                    double[] posRes, negRes;
                    posRes = new double[posVals.length];
                    negRes = new double[negVals.length];
                    Vector<DoublePair> sorting = new Vector<DoublePair>();
                    for (int j = 0; j < this.pos.length; j++) {
                              posRes[j] = cl.evaluate((T) pos[j]);
                              sorting.add(new DoublePair(posRes[j], posVals[j]));
                    }
                    for (int j = 0; j < this.neg.length; j++) {
                              negRes[j] = cl.evaluate((T) neg[j]);
                              sorting.add(new DoublePair(negRes[j], -negVals
[j]));
                    }
                    double best = 0, sum = 0, rest;
                    double totalSum = 0;
                    int index = 0;
                    Collections.sort(sorting);
                    for (DoublePair d : sorting)
                              totalSum += d.val;
                    for (int j = 0; j < sorting.size() - 1; j++) {
                              sum += sorting.get(j).val;
                              rest = totalSum - sum;
                              if (sum - rest < best) {
                                        best = sum - rest;
                                        index = j;
                              }
```

```
                }
                cl.setThreshold((sorting.get(index).sortable + sorting
                                .get(index + 1).sortable) / 2);
                if (best < skrozBest) {
                        skrozBest = best;
                        res = cl;
                }
        }
        if (res == null)
                System.err.println("Error: No best classifier");
        return res;
}
```

7 Literatura

[1] David E. Goldberg, *Genetic Algorithms in Search, Optimisation and Machine Learning,* Addison-Wesley 2003

[2] Randy L. Haupt, Sue Ellen Haupt, *Practical Genetic Algorithms*, Second Edition, John Wiley & Sons, Inc. Hoboken, New Jersey, 2004

[3] Jiri Matas and Jan Šochman, *AdaBoost*, Centre for Machine Perceptron, Czech Technical University, Prague
URL: http://www.robots.ox.ac.uk/~az/lectures/cv/adaboost_matas.pdf

[4] Reinhard Klette and Azriel Rosenfeld, *Geometric Methods for Digital Picture Analysis ,* Morgan Kaufman, San Francisco 2004

[5] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, *Introduction to Algorithms*, The MIT Press, Cambridbe, Massachusetts, 1990

[6] www.Spam-Filter-Review.com

[7] http://en.wikipedia.org/wiki/Lab_color_space

[8] John H. Holland, *Adaptation in Natural and Artificial Systems*, University of Mitchigan Press, 1975