SVEUČILIŠTE U ZAGREBU
FAKULTET ELEKTROTEHNIKE I RAČUNARSTVA

DIPLOMSKI RAD br. 1479

SUSTAV ZA DETEKCIJU PLAGIJATA TE ODREĐIVANJE AUTORSTVA IZVORNIH KODOVA

Dino Rakipović

Zagreb, lipanj 2017.

Velika zahvala mentoru Anti Đereku na svim savjetima i pomoći tijekom izrade ovog rada. Također, velika hvala Goranu Golubu i Dorianu Šulcu za puno korisnih razgovora te implementacijskih savjeta.

Sadržaj

Uvod

Plagijat (eng. *Plagiarism*) ili postupak krađe tuđeg rada postaje sve veći problem u današnjem svijetu te ga pronalazimo akademskom(npr. eseji, znanstveni članci) i neakademskom(npr. knjige, pjesme) svijetu. Razlog tomu je što količina podataka na internetu, koji je postao dostupan velikom broju ljudi, raste velikom brzinom te je vrlo lagano ukrasti tuđi rad i predstaviti ga kao vlastiti. Ručna potraga za plagijatima na desecima tisuća dokumenata bi trajala vječno te se razvijaju sustavi za detekciju plagijata. Takvi sustavi ne znaju direktno odrediti promatraju li plagijat već najčešće korisniku ispisuju sličnost među parovima dokumenata kako bi korisnik što lakše i brže odlučio. U ovom radu naglasak će biti na detekciji plagijata izvornih kodova. Cilj je izraditi sustav koji bi ubrzao i uvelike pomogao u detekciji plagijata među izvornim kodovima ljudima koji se brinu o programerskim natjecanjima, laboratorijskim vježbama itd.

Detekciji pristupamo iz dva kuta, jedan je određivanje autora izvornog koda tzv. deanonimizacija autorstva, a drugi određivanje sličnosti među parovima izvornih kodova. Određivanje sličnosti među parovima izvornih kodova je problem opisan i implementiran u završnom radu autora [1], no ovdje je implementirano rješenje koje je i do pet puta brže (skup od 1000 izvornih kodova se izvrši u 23 sekunde) od svoje prve inačice te koje je robusnije na više pokušaja plagiranja izvornog koda kao što je zamjena for u *while* petlje. Za utvrđivanje autorstva izvornih kodova bitno je primjetiti da svaki autor dok piše izvorni kod ostavlja svoj jedinstveni otisak, ukoliko se ne radi o projektu na kojem postoje točno utvrđena pravila kojih se svi pridržavaju. Ovdje se ipak kreće od pretpostavke da svaki autor ima svoj jedinstveni stil te su korištene tehnike strojnog učenja kako bi ih naučili razlikovati. U konkretnom slučaju korišten je klasifikator slučajne šume. Izvorni kod se pretvara u brojčani vektor značajki. Neke od značajki su frekvencija korištenja ključnih riječi programskog jezika ili koristi li autor više tabove ili razmake na počecima linija. Klasifikator je naučen na dva skupa izvornih kodova preuzetih sa stranica HONI-a(Hrvatsko otvorene natjecanja u informatici). Na manjem skupu

od 29 autora postignuta je točnost od 94% dok je na većem skupu sa 216 autora točnost iznosila 76%.

Dva pristupa su na kraju integrirana pod istim web sučeljem nazvanim *Turtle*. *Turtle* je prva, prema autorovom najboljem znanju, web aplikacija koja objedinjuje ova dva pristupa. Ovo sučelje nudi predaju datoteka s izvornim kodovima te uvid u sve parove izvornih kodova za koje sumnja da su plagijati te ispisuje njihove sličnosti. Također kako bi detekcija bila još lakša i brža nudi uvid u izvorne kodove svakog para te boja slične dijelove jednakim bojama. Kako bi koristili utvrđivanje autorstva izvornih kodova najprije moramo sustavu predati datoteku s izvornim kodova na kojima se klasifikator trenira. Nakon što sustav istrenira klasifikator, on se sprema te ga se može kasnije koristiti. Korištenjem istreniranog klasifikatora i predajom izvornih kodova za koje želimo utvrditi autorstvo dobijemo za svaki od predanih izvornih kodova tri autora za koje klasifikator odredi da imaju najveću vjerojatnost biti autori tog izvornog koda. Utvrđivanje autorstva je veliki korak naprijed nad određivanjem sličnosti jer nam omogućuje detekciju plagijata među raznim akademskim godinama ukoliko se laboratorijski zadaci mijenjaju. Za kraj treba naglasiti da sustav trenutno podržava samo jezik $C++$, ali je u implementacija ostavljena mogućnost lakog dodavanja novih jezika.

1. Utvrđivanje autorstva

U svijetu u kojem ne postoje unaprijed određena pravila pisanja programskog koda možemo pretpostaviti da svaki programer ostavlja svoj jedinstveni otisak dok programira. Cilj nam je kreirati klasifikator koji bi nam mogao odvojiti autore prema njihovom stilu programiranja. Ovakav klasifikator bi bilo moguće primjeniti na raznim open source projektima na kojima autori razvijaju kod anonimno te bi takav klasifikator mogao narušiti privatnost programera, ali ipak u ovom radu veći naglasak dan je na detekciju plagijata izvornih kodova te ovakav klasifikator koristimo za utvrđivanje autorstva na laboratorijskim vježbama ili na programerskim natjecanjima gdje su nam poznati neki identifikatori autora.

Za rješavanje ovog problema korišteno je strojno učenje. Strojno učenje je grana umjetne inteligencije koja se bavi algoritmima koji mogu učiti na i raditi predviđanja nad skupovima podataka [2]. Postoje dvije vrste učenja, učenje pod nadzorom i učenje bez nadzora. Za učenje pod nadzorom potrebni su labelirani podaci te ih učimo kategorizirati, dok učenje bez nadzora pokušava uočiti uzorke iz podataka bez korištenja labela. Korišteno je nadzirano učenje gdje su podaci izvorni kodovi, a labele autori koji su ih napisali. Konkretno, korišten je klasifikator slučajne šume. Konfiguracija i definicija klasifikatora je detaljnije opisana u 1.3. Klasifikacija je postupak u kojem određujemo kojoj kategoriji (od unaprijed određenih) novi podaci pripadaju. Algoritmi strojnog učenja ulazne podatke primaju u obliku brojeva pa je potrebno izvorni kod pretvoriti u vektor u kojem će svaki broj biti neka od značajki. Što bolje značajke odaberemo algoritam će bolje moći odvajati kategorije tj. autore. Skup odabranih značajki reflektira stil kojim programer piše kod te je nastao po uzoru na [3]. Skup značajki je podijeljen na leksičke, sintaksne i strukturalne. Leksičke i strukturalne značajke dobijemo parsiranjem izvornog koda, a sintaksne iz apstraktnog sintaksnog stabla te je postupak izvlačenja detaljnije opisan 1.1. Na kraju poglavlja 1.5 predstavljeni su i rezultati nad dva skupa izvornih kodova.

1.1 Izvlačenje značajki

Kao što je već spomenuto, kako bi algoritmi strojnog učenja radili potrebni su im brojčani podaci kao ulazi. Izvorni kod se u brojčani vektor značajki pretvara koristeći ideju prvi put opisanu u radu [3], a ideja je da se izvorni kod pretvori u vektor značajki sastavljen od tri dijela, leksičkog, sintaksnog i strukturnog. Leksičke i strukturalne značajke se dobiju izravno parsiranjem izvornog koda, dok nam je za sintaksne značajke potrebno apstraktno sintaksno stablo izvornog koda. Ovako definiran skup značajki je drugačiji za svaki pojedini programski jezik zbog različitosti među njima (npr. drugačije ključne riječi) te je potrebno napisati poseban parser za svaki od njih. U ovom radu naglasak je na programskom jeziku C++ te je izvlačenje značajki implementirano samo za njega.

U nastavku su detaljno opisana i objašnjena sva tri tipa značajki. Većina tih značajki preuzeta je iz [3] dok su neke ideja samog autora. U većini značajki korištena je matematička operacija prirodnog logaritmiranja zbog svojstva da kako idemo prema većim vrijednostima ona sve manje i manje raste te dobro opisuje relativne razlike među značajkama. Neke od značajki su podijeljenje s duljinom izvornog koda kako bi bolje opisale frekvenciju pojavljivanja te značajke.

1.1.1 Leksičke značajke

Leksičke značajke opisuju preferira li autor izvornog koda neke ključne riječi više od drugih(npr. for više od while), koristi li više funkcije ili piše monolitan kod, razne statistike(npr. prosječan broj parametara unutar funkcija), itd. Također izvorni kod se tokenizira te se računa frekvencija tako dobivenih tokena. *Tablica* 1.1 detaljno opisuje svaku od korištenih značajki.

1.1.2 Strukturne značajke

Strukturne značajke opisuju kakvu strukturu autor koristi dok piše izvorni kod, npr. koristi li tabove ili razmake na početku linije, piše li novu liniju prije nego otvori kontrolni blok, itd. *Tablica* 1.2 detaljno opisuje svaku od korištenih značajki.

Tablica 1.1: Definicija leksičkih značajki

Ime značajke	*Definicija*	*Izraz*	*Veličina*
Frekvencija unigrama	Unigram definiramo kao jednu riječ izvornog koda	UnigramFreq	dinamično, oko 20000 za 2160 izvornih kodova (216 autora)
Broj naredbi grananja i petlji	Zbroj svih pojavljivanja naredbi grananja i petlji(for, while, do, if, else if, else, switch)	ln(zbroj/duljina)	7
Broj ključnih riječi	Zbroj svih pojavljivanja ključnih riječi programskog jezika, konkretno C++.	ln(zbroj_kljucne/duljina)	1
Ternarni operatori	Broj pojavljivanja ternarnog operatora	ln(broj_ternarnih/duljina)	1
Komentari	Broj pojavljivanja komentara	ln(broj_kom/duljina)	1
Konstante	Broj pojavljivanja znakovnih i brojčanih konstanti	ln(broj_konst/duljina)	1
Makro naredbe	Broj pojavljivanja makro naredbi	ln(broj_makro/duljina)	1
Funkcije	Broj funkcija	ln(broj_fun/duljina)	1
Tokeni	Token je ekvivalentan unigramu, zbroj svih tokena	ln(broj_token/duljina)	1
Mjere duljine linija	Standardna devijacija i prosjek duljine linija	stddev(linije), avg(linije)	2
Mjere parametara funkcija	Standardna devijacija i prosjek broja parametera funkcija	stddev(param), avg(param)	2
Operatori	Zbroj pojavljivanja svih operatora programskog jezika	ln(zbroj_op/duljina)	1

Tablica 1.2: Definicija strukturnih značajki

Ime značajke	*Definicija*	*Izraz*	*Veličina*
Tabovi	Broj svih tabova	ln(broj_tabova/duljina)	1
Razmaci	Broj svih razmaka	ln(broj_razmaka/duljina)	1
Omjer razmaka	Omjer razmaka(tabovi se broje) i ostalih znakova	ln(zbroj_kljucne/duljina)	1
Nova linija prije vitičastih zagrada	Koristi li autor novi red kada otvara vitičastu zagradu	boolean	1
Tabovi ili razmaci na početku linije	Koristi li autor tabove ili razmake na početku linije	boolean	1

1.1.3 Sintaksne značajke

Sintaksne značajke se dobiju kako je već spomenuto iz apstraktnog sintaksnog stabla izvornog koda. One su što se vremena tiče najskuplje jer kreacija apstraktnih sintaksnih stabala nije brza, no trebale bi dati odlične značajke koje bi uvelike pomogle u deanonimizaciji. Apstraktna sintaksna stabla(ASS) su kreirana koristeći alat *joern* [4]. Ovaj alat nudi posebnu skriptu *joern-parse* koja parsira i vraća čvorove

i bridove apstraktnog sintaksnog stabla. Postoji 58 različitih tipova čvorova(detalji u dodatku A) koje definira *joern.Tablica* 1.3 detaljno opisuje svaku od korištenih značajki.

Tablica 1.3: Definicija sintaksnih značajki

Ime značajke	*Definicija*	*Izraz*	*Veličina*
Bigrami čvorova	Bigrami čvorova su dva čvora koja su povezana u ASS-u	bigrami_čvorovaTF	dinamičko, oko 150000 za 2160 izvornih kodova
Tip čvora ASS-a	Frekvencija pojavljivanja tipa čvora ASS-a	tip_čvoraTF	58
Vrijednost lista ASS-a	Frekvencija vrijednosti listova ASS-a	list_vTF	dinamičko, oko 10000 za 2160 izvornih kodova

1.2 Selekcija značajki

Ovako kreirane značajke rezultiraju u ogromnim, rijetkim (eng. *sparse*) vektorima, čija veličina nekada doseže i stotine tisuća brojeva. Razlog tomu leži u definiciji značajki poput frekvencije tokena, frekvencije bigrama, itd. Rijetkost očitujemo u velikom broju nula unutar vektora. Rijetkost, također, može uzrokovati loš izabir idućeg čvora u klasifikatoru slučajne šume te s time i lošije rezultate. S velikim vektorima također dolazi i do puno sporijeg učenja klasifikatora jer će sva stabla odluka unutar slučajne šume imati više čvorova. Zbog svih navedenih razloga prije samog učenja klasifikatora napravljena je selekcija značajki koja odabire manji broj značajki koje sadrže dovoljno informacija da bi se klasifikator bolje i brže naučio. Tehnika selekcije značajki je mnogo pa se detaljnije ulazi samo u one koje su korištene u ovom radu, a to su selekcija značajki po sadržaju informacije te selekcija značajki po iznosu varijance. Više o rezultatima sa i bez selekcije značajki u poglavlju 1.5.

1.2.1 Selekcija značajki po sadržaju informacije

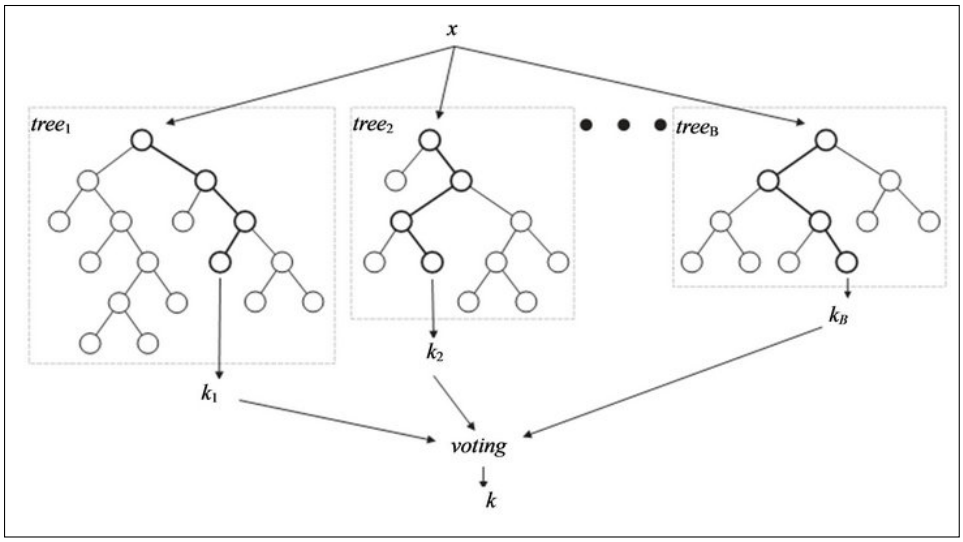
Svaka pojedinačna značajka vektora značajki nosi sa sobom neku količinu ili sadržaj informacije(eng. *information gain*), te nam ona govori koliko je ta značajka bitna za klasifikaciju. Selektiramo samo one značajke koje sa sobom nose najveći sadržaj informacije. Definicija sadržaja informacije je detaljnije opisana u 1.3.2.

1.2.2 Selekcija značajki po iznosu varijance

Značajke su brojevi pa nad njima možemo računati razne statistike pa tako i varijancu. Ova selekcija značajki odbacuje sve značajke koje ne pređu unaprijed određenu granicu(eng. *threshold*) iznosa varijance.

1.3 Slučajna šuma

Slučajna šuma [5] je klasifikator koji se sastoji od kolekcije nezavisnih stabala odlučivanja. Svako od stabala predstavlja jedan glas u većinskom donošenju odluke. Odluka se donosi zbrajanjem glasova te se odabire odluka s najvećim brojem glasova [6]. Ovo detaljnije možemo vidjeti na slici 1.1. Slučajna šuma jer je u svojoj osnovi samo skup stabala vrlo dobro podnosi veliku dimenzionalnost podataka (što za ovaj problem očekujemo) i ne očekuje linearnu odvojivost vektora značajki te je iz tih razloga odabrana kao korišteni algoritam.



Slika 1.1: Arhitektura slučajne šume [7]

Svako od N stabala odluke je izgrađeno nasumičnim uzorkovanjem s ponavljanjima skupa za treniranje tako da se uzorkuje podskup duljine *N*. Stabla se grade do maksimalne moguće dubine iako postoje instance algoritma u kojem se stabla podrezuju. U izgradnji stabla ponovno se slučajno odabire podskup značajki kojih ima *M*. Veličina tog podskupa je hiperparametar algoritma, u literaturi [8]

se za klasifikacijski problem preporuča veličina od $\sqrt{M}$. Od tog podskupa treba odabrati najbolju značajku koja će biti iskorištena za idući čvor stabla. Odabir najbolje značajke uobičajeno se radi metodama Gini nečistoće ili uzajamnog sadržaja informacije.

1.3.1 Gini nečistoća

Gini nečistoća je mjera koliko često bi nasumično odabrana značajka iz nekog skupa bila krivo klasificirana ako bi ju se nasumično klasificiralo s obzirom na to kakva je razdioba značajki po razredima u podskupu svih značajki. Drugim riječima gini nečistoća je kriterij koji teži minimizaciji vjerojatnosti krive klasifikacije [9]. Računamo ju na sljedeći način [10]:

$$I_g(t) = 1 - \sum_{i=1}^{c} p(i|t)^2 \tag{1.1}$$

gdje je $p(i|t)$ broj značajki koje pripadaju klasi i za čvor t.

1.3.2 Uzajamni sadržaj informacije

Uzajamni sadržaj informacije je koncept baziran na entropiji. Entropija je definirana kao količina informacije koju nosi neka poruka te ju računamo:

$$H(t) = -\sum_{i} p(x_i) * log_2 p(x_i) \tag{1.2}$$

gdje su $p(x_i)$ vjerojatnosti svake od klasa.
Uzajamni sadržaj informacije definiran je kao:

$$I(X; Y_i) = H(X) - H(X|Y_i) \tag{1.3}$$

gdje je X klasa(autor), a Y_i i-ta značajka iz skupa. Intuitivno ga možemo zamisliti kao količinu informacije koju daje značajka i za klasu kojoj pripada.

1.4 Prikupljanje podataka

Izvorni kodovi korišteni u eksperimentima djelo su učenika srednjih i osnovih škola koji su se natjecali na HONI-u[1] u godini 2016-2017. Skupljena su dva skupa podataka, jedan od 216 autora gdje svaki od njih ima 10 izvornih kodova i drugi od

[1] http://www.hsin.hr/honi/

29 autora s također 10 izvornih kodova. Razlika je u tome što su u većem skupu izvorni kodovi birani nasumično iz skupa svih rješenja autora, dok su u manjem skupu oni odabrani kao rješenja istih 10 zadataka. Cilj ovakvog odabira je dobiti što objektivniju točnost naučenog klasifikatora. Oba skupa su dostupna na poveznici[2] s dna stranice. HONI je odabran jer se činilo zanimljivo isprobati algoritam na domaćem natjecanju iako bi rezultati bilo vjerojatno bolji da se koristilo natjecanje poput Google Code Jam-a[3] koje je znatno veće i nudi više mogućnosti.

1.5 Rezultati i rasprava

Napravljeno je nekoliko različitih eksperimenata s različitim hiperparametrima i selekcijom značajki kako bi dobili što bolju sliku kako algoritam radi. Svaki od eksperimenata je odrađen k-preklop među validacijom(eng. *k-fold cross validation*), gdje je k u našem slučaju 10 jer imamo po 10 rješenja za svakog autora, kako bi dobili što objektivnije i točnije rezultate. U svakoj iteraciji 9 izvornih kodova autora birano je u skup za treniranje, a preostali izvorni kod u skup za testiranje. Svako od stabala unutar slučajne šume građeno je do samog kraja te je za odabir novog čvora korišteno $\sqrt{N}$ značajki kako je to preporučeno u [8]. Ovi hiperparametri su se pokazali daleko najboljima te ih nije bilo potrebe mijenjati te je jedini varijabilni hiperparametar bio broj drveća unutar slučajne šume. Također su prikazane po dvije slike za svaki eksperiment, jedna sa skupom od leksičkih i strukturnih značajki te jedna sa skupom leksičkih, strukturnih i sintaksnih značajki. To je napravljeno kako bi što bolje shvatili koliko su koje značajke bitne za prepoznavanje autora.

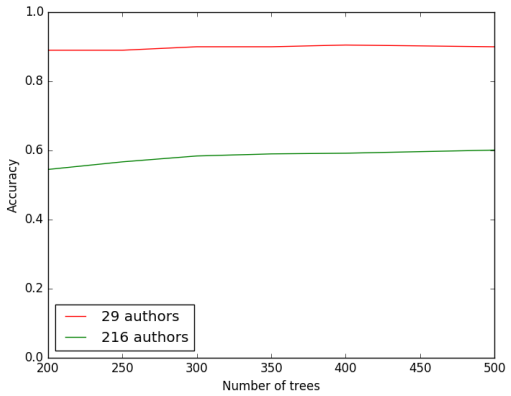
1.5.1 Rezultati bez selekcije značajki

U ovom poglavlju prikazani su rezultati bez selekcije značajki. Krenulo se s pretpostavkom da će nam sintaksne značajke donijeti veliko poboljšanje klasifikatora što spominju i [3], no kao što možemo vidjeti na slikama 1.2 i 1.3 sintaksne značajke nisu donijele veliko poboljšanje klasifikatora na manjem skupu autora, dok su na

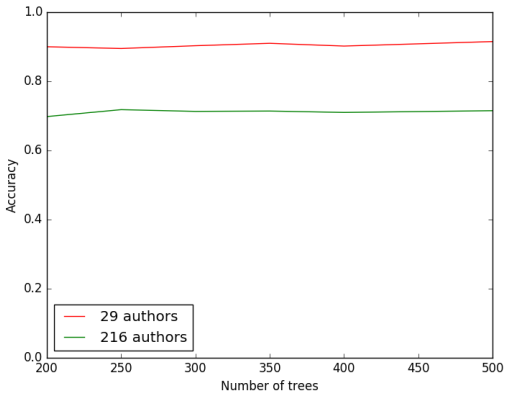
[2] `https://www.dropbox.com/s/jqtsj2zcb285rti/master_thesis_dataset.zip?dl=0`

[3] `https://code.google.com/codejam/`

većem skupu autora ukupnu točnost čak dosta i smanjili.



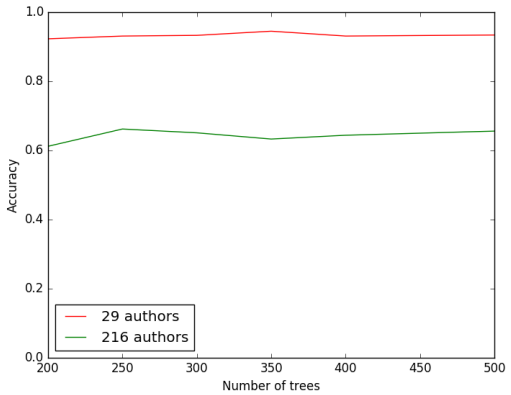
Slika 1.2: Točnost klasifikatora uz leksičke, strukturne i sintaksne značajke



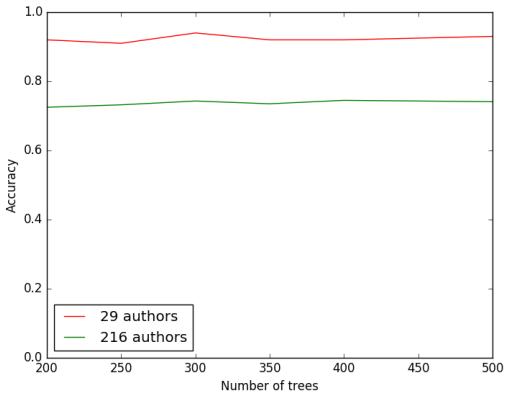
Slika 1.3: Točnost klasifikatora uz leksičke i strukturne značajke

1.5.2 Rezultati uz selekciju značajki sadržajem informacije

U ovom poglavlju predstavljam rezultate kada je korištena selekcija značajki uzajamni sadržajem informacije koja bi zbog prirode značajki koje su dosta rijetke trebala povećati točnost klasifikatora s obzirom na slučaj bez selekcije značajki što se doista i pokazalo istinito te to možemo vidjeti na slikama 1.4 i 1.5. Primjećujemo da su nam sintaksne značajke i u ovom slučaju donijele pogoršanje točnosti.
Također pošto selekcija značajki smanjuje dimenzionalost vektora značajki sa stotina tisuća na desetke tisuća donosi nam veliko ubrzanje učenja klasifikatora.



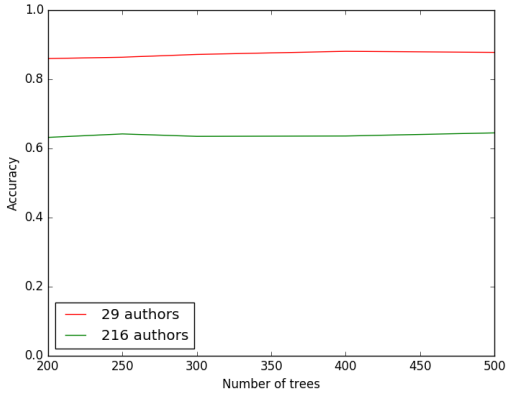
Slika 1.4: Točnost klasifikatora uz leksičke, strukturne i sintaksne značajke



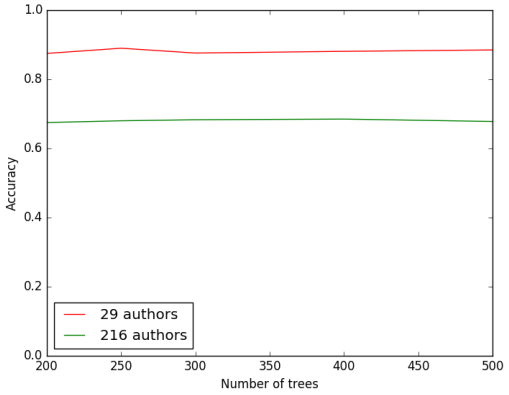
Slika 1.5: Točnost klasifikatora uz leksičke i strukturne značajke

1.5.3 Rezultati uz selekciju značajki varijancom

U ovom poglavlju predstavljam rezultate korištenjem selekcije značajki varijancom, u eksperimentu je korištena granica varijance od 0.03 jer je pokazala najbolje rezultate. Možemo primjetiti da su rezultati bolji nego kada smo koristili sve značajke, no lošiji su nego sa selekcijom uzajamnim sadržajem informacije što vidimo na slikama 1.6 i 1.7.



Slika 1.6: Točnost klasifikatora uz leksičke, strukturne i sintaksne značajke



Slika 1.7: Točnost klasifikatora uz leksičke i strukturne značajke

1.5.4 Rasprava

Prva stvar koju možemo primjetiti da što manji broj autora predamo klasifikatoru na treniranje to su nam rezultati bolji, što ima i smisla jer mora odlučiti na manjem broju kategorija, a ima jednako izvornih kodova za svakog autora kao u slučaju s velikim brojem autora. Sintaksne značajke nam uglavnom donose lošije rezultate nego što smo pretpostavljali te vidjeli u [3]. Dodatno one donose stotine tisuća značajki i znatno usporavaju treniranje te su iz tih razloga izbačene za potrebe web aplikacije *Turtle*. Kao zaključak naveo bih da su rezultati dovoljno dobri za potrebe detekcije plagijata, ali ipak nisu ni blizu rezultatima iz [3]. Može se primjetiti da se ovako definiran skup značajki izvornog koda može primjeniti na skoro svaki poznatiji programski jezik te bi to bilo zanimljivo isprobati u budućem radu.

2. Određivanje sličnosti izvornih kodova

Ne postoji sustav specijaliziran za detekciju plagijata koji može sa sto postotnom sigurnošću utvrditi da je nešto plagijat. Kako bi odredili plagijat potreban je ljudski faktor. Odmah možemo uočiti da to nije baš uvijek efikasno, kada bi morali pronaći plagijate među tisućama dokumenata čovjeku bi trebalo puno vremena. Upravo iz tog razloga razvijamo sustav koji bi odredio sličnost među parovima dokumenata te izbacio parove za koje smatra da nikako ne mogu biti plagijat te uvelike ubrzao i olakšao posao ljudima. Dokumenti mogu biti teksutalne datoteke, izvorni kodovi, pjesme, itd. U ovom radu naglasak je na detekciji plagijata izvornih kodova te je u nastavku to detaljnije opisano.

Određivanje sličnosti izvornih kodova u svrhu detekcije plagijata je relativno neistraženo područje. Postoje dva vrlo slična, ali sada već stara sustava (nastali su prije više od 10 godina op.a) [11] [12] koji se baziraju na računanju otiska(eng. *documentfingerprint*) izvornog koda algoritmom *winnowing* koji je detaljnije opisan u [13]. *Turtle*, kao i sustav implementiran na završnom radu [1], koristi *winnowing* za račun otiska izvornog koda te preoblikovanje izvornog koda prije nego se otisak računa(detaljnije opisano u radu [14]), no s dodatnim poboljšanjima. Neka od poboljšanja su veći broj preoblikovanja, što rezultira u boljim mjerama sličnosti i brže ukupno vrijeme. U nastavku poglavlja detaljnije su opisani koraci kojim sustav dolazi do mjera sličnosti, najvažnije pomoćne strukture podataka i algoritmi te usporedba s performansama starog sustava.

2.1 Izračun otiska izvornog koda

Otisak izvornog koda definiran je kao jedan podskup iz skupa izračunatih sažetaka (eng. *hash*) k-grama [13]. K-gram je uzastopni podniz duljine k. Odmah primjećujemo da k-grama ima $n - k - 1$ za znakovni niz duljine n te su ta dva broja vrlo

blizu za manje vrijednosti k. Iako nigdje u literaturu nije navedeno koliko točno bi trebao biti k, najbolje rezultate su pokazale manje vrijednosti(otprilike prosjek duljine prosječnih duljina ključnih riječi programskog jezika na kojem se radi). Iz tog razloga potreban nam je efikasan algoritam kako bi izračunali sažetke svakog k-grama. Jedan od takvih algoritama je *Rabin-Karp* [15].

2.1.1 Rabin-Karp algoritam

Rabin-Karp je algoritam koji nam u linearnoj $O(n)$ složenosti računa sažetke svih k-grama ulaznog znakovnog niza. Algoritam radi tako da postupno gradi rješenje, iz i^{tog} sažetka se računa $i+1$. Ako k-gram zapišemo kao $c_1...c_k$ tada sažetak računamo:

$$H(c_1...c_k) = c_1 * b^{k-1} + c_1 * b^{k-2} + ... + c_k \tag{2.1}$$

gdje su b baza, a c_i ascii vrijednost i^{tog} znaka. Za bazu se uzima prost broj kako bi se izbjegle kolizije.
Idući sažetak tada računamo tako da prethodno izračunatom sažetku oduzmemo prvi član, pomnožimo sve s b i dodamo novi znak:

$$H(c_2...c_k + 1) = (H(c_1...c_k) - c_1 * b^{k-1}) * b + c_{k+1} \tag{2.2}$$

Svaki uzastopni sažetak izračunat je uz samo tri operacije i tu možemo shvatiti linearnu složenost algoritma. Bitno je naglasiti da se iz ulaznog znakovnog niza prije računanja izbacuju nebitni znakovni poput razmaka.

2.1.2 Winnowing

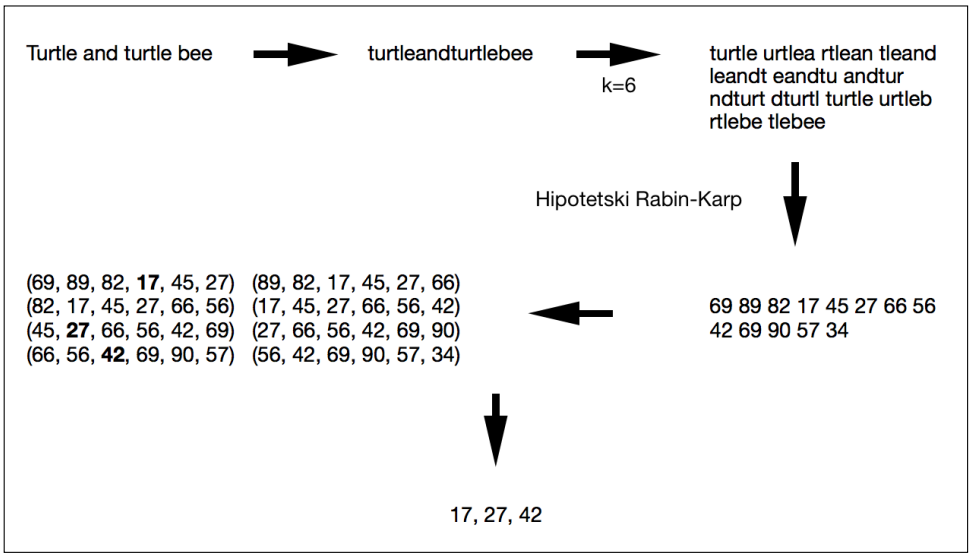
Winnowing je algoritam za računanje otiska izvornog koda. Na ulaz prima niz sažetaka ranije izračunatih *Rabin-Karp* algoritmom te ispisuje izračunati otisak. U nastavku je pseudokod algoritma:

Algorithm 1 Winnowing

Ulaz: K – niz sažetaka k-grama.
Izlaz: otisak F.

```
windows := split_to_windows(w)
for (i := 0; i < n; ++i) do
    w_min := min(windows[i])
    if (more_than_one_same_val) then
        if (w_min is rightmost) then
            F.append(w_min)
        end if
    end if
end for
return F
```

Primjer rada *Rabin-Karp* i *Winnowing* algoritama vidimo na slici 2.1.



Slika 2.1: Primjer rada algoritama $Rabin - Karp$ i *Winnowing*

2.2 Preoblikovanje izvornog koda

Glavni razlog za korištenje preoblikovanja koda prije nego krenemo računati otisak je taj što algoritmi za računanje otiska pa tako i *winnowing* nisu otporni na mnoge pokušaje izmjene koda kako ih se ne bi detektiralo kao plagijate. Nabrojimo neke od njih:

- Dodavanje beskorisnih komentara
- Dodavanje beskorisnog koda
- Dodavanje praznina
- Zamjena imena varijabli
- Promjena jedne petlje u drugu
- Rastav naredbi na podbaredbe
- Izdvajanje koda u funkcije
- Promjena redoslijeda naredbi

Odmah možemo primjetiti da je *winnowing* otporan na dodavanje praznina jer se one brišu prije računanja sažetaka. Također izdvajanje koda u funkcije se detektira jer nema razlike gdje točno se isječci koda nalaze za rezultat algoritma. Za ostale modifikacije trebamo napisati posebne funkcije koje će preoblikovati izvorni kod kako bi i one mogle biti detektirane. Naravno, modifikacije kao što je rastav naredbi na podnaredbe je jako teško detektirati bez korištenja jezičnog prevodica i neke vrste sintaksnih stabala te se tu limitiramo i njih *Turtle* ne zna detektirati. U nastavku poglavlja opisana su korištena preoblikovanja izvornog koda. Treba naglasiti da ovakvih preoblikovanja može biti jako puno te je implementacija dizajnirana tako da se svako preoblikovanje može vrlo lako dodati.

2.2.1 Zamjena imena varijabli

Ideja je da se sva imena varijabli zamjene univerzalnim imenom, u ovom slučaju odabrano je ime *var*. Također mijenjamo i sve konstante kako bi mogli detektirati i

promjenu redoslijeda naredbi. Prvo sve te varijable moramo pronaći unutar izvornog koda, a to radimo tako da izgradimo automat za željeni programski jezik koji će nam znati reći kada smo u stanju imena varijable. Kako bi automat znao razlikovati ključnu riječ od varijable potrebno je izgraditi strukturu podataka koja će mu reći promatra li ključnu riječ jezika. Za te potrebe izgrađeno je prefiksno stablo [16] od svih ključnih riječi koje nam tu funkcionalnost omogućava u jako brzom vremenu, zbog male duljine ključnih riječi možemo reći u $O(1)$. Implementirani automat također prepoznaje stanje *komentar* te izbacuje sve komentare iz izvornog koda pa time rješavamo i modifikaciju dodavanjem komentara. U dodatku B B vidimo izgled izvornog koda prije i poslije izvršavanja automata.

2.2.2 Zamjena while petlji u for petlje

Ovdje je opisan postupak za programski jezik $C++$, no postupak se može uz male preinake primjeniti i na sve poznatije programske jezike. Prvo primjetimo oblik *while* petlje:

```
1   init; while(test){ body; post; }
```

te ga zapišimo kao *for* petlju:

```
1   for(init; test; post){ body; }
```

Sada samo pronađemo ovakve isječke unutar koda i zamijenimo ih, npr. korištenjem regeks operacija.

2.3 Izračun sličnosti izvornih kodova

Nakon što smo izračunali otisak izvornog koda još samo trebamo izračunati sličnosti svih parova. Najprije se izračuna mapa indeksa(eng. *index mapping*) koja za svaki otisak sprema sve izvorne kodove u kojem se on nalazi. Nakon toga računaju se dvije sličnosti, sličnost prvog koda prema drugom kodu x_{12} te sličnost drugog koda prema prvom kodu x_{21} i to tako da se prebroje pojavljivanja otisaka koji su im zajednički te se taj broj normalizira ukupnim brojem otisaka prvog člana para. Dvije sličnosti imamo kako bi što bolje opisali slučajeve kada imamo dva izvorna

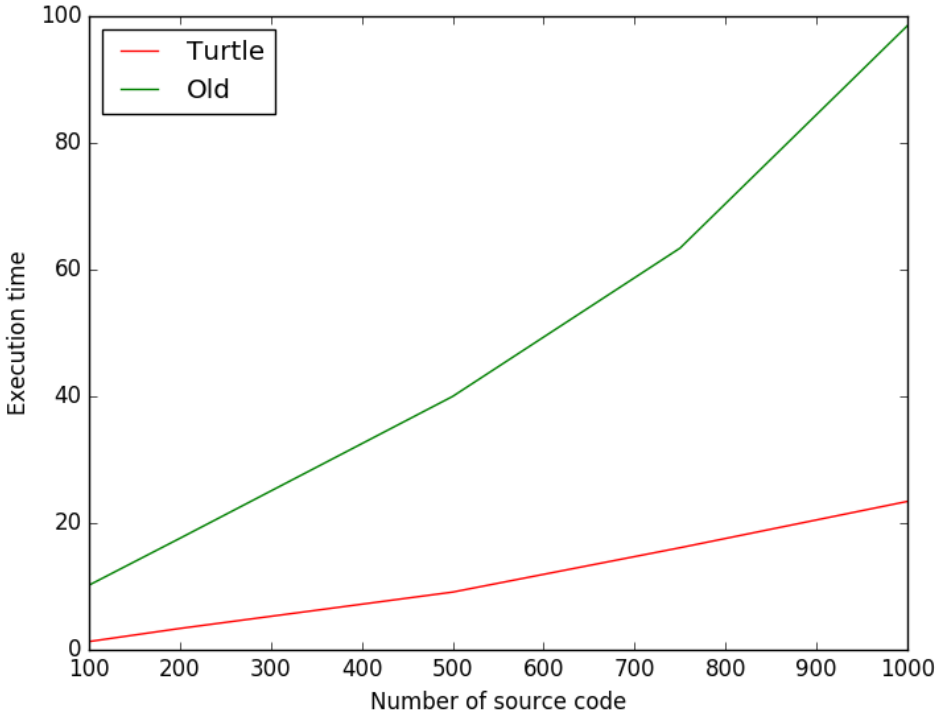
koda koja imaju veliku razliku u broju linija. S ovakvim računanjem sličnosti također rješavamo problem dodavanja beskorisnog koda prethodno opisanog. Za kraj nam samo ostaje definirati funkciju koja bi ove dvije sličnosti transformirala u jednu kako bi parove mogli sortirati:

$$f(x_{12}, x_{21}) = (x_{12} + x_{21}) * min(x_{12}, x_{21}) \tag{2.3}$$

Odabrana je ova funkcija jer se u praksi pokazala kao odličan izbor. Ova funkcija nam također filtrira parove kako ne bi korisniku prikazivali parove koji sigurno nisu plagijati i to tako da odredimo granicu (eng. *threshold*) preko koje vrijednost funkcije f mora preći.

2.4 Usporedba sa starim sustavom

Kao što je spomenuto ranije ovaj sustav je novo implementirana verzija sustava implementiranog za završni rad te su u ovom poglavlju uspoređeni ti sustavi. Najveća prednost novog sustava je u njegovoj brzini što vidimo na slici 2.2.



Slika 2.2: Usporedba vremena izvršavanja

Mjereno je vrijeme od primitka izvornih kodova pa sve do ispisa parova sličnosti. Možemo primjetiti da oba sustava imaju linearnu složenost, no zbog puno bolje i pametnije implementacije *Turtle* je i do pet puta bolji (slika 2.2). *Turtle* također zbog više preoblikovanja izvornog koda nudi bolju detekciju nekih modifikacija(npr. zamjena for u *while* petlje). Ostatak modifikacija te izračunatih sličnosti nisu posebno opisivani jer su vrlo slični već opisanima u [1].

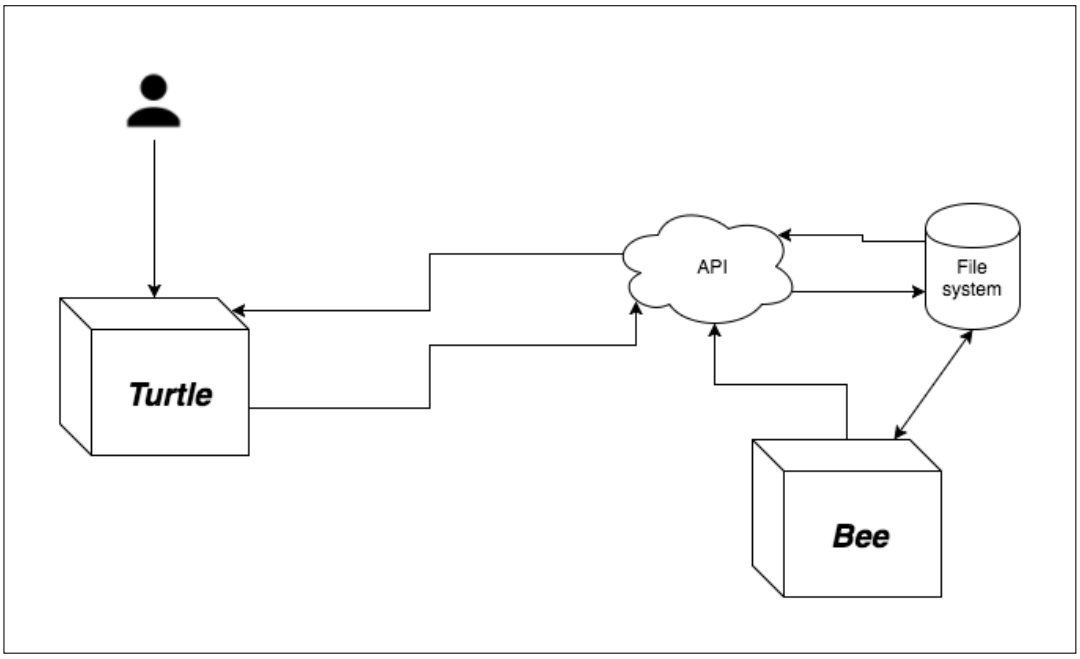
3. Razvijeni sustav za detekciju plagijata te utvrđivanje autorstva izvornih kodova

Razvijeni sustav za detekciju plagijata sastoji se od dvije komponente čiji su algoritmi opisani u prethodnim poglavljima. Ovdje ukratko opisujemo njihove implementacije te arhitekturu cijelog sustava. Prva komponenta razvijenog sustava je utvrđivanje autorstva izvornog koda te je ona razdvojena u poseban projekt nazvan *Bee*. Ovo omogućuje lakši razvoj algoritama strojnog učenja s kojim utvrđujemo autorstvo te nam daje veću fleksibilnost za korištenje te komponente izvan konteksta detekcije plagijata. *Bee* je dizajniran kao skripta komandne linije kojoj se predaju skup za treniranje i skup za testiranje te koja vraća točnost naučenog klasifikatora jer nam taj način nudi lako isprobavanje raznih konfiguracija i skupova izvornih kodova. *Turtle* je druga i ujedno glavna komponenta sustava koja određuje sličnosti izvornih kodova te nudi korisnicima web sučelje za pristup algoritmima detekcije plagijata. Korisnici mogu vidjeti sortirane parove izvornih kodova te njihove pripadne sličnosti nakon što uplodaju željenu datoteku s izvornim kodovima. Sustav nudi i detaljniji uvid u parove izvornih kodova i to tako da jednakim bojama boja slične dijelove izvornih kodova. Arhitektura cijelog sustava detaljnije je opisana je u poglavlju 3.1. Sustav se također spaja na sustav *Bee* te je moguće trenirat klasifikator i predviđati autore kroz njegove sučelje.

Ovaj sustav je prvi sustav koji nudi korisnicima detekciju plagijata kroz dva različita načina te je ovo veliko unaprijeđenje na sve dosadašnje sustave za detekciju plagijata izvornih kodova. *Turtle*, a također i *Bee* trenutno podržavaju samo programski jezik $C++$ iako je u implementacijama obje komponente ostavljenja mogućnost lakog dodavanja novih jezika, ali to je ostavljeno za budući rad. Ovo poglavlje opisuje i korisničko sučelje sutava *Turtle*.

3.1 Arhitektura sustava

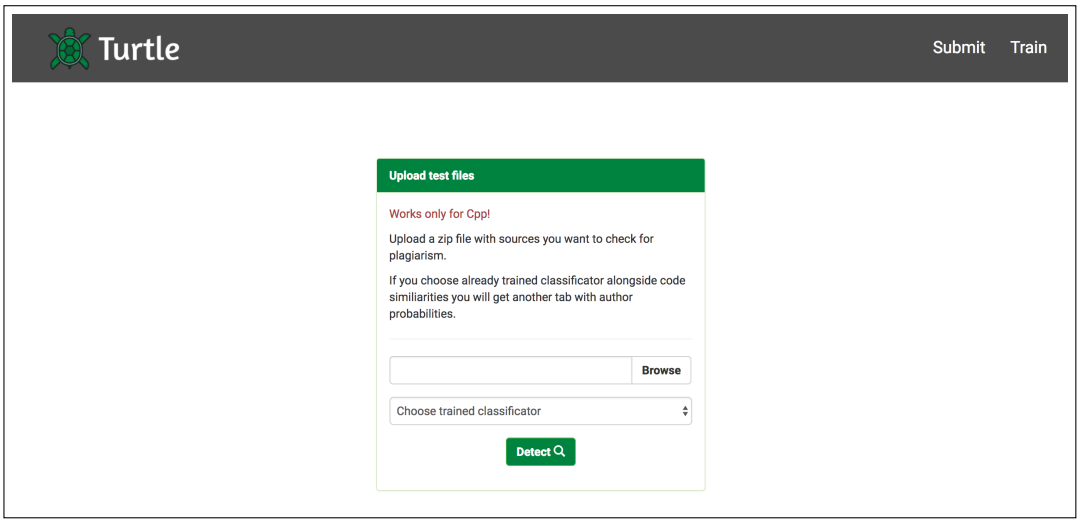
Spomenuto je da je *Bee* dizajniran kao skripta komandne linije te je bilo potrebno nekako ga spojiti s drugom komponentom. To je napravljeno tako da je napisan jednostavan web server koji nudi nekoliko API endpoint-a na korištenje ostalim aplikacijama. Ti endpoint-i izvršavaju metode poput učenja klasifikatora ili predviđanja autora. Ovakav dizajn nam nudi mogućnost korištenja utvrđivanja autorstva i izvan konteksta sustava *Turtle*, a njemu nudi lagano povezivanje. Klijentski kod *Turtle* aplikacije se kada to korisnik zatraži spaja na API endpoint-e ponuđene iz *Bee* te ispisuje vraćene podatke korisniku na ekran. Sve ovo detaljnije možemo vidjeti na slici 3.1.



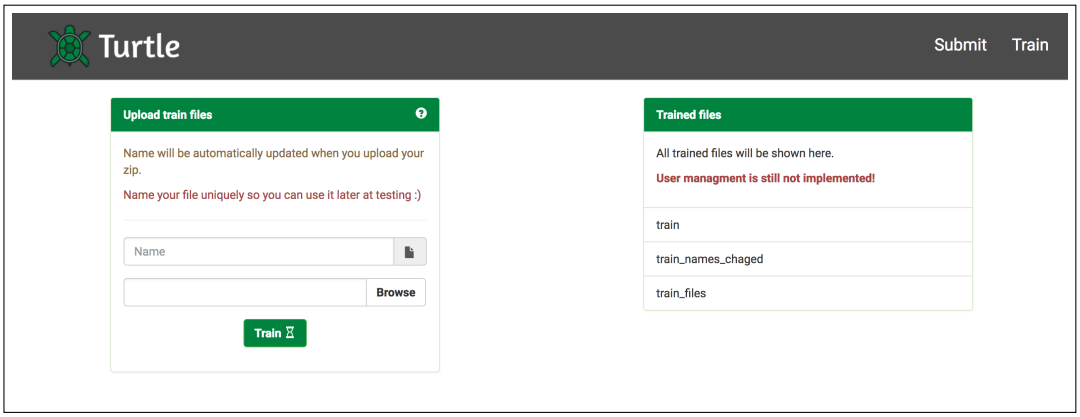
Slika 3.1: Arhitektura razvijenog sustava

3.2 Korisničko sučelje

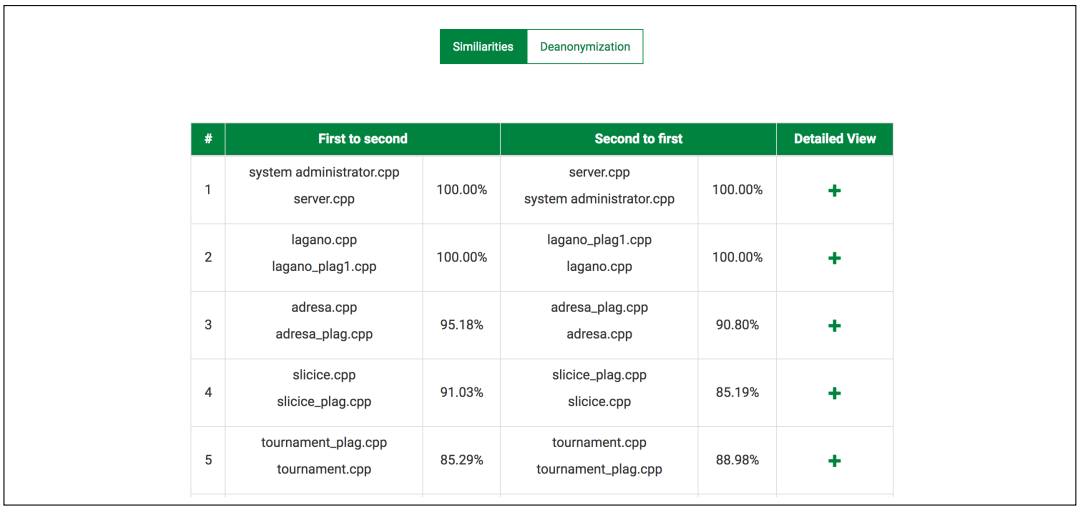
U ovom potpoglavlju opisano je korisničko sučelje sustava *Turtle* kroz nekoliko slika.



Slika 3.2: Početna stranica, ovdje uploadamo zip datatoteku koju želimo provjeriti, također možemo izabrat unaprijed trenirani klasifikator kako bi mogli dobili i najvjerojatnije autore svakog koda.

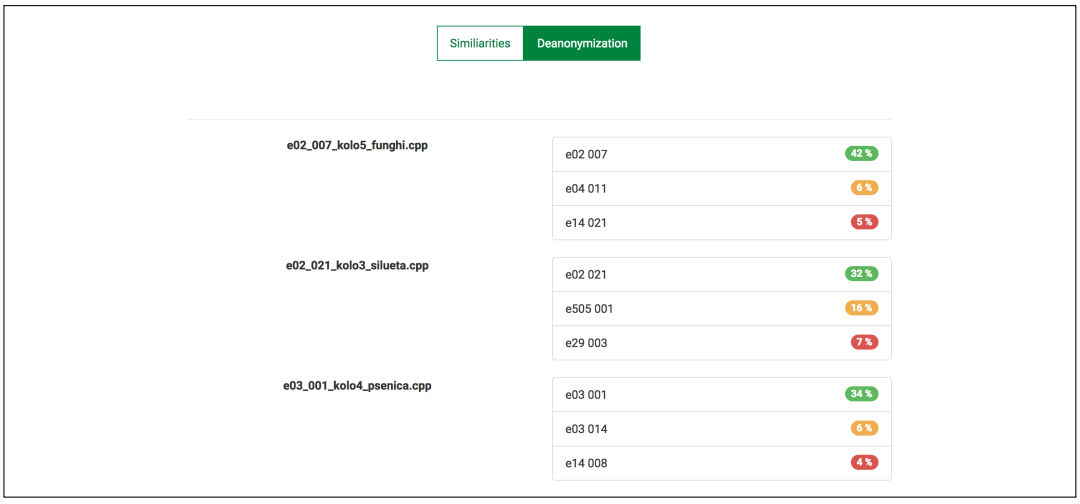


Slika 3.3: Stranica za treniranje klasifikatora, uploada se zip datoteka te nakon što je učenje završeno naučeni klasifikator se pojavi u listi desno.



Slika 3.4: Nakon što sustav pronađe sve sličnosti ispiše nam ih u tablicu sortirane od najveće do najmanje sličnosti.

Slika 3.5: Ako kliknemo na više detalja koji postoji za svaki par izvornih kodova dobijemo ovakav prozor gdje su obojani dijelovi koda koji su slični i što nam uvelike olakšava detekciju plagijata.



Slika 3.6: Ako smo uz detekciju plagijata odredili da želimo i provjeriti tko bi mogli biti autori naših kodova na tabu *Deanonymization* dobijemo za svaki izvorni kod listu od 3 najvjerojatnija autora.

3.3 Implementacijski detalji

3.3.1 Turtle

Za implementaciju korišten je programski jezik *Python 2.7.* Web server napisan je u random okviru *Flask* [1]. Klijentski dio aplikacije implementiran je u *ReactJS* [2] frameworku kako bi korisnici imali što bolji doživljaj korištenja aplikacije te zbog asinkronog dohvata podataka.

3.3.2 Bee

Za implementaciju korišten je programski jezik *Python 2.7* te njegova biblioteka *scikit-learn* [3] koja nudi pristup mnogim algoritmima strojnog učenja na vrlo jednostavan način. Za implementaciju slučajne šume korištena je klasa *RandomForestClassifier* [4]. Selekcija značajki uzajamnim sadržajem informacije implementirana je pomoću klase *ExtraTreeClassifier* [5] koji omogućava biranje idućih čvorova uzajamnim sadržajem informacije (parametar "criterion") te nam nakon učenja omogućava pristup najbitnijim značajkama. Selekcija značajki iznosom varijance implementirana je korištenjem klase *VarianceThreshold* [6]. Web dio komponente napisan je u radnom okviru *Flask*.

[1] http://flask.pocoo.org/

[2] https://facebook.github.io/react/

[3] `http://scikit-learn.org/stable/`

[4] `http://scikit-learn.org/stable/modules/generated/sklearn.ensemble.RandomForestClassifier.html`

[5] `http://scikit-learn.org/stable/modules/generated/sklearn.ensemble.ExtraTreesClassifier.html`

[6] `http://scikit-learn.org/stable/modules/generated/sklearn.feature_selection.VarianceThreshold.html`

Zaključak

Ovaj sustav bio bi od velike pomoći velikom broju fakulteta, škola i brojnim organizatorima programerskih natjecanja jer u relativno kratkom roku može obraditi veliki broj podataka što je i dokazano testovima i uvelike bi smanjio pokušaje pisanja plagijata. Iako rezultati iz literature za deanonimizaciju autora nisu postignuti, opet su bili dovoljno dobri kako bi se mogli koristiti unutar sustava.

Duboko vjerujem da bi se ovaj sustav trebao nastaviti razvijati te kao najbitniju stavku proširenja vidim dodavanje novih programskih jezika poput *Pythona* i *Jave* jer se ti jezici se danas najviše koriste za pisanje laboratorijskih vježbi na fakultetima. Dalje kako bi bili otporniji na više modifikacija izvornog koda treba napisat dodatne postupke preoblikovanja koda. Trebalo bi smisliti i nove leksičke, strukturne i sintaksne značajke kako bi deanonimizacija autora radila bolje te isprobati kako bi se ponašali drugačiji klasifikatori na ovom problemu.

Kao završni cilj *Turtle* vidim kao sustav koji će fakultetima omogućiti lagano upravljanje laboratorijskim vježbama za niz predmeta koji bi uključivao sve od predaje rješenja do evaluacije tih rješenja te detekcije plagijata.

Literatura

[1] D. Rakipović. *Sustav za detekciju plagijata*. 2015.

[2] R. Kohavi i F. Provost. *Glossary of Terms*. URL: http://ai.stanford.edu/~ronnyk/glossary.html.

[3] A. Caliskan-Islam i dr. *De-anonymizing Programmers via Code Stylometry*. 2014. URL: https://www.princeton.edu/~aylinc/papers/caliskan-islam_deanonymizing.pdf.

[4] F. Yamaguchi. *Joern documentation*. 2017. URL: http://www.mlsec.org/joern/.

[5] L. Breiman. *Random Forests*. Machine Learning 45. 2001.

[6] Nikola Bogunović. *Algoritmi strojnog učenja - 1, Strojno učenje*. predavanja, http://www.zemris.fer.hr/predmeti/kdisc/Algoritmi_1.ppt. Sveučilište u Zagrebu, Fakultet elektrotehnike i računarstva, Hrvatska.

[7] C. Nguyen, Y. Wang i H. M. Nguyen. "Random forest classifier combined with feature selection for breast cancer diagnosis and prognostic". *Journal of Biomedical Science and Engineering* 6 (2013), str. 3.

[8] T. Hastie, R. Tibshirani i J. Friedman. *The elements of statistical learning*. Springer, 2009.

[9] *DecisionTree*. predavanja, http://www.cse.msu.edu/~cse802/DecisionTrees.pdf. Michigan State University, USA.

[10] Sebastian Raschka. https://sebastianraschka.com/faq/docs/decision-tree-binary.html.

[11] Aiken A. URL: https://theory.stanford.edu/~aiken/moss/.

[12] Karlsruhe Institute of Technology. URL: https://jplag.ipd.kit.edu/.

[13] S. Schleimer, D. S. Wilkerson i A. Aiken. *Winnowing: Local Algorithms for Document Fingerprinting*. URL: http://theory.stanford.edu/~aiken/publications/papers/sigmod03.pdf.

[14] D. Šulc. *Algoritmi i metode mjerenja sličnosti izvornog koda*. 2015.

[15] R. M. Karp i M. O. Rabin. *Pattern-matching algorithms*. 1987.

[16] Paul E. Black. *trie*. online, Dictionary of Algorithms and Data Structures. 2011. URL: https://xlinux.nist.gov/dads/HTML/trie.html.

Sustav za detekciju plagijata te određivanje autorstva izvornih kodova

Sažetak

Plagijati postaju sve veći problem u današnjem svijetu, zbog ogromnog rasta broja informacija na internetu vrlo se jednostavno može ukrasti tuđi rad te predstaviti ga kao svoj, tj. plagirati. U sklopu ovog diplomskog rada naglasak je bio na pronalasku plagijata među izvornim kodovima te smo tom problemu pristupili iz dva kuta, prvi je pronalazak sličnosti među parovima izvornih kodova korištenjem otisaka izvornih kodova dok je drugi određivanje autora izvornog koda metodama strojnog učenja. Na kraju su ova dva pristupa spojena u web aplikaciju *Turtle* koja vrlo uspješno i u kratkom roku pronalazi plagijate među velikim brojem izvornih kodova(npr. laboratorijske vježbe ili programerska natjecanja).

Ključne riječi: Strojno učenje, plagijati, deanonimizacija, detekcija plagijata, izvorni kod

System for plagiarism and authorship detection of source code

Abstract

Plagiarism is becoming bigger and bigger problem in today society, the reason behind it is that information available online is growing exponentialy and it's easy to steal from other authors and present it like your work. This thesis tackles the problem of detecting source code plagiarism from two different approaches, one is computing similiarities between source code using source code fingerprints and other is deanonymizing authors of source code using machine learning. This approaches are then connected to a web application *Turtle*. This application can detect plagiarism between large number of source code(e.g. laboratory assignments, programming competitions) rather well.

Keywords: Machine learning, plagiarism, deanonymization, plagiarism detection, source code

A. Tipovi čvorova apstraktnog sinstaksnog stabla

Tablica A.1: Tipovi čvorova apstraktnog sintaksnog stabla

Additive Expression	AndExpression	Argument	ArgumentList
ArrayIndexing	AssignmentExpr	BitAndExpression	BlockStarter
BreakStatement	Callee	CallExpression	CastExpression
CastTarget	CompoundStatement	Condition	ConditionalExpression
ContinueStatement	DoStatement	ElseStatement	EqualityExpression
ExclusiveOrExpression	Expression	ExpressionStatement	ForInit
ForStatement	FunctionDef	GotoStatement	Identifier
IdentifierDecl	IdentifierDeclStatemetn	IdentifierDeclType	IfStatement
IncDec	IncDecOp	InclusiveOrExpression	InitializerList
Label	MemberAccess	MultiplicativeExpression	OrExpression
Parameter	ParameterList	ParameterType	PrimaryExpression
PtrMemberAccess	RelationalExpression	ReturnStatement	ReturnType
ShiftExpression	Sizeof	SizeofExpr	SizeofOperand
Statement	SwitchStatemetn	UnaryExpression	UnaryOp
UnaryOperator	WhileStatement		

B. Primjer izvornog koda prije i poslije automata

```
1 #include <cstdio>
2 #include <iostream>
3
4 using namespace std;
5
6
7
8 int main () {
9
10   int a=0;
11   int sol=0;
12
13   for (int i=0; i<4; i++) {
14     cin >>a;
15     sol+=min (a, 12-a);
16   }
17
18   cout <<sol;
19
20   return 0;
21 }
```

Listing B.1: Prije

```
1 using namespace std;
2
3
4
5 int var () {
6
7   int var=var;
8   int var=var;
9
10  for (int var=var; var<var; var++) {
11    cin >>var;
12    var+=var (var, var-var);
13  }
14
15  cout <<var;
16
17  return var;
18 }
```

Listing B.2: Poslije