Effect of source-code preprocessing techniques on plagiarism detection accuracy in student programming assignments

Novak, Matija

Doctoral thesis / Disertacija

2020

Degree Grantor / Ustanova koja je dodijelila akademski / stručni stupanj: **University of Zagreb, Faculty of Organization and Informatics / Sveučilište u Zagrebu, Fakultet organizacije i informatike**

Permanent link / Trajna poveznica: https://urn.nsk.hr/urn:nbn:hr:211:528052

Rights / Prava: In copyright / Zaštićeno autorskim pravom.

Download date / Datum preuzimanja: **2024-04-25**

SVEUČILIŠTE U ZAGREBU
FAKULTET ORGANIZACIJE I INFORMATIKE
VARAŽDIN

Repository / Repozitorij:

Faculty of Organization and Informatics - Digital Repository

DIGITALNI AKADEMSKI ARHIVI I REPOZITORIJI

University of Zagreb

FACULTY OF ORGANIZATION AND INFORMATICS

Matija Novak

EFFECT OF SOURCE-CODE PREPROCESSING TECHNIQUES ON PLAGIARISM DETECTION ACCURACY IN STUDENT PROGRAMMING ASSIGNMENTS

DOCTORAL THESIS

Varaždin, 2020

University of Zagreb

FACULTY OF ORGANIZATION AND INFORMATICS

Matija Novak

EFFECT OF SOURCE-CODE PREPROCESSING TECHNIQUES ON PLAGIARISM DETECTION ACCURACY IN STUDENT PROGRAMMING ASSIGNMENTS

DOCTORAL THESIS

Supervisors:
professor Dragutin Kermek, PhD
professor Mike S. Joy, PhD

Varaždin, 2020

Sveučilište u Zagrebu

FAKLULTET ORGANIZACIJE I INFORMATIKE

Matija Novak

UTJECAJ TEHNIKA ZA PREDOBRADU IZVORNOG KODA NA TOČNOST OTKRIVANJA PLAGIJATA U STUDENTSKIM ZADACIMA IZ PROGRAMIRANJA

DOKTORSKI RAD

Mentori:
prof. dr. sc. Dragutin Kermek
prof. dr. sc. Mike S. Joy

Varaždin, 2020.

DOCTORAL THESIS INFORMATION

I. AUTHOR

Name and surname	Matija Novak
Date and place of birth	16.03.1987, Čakovec
Faculty name and graduation date for level VII/I	Faculty of Organization and Informatics, 19 September 2008
Faculty name and graduation date for level VII/II	Faculty of Organization and Informatics, 10 September 2010
Current employment	Faculty of Organization and Informatics

II. DOCTORAL THESIS

Title	Effect of source-code preprocessing techniques on plagiarism detection accuracy in student programming assignments
Number of pages, figures, tables, appendixes, bibliographic information	223 pages, 49 figures, 66 tables, 12 appendices, 201 items of bibliographic information
Scientific area and field in which the title has been awarded	Scientific area Social Sciences, Scientific field Information and Communication Sciences
Supervisors	prof. Dragutin Kermek, PhD prof. Mike Joy, PhD
Faculty where the thesis was defended	Faculty of Organization and Informatics
Thesis mark and ordinal number	157

III. GRADE AND DEFENCE

Date of doctoral thesis topic acceptance	17 July 2017
Date of doctoral thesis submission	03 September 2019
Date of doctoral thesis positive grade	17 December 2019
Grading committee members	prof. Jasminka Dobša, PhD prof. Marjan Krašna, PhD prof. Neven Vrček, PhD
Date of doctoral thesis defence	03 February 2020
Defence committee members	prof. Jasminka Dobša, PhD prof. Marjan Krašna, PhD prof. Zlatko Stapić, PhD
Date of promotion	

SUPERVISOR INFORMATION

Dragutin Kermek received 1999. his Ph.D. in Information Sciences from University of Zagreb, Croatia. From 1986. to 1993. he worked as a programmer, software architect, and project leader at the Center for informatics at the University of Zagreb Faculty of organization and informatics in Varaždin. He has published over 60 research and professional papers in various international and domestic journals, books and conferences. He has participated in several scientific projects. He is a member of: ACM, IEEE Computer Societay, HrOpen, HUJAK. His research interests include Web Engineering, Design Patterns, and E-learning. He served at University of Zagreb Faculty of Organization and Informatics as Vice dean for Academic affairs in 3 consecutive terms from academic year 2005/2006 to 2010/2011. Currently, he is a Full Professor at the Department of Theoretical and Applied Foundations of Information Sciences at the University of Zagreb Faculty of Organization and Informatics. He teaches following courses: Web design and programming, Advanced Web technologies and services, Design patterns, E-learning systems.

Mike Joy is a professor in the Department of Computer Science at the University of Warwick. His research interests focus on educational technology and computer science education, and he is the author or co-author of over 200 journal and conference papers. His current research topics include source-code plagiarism and mobile learning. Professor Joy has master degrees in Mathematics from Cambridge University and in Post-Compulsory Education from the University of Warwick, and a PhD in Computer Science from the University of East Anglia. He is a Chartered Engineer, a Chartered Scientist, a Chartered Fellow of the British Computer Society, a Senior Member of the ACM and a Senior Fellow of the UK Higher Education Academy. Having recently worked on European Union funded projects “BlogForever” and “MALog”, he is currently leading the Erasmus+ project “CONSTRUIT!” which is investigating new principles and tools to facilitate collaboration between educators and learners.

ACKNOWLEDGMENTS

I must admit, the most difficult part of this thesis was to write this page.

Well, where to start? Maybe the best would be to start at the beginning, but the big question is where is the real beginning? Is it the when I enrolled my PhD in November 2013 or is it in July 2013 when my wife (at that point she was not my wife, but you get the point) told me I should apply for a teaching assistant position which also meant to do the PhD. Who knows maybe all started even before.

Looking at the whole PhD journey (a long, sometimes tiring, but joyful journey), there were many people from whom I got help and support. If I would mention everybody, I believe it wouldn't fit on this page. But one thing is sure, without the help and support from my family and friends I wouldn't have made it.

First, I wanna thank my wife for encouraging me to start and helping me go through it. Thank you Maja, without your support I would not have done it. Also, thank you for taking care of the children while I was working, and for listening to my complaints when I needed to talk. You are the one that is keeping this family together. You are a wonderful wife and mother, I love you. I wanna thank my children Helena and Viktor for distracting me in important moments and in this way helping me to relax and then to continue with a fresh mind to work. I apologize for every moment when I was grumpy or angry and not the best version of myself and thank you for keeping up with me.

I am extremely grateful to my parents for their love, patience and help for educating and preparing me for the future. You have supported me since first grade. Thank you for being there for me. Also, I express my thanks to my younger brother who I teased since we were kids but who showed me that I can always trust him that he will help me when it is important.

A big thank you to my supervisors prof. Dragutin Kermek and prof. Mike Joy who have provided invaluable guidance through this research and followed the wise words of Yoda "Always pass on what you have learned." Special mention goes to prof. Kornelije Rabuzin who helped me write my first article back in 2010 and from whom I learned a lot, and my first informatics teacher Dejan Drabić who thought me my first programming language and showed me the beauty of programming.

My special thanks goes to all of my lunch buddies Jelena, Ana, Dijana, Irena, Martina, and Barbara – thank you for bringing the fun to this journey. I want to also mention people from Warwick University – Ayman, Ade, Dimah, and Olfat, it was a real pleasure to meet you and I hope we have a chance to meet again sometimes. Thanks to Ladislav who was the best roommate that one can wish for.

At last, my gratitude goes to everybody else who directly or indirectly helped me to complete this thesis.

ABSTRACT

Plagiarism is a serious problem in academia and students cheat for various reasons, but whatever the reason such behaviour should not be accepted. While it is easy to control plagiarism in classrooms with few students it can be a challenge to do it in a classroom with one hundred students or more. To help teacher detect plagiarism similarity detection tools are built, usually called plagiarism detection tools. While in academia plagiarism can be done in many areas the two most common are textual and programming assignments. In this thesis, the focus is on detecting plagiarism in student programming assignments. Since the tools are not perfect there is always room for improvement and one possibility to improve the plagiarism detection quality is the usage of preprocessing techniques.

Preprocessing techniques have been used in many plagiarism detection tools but there is not much research focusing on the effects of such techniques. To investigate the effect of preprocessing techniques on plagiarism detection tools an experiment was conducted on six tools using five techniques on two different datasets, whereby one dataset is publicly available. To be more precise the six tools were actually three tools whereby each tool had two modes to operate the specialized mode which is specially developed to perform a source-code comparison and textual mode developed for normal text comparison.

In this experiment two hypotheses were stated, one focusing on the differences between the preprocessing techniques and when no preprocessing technique is used and other focusing on differences between two different techniques. In addition to the hypothesis one research question was stated to give more insight into the effects of the preprocessing techniques.

Results of the experiment were analysed quantitatively using the multifactor analysis of variance and qualitatively by analysing the most interesting cases. The whole process of detection and statistical analysis was automated using the newly developed system called Multiple Plagiarism Checker and the system R. The experimental results confirmed both hypotheses showing that using preprocessing has a positive effect on the quality of plagiarism detection and that some techniques gave better results than others. The most interesting result of this research is that by using preprocessing techniques textual versions of the tools outperformed in some cases the specialized version of the tool developed specifically for source-code similarity detection.

Keywords: plagiarsim detection; high education; source-code; programming assignements; preprocessing tehniques; comparison; program similarity

PROŠIRENI SAŽETAK

Velik broj ljudi upisuje se svake godine na razna sveučilišta diljem svijeta što ukazuje da ljudi prepoznaju važnost edukacije, a istraživanja poput „Income and happiness: Towards a unified theory" [42] potvrđuju da su educirani ljudi u prosjeku i sretniji. Glavni razlog upisa na sveučilište jest ili bi trebao biti stjecanje znanja koja se mogu iskoristiti kasnije u životu. Iako je stjecanje znanja najvažniji dio studiranja događa se da studenti pokušavaju varati na ispitima, u zadaćama i sl. kako bi dobili potvrdu za znanje koje nisu stekli, zato nastavnici traže načine za sprečavanje varanja, odnosno načine pronalaženja studenata koji su varali. Postoje razni oblici varanja, a posebna vrsta jest *plagiranje*. Pojam *plagiranje* definira se kao [43] „čin preuzimanja pisanog dijela drugih osoba i predstavljanje istog kao vlastitog".

Fokus ovog rada jest otkrivanje plagijata, preciznije otkrivanje plagijata u programskim zadacima odnosno otkrivanje plagijata izvornog koda. Motivacija za ovo istraživanje proizlazi iz činjenice da je teško otkriti plagijate u velikim grupama sa 100 ili više studenata. Također, studenti koriste različite metode prikrivanja (eng. *obfuscation methods*) kako bi sakrili plagiranje što otežava otkrivanje. Takve metode variraju u svojoj složenosti (prema Faidhi i Robinson [44]) i mogu se podijeliti u dvije grupe [38, 85]: strukturne i leksičke. Osim navedenog, sam autor je nastavnik na predmetima koji se bave podučavanjem programiranja te je zato primarni interes otkrivanje plagijata u programskim zadacima, a ne u nekim drugim oblicima zadataka poput eseja i sl.

Kako bi varali studenti prilikom izrade programskih zadataka „kopiraju dijelove izvornog koda ili cijeli kod od svojih kolega ili pronalaze sličan izvorni kod na Internetu i koriste ga bez navođenja izvora." [85] Kako bi otkrili plagijate na velikom broju predanih rješenja koriste se razni alati za automatsko otkrivanje plagijata (u daljnjem tekstu *alati*). Ottenstein se već 1976. godine bavio tim problemom u svom članku [146] „Algoritamski pristup otkrivanju i sprečavanju plagijata".

Potrebno je napomenuti da je fraza *otkrivanje plagijata* neprecizna [132] jer opisuje primarnu primjenu alata, a ne stvarnu mogućnost alata koja je *otkrivanje sličnosti*. Alati koji danas postoje ne mogu se koristiti za izricanje optužbi za plagiranje [139] jer nijedan alat ne može razlikovati stvarni plagijat od sličnosti koja je nastala slučajno. Alati vrše automatsko pronalaženje sličnosti u rješenjima zadataka i predlažu potencijalne plagirane parove, a nastavnik je zadužen da donese finalnu odluku o tome radi li se o stvarnom plagijatu ili ne.

U kontekstu izvornog programskog koda Parker i Hamblen definiraju *plagirani program* kao [147] „... program koji je nastao iz nekog drugog programa s malim brojem rutinskih transformacija. Rutinske transformacije obično su tekstualne promjene koje ne zahtijevaju detaljno razumijevanje samog programa. Nažalost, plagiranje u programskim zadacima je olakšano studentima u predmetima s velikim brojem studenata te pojavom osobnih računala,

računalnih mreža i uređivača teksta jednostavnih za korištenje."

Kako ne postoji jedinstveni dogovor o tome što se smatra plagijatom, u ovom istraživanju definicija plagiranja u programskim zadacima je sljedeća:

> *Plagiranje u programskim zadacima je čin preuzimanja značajnog dijela izvornog koda (pa i cijelog izvornog koda) od drugih studenata ili s Interneta i korištenje istog bez navođenja izvora. Pod 'značajnim dijelom' smatra se da je sličnost između dva rješenja programskog zadatka dovoljno velika da ekspert (nastavnik, etičko povjerenstvo i sl.) određeni studentski rad smatra dovoljno „stvarnim" plagijatom te optuži studenta za plagiranje.*

Kao što je već rečeno postoje razni alati za otkrivanje sličnosti. Prema članku „Source-code Similarity Detection and Detection Tools Used in Academia: A Systematic Review" [140], samo iz domene otkrivanja plagijata izvornog koda postoji 120 članaka koji se bave razvojem novog alata. Nažalost većina nije dostupna iako su razvijeni nakon 2010. godine. Kako se alati razlikuju u mnogo elemenata prilikom odabira alata korisno je što postoje članci koji se bave pregledom i usporedbom svojstava alata [101, 116, 126, 139, 167] i/ili koji provode eksperimente s alatima [52, 63, 71, 157, 178, 183, 185]. Prema saznanju autora najveći eksperiment koji vrši statističku usporedbu alata proveli su Ragkhitwetsagul i ostali 2016. godine [157] gdje su testirali 30 alata.

Za ovo istraživanje odabrana su tri alata (JPlag [154], SIM [61] i Sherlock [85])[1] koja se nalaze u top 5 najboljih alata prema postojećim istraživanjima i koji imaju potrebne osobine kao što je podržani programski jezik (koristit će se samo alati koji podržavaju Java programski jezik), mogućnost korištenja izvan mreže (eng. *offline*) i sl. Mrežni (eng. *online*) alati su isključeni jer su studenska rješenja obično povjerljiva i ne smiju se dijeliti. Detaljan opis postupka odabira alata dan je u poglavlju 4. Kako svaki od alata ima dva načina rada, tekstualni (označavan sa sufiksom -text) i specijalizirani za detekciju izvornog koda (označavan sa sufiksom -java), oni su testirani u oba načina i promatrani su kao zaseban alat. Zbog toga se može reći da je u eksperimentu korišteno šest alata (svaki alat u dva načina rada).

Različiti alati koriste različite pristupe i implementiraju različite algoritme za izračun sličnosti između dokumenata. U početku su znanstvenici poput Ottensteina [146], Donaldsona i ostalih [41] koristili Halstead metrike [65] za otkrivanje plagijata, no ubrzo se taj pristup otkrivanja plagijata počeo smatrati slabim. Drugim riječima, točnost otkrivanja (dalje u tekstu *točnost*) nije bila dobra. U idealnoj situaciji alat bi imao savršenu točnost kada ne bi bilo lažno pozitivnih parova (eng. *false positives*) i kada bi bili pronađeni svi plagirani parovi. Znanstvenici su od tada pokušali unaprijediti točnost otkrivanja plagijata koristeći nove pristupe [31, 143,

[1] Napomena, postoji SIM alat od autora Githcel i Tran iz 1999. godine [59] (koji se u ovom radu označava kao SIM-GT) i SIM čiji je autor Grune iz 1989. godine [61] (koji se u ovom radu označava SIM), također postoji Sherlock sa Sveučilišta Warwick (koji se u ovom radu označava Sherlock) [85] i Sherlock sa Sveučilišta Sydney (koji se u ovom radu označava Sherlock-Sydney) [176].

201], razvijajući nove tehnike za izračun sličnosti [151, 155, 170], kreirajući nove tehnike za otkrivanje plagijata [81, 125, 165], poboljšavajući prezentaciju rezultata [56, 123, 127] i sl.

Jedan način da se poboljša točnost otkrivanja plagijata jest primjena tehnika predobrade [6, 38, 95]. Neki autori [38] definiraju predobradu kao prvi korak u procesu otkrivanja plagijata, što je ujedno i fokus ovog istraživanja. U ovom istraživanju pod tehnikom predobrade smatra se *tehnika koja rezultira promjenom izvornog koda i sl., a koja se može koristiti u različitim alatima za otkrivanje plagijata*, dakle bez ograničenja na samo jedan alat. Za predobradu izvornog koda mogu se koristiti različite tehnike predobrade [6, 30, 38, 95], uključujući: izbacivanja praznih znakova i komentara, izbacivanje uvoznih (eng. *import*) i uobičajenih deklaracijskih naredbi, izbacivanje pristupnih naredbi (eng. *access specifiers*), promjena tipova podatka u sinonime, promjena sinonima s generičkim konstruktima, promjena redoslijeda izvornog koda, izbacivanje predložaka, izbacivanje izvornog koda dobivenog od nastavnika i sl.

Odabir tehnika za ovo istraživanje baziran je na dostupnosti trenutno relevantnih tehnika prema znanstvenoj literaturi. Pošto nije dostupna direktna usporedba tehnika, odabrane tehnike možda nisu najbolje u području otkirvanja plagijata. Odabrane tehnike su: uklanjanje komentara (eng. *Remove Comments* - RC) — brisanje svih komentara iz izvornog koda; uklanjanje predloška (eng. *Template Exclusion* - TE) — brisanje koda koji je nastavnik dao studentima na korištenje; uklanjanje uobičajenog koda (eng. *Common Code Remove* - CCR) — brisanje koda koji je toliko učestali da ne može pomoći pri donošenju ikakvih odluka (npr. jednostavni konstruktor, anotacije i sl.). Dodatno korištene su i dvije kombinacije tehnika. Prva kombinacija je ona gdje se sve tri tehnike koriste zajedno (eng. *All techniques without Normalisation* - AllnoNOR), i druga kombinacija (eng. *All techniques with Normalisation* - AllNOR) kojoj se dodaje još tehnika normalizacije koda (eng. *Normalisation* - NOR) koja se koristi u alatu Sherlock. Detaljan opis tehnika dan je u poglavlju 7.

Kako bi se mogla evaluirati i usporediti točnost otkrivanja, potrebne su neke metrike. Mozgovoy [131] je opisao dva principa evaluacije. Prvi je izračun preciznosti (eng. *precision*) i odaziva (eng. *recall*), kao što je koristio Whale [185]. Drugi je izračun najvećeg lažnog rezultata (eng. *highest false match* – HFM) i razmaka (eng. *separation*), što je razlika između najmanjeg ispravnog rezultata (eng. *lowest correct result*) i HFM-a, kao što su koristili Hoad and Zobel [74]. Mjere koje će se koristiti u ovom istraživanju su preciznost, odaziv i F-beta (koja se izračunava na temelju preciznosti i odaziva), pošto su one korištene u raznim postojećim istraživanjima [31, 134, 154, 157, 170]. Detaljan opis metrika nalazi se u poglavlju 5. Potrebno je napomenuti da su rezultati različitih istraživanja [139] „usporedivi do neke mjere, ali potrebno ih je uspoređivati s oprezom jer su istraživanja provedena nad različitim skupovima podataka."

U ovom istraživanju koriste se stvarni podaci studentskih rješenja i podaci otvorenog skupa SOCO (SOurce COde Reuse) skupa podataka opisanog u članku „On the Detection of SOurce COde Re-use" [52], s PAN@FIRE2014 natjecanja. SOCO skup je podijeljen na *kolekciju za treniranje* i *kolekciju za testiranje*. Za obje kolekcije poznat je broj stvarnih plagijata, no postoji razlika u načinu provjere stvarnih plagijata. Kolekcija za treniranje provjerila su ručno tri

eksperta, dok su kolekciju za testiranje provjerili sami alati gdje se "*par mora pojaviti u barem 66% testiranih alata kako bi se smatrao relevantnim*" [52]. Kolekcija za treniranje kreirana je na skupu prezentiranog u članku „Plagiarism detection across programming languages" [5], a sastoji se od "*zadataka studenata koji sadrže primjene ponovo korištenog koda koji je profesor ručno pronašao i prijavio.*" [51]. Kolekcija za testiranje kreirana je iz Google Code Jam natjecanja (ista kolekcija korištena je u članku „Uncovering source code reuse in large-scale academic environments" [50]), a podijeljena je na šest zadataka [52] različitih scenarija i težina s različitim odnosom broja plagiranih i neplagiranih datoteka. Iako se u ovom istraživanju koristi svih 7 zadataka (1 iz kolekcije za treniranje i 6 iz kolekcije za testiranje) statistički su analizirana četiri zadatka iz kolekcije za testiranje.

Stvarni podaci studentskih rješenja (eng. *Real Student Solution* - RSS) koji su korišteni preuzeti su iz jednog predmeta na sveučilišnom diplomskom studiju. Predmet koristi Java-u kao glavni programski jezik. U eksperimentu se koriste dva zadatka iz predmeta od kojih se jedan analizira statistički. Predana rješenja mogu osim datoteka Java izvornog koda sadržavati i druge tipove datoteka poput slika, tekstualnih datoteka, XML datoteka, PDF datoteka i SQL datoteka. Također su mogući razni dodatni jezici u predanim rješenjima kao što su: HTML, CSS, JavaScript i sl. Svi zadaci su veliki, s jednom ili više datoteka, a sadrže od 400 linija izvornog koda (eng. *lines of source-code* – LOC) pa do 4000 linija koda, a iznimno i više. Prosječno je 51 predano studentsko rješenje po zadatku. U ostatku dokumenta pod jednim skupom rješenja (ili samo skupom podataka) smatra se jedan zadatak iz predmeta iz jedne akademske godine (dalje u tekstu *godine*). Znanstvenici koji se bave područjem otkrivanja plagijata izvornog koda u akademskoj sredini uglavnom rade s uvodnim predmetima iz programiranja i s rješenjima koja imaju u prosjeku 300 linija koda. Ovo istraživanje bavi se datotekama izvornog koda koje u prosjeku imaju 1800 linija koda. Muddu i ostali [134] koriste datoteke izvornog koda koje imaju veći broj linija koda, no one ne potječu iz *akademskog* okruženja. Detaljniji opis skupova podataka moguće je pročitati u poglavlju 6.

Kao što je već spomenuto fokus ovog istraživanja su tehnike predobrade izvornog koda. Kako postoje različite tehnike predobrade, opravdano je pitanje utječu li one stvarno na točnost alata. Pregled literature ukazuje na to da tehnike predobrade utječu na točnost (kao primjer pogledati [6]). Iako se pretpostavlja da postoji razlika u točnosti, potrebno je razmotriti jesu li sve tehnike predobrade jednako dobre ili postoje razlike između njih i ako da, koliko su te razlike velike. Osim toga, još uvijek nije utvrđeno daju li te tehnike uvijek identične rezultate ili njihov utjecaj ovisi o programskom jeziku ili tipu zadatka ili o nečem drugom. Također, postavlja se pitanje kada je i zašto neka tehnika bolja od neke druge. Prema saznanju autora, do sad nije proveden nijedan eksperimentalni dizajn koji testira tehnike predobrade pa sva ta otvorena pitanja i problemi čine motivaciju za ovim istraživanjem.

Sveobuhvatni cilj ovog istraživanja je *identificirati utjecaj tehnika predobrade (eng. PreProcessing Technique - PPT) na točnost otkrivanja plagijata u studentskim zadacima iz programiranja*. Sveobuhvatni cilj podijeljen je na dva manja cilja:

- C1: *Ispitati ima li primjena PPT-a utjecaj na točnost otkrivanja plagijata.*
- C2: *Ispitati kako, kada i zašto različite PPT-e utječu na točnost otkrivanja plagijata.*

Iz ciljeva proizlaze sljedeće hipoteze:

- H1: *Za testirane alate i zadatke postoji razlika u točnosti otkrivanja plagijata između barem jedne PPT-e i kad nijedna PPT-a nije korištena.*
- H2: *Za testirane alate i zadatke postoji razlika u točnosti otkrivanja plagijata između barem dvije različite PPT-e.*

Kako bi se cilj C2 trebao djelomično ostvariti hipotezom H2, ostatak će biti pokriven sa sljedećim istraživačkim pitanjem:

- IP1: *Kako različite PPT-e utječu na točnost otkrivanja plagijata pomoću testiranih alata i zadataka?*

Korištena metodologija (više detalja u poglavlju 3) sastoji se od dva dijela: kvantitativne analize i kvalitativne analize. Za kvantitativnu analizu koristi se eksperimentalni dizajn [46, 124]. Kako bi se obavile usporedbe, ostvario prvi cilj (C1) i testirala prva hipoteza (H1), korištena je višefaktorska analize varijance [46, 124] (ANOVA) uz korištenje *bootstrap* metode [46, 87] zbog problema vezanih za ispunjenje pretpostavaka za testiranje korištenjem višefaktorske ANOVE. Varijable u analizi za RSS su: alat za otkrivanje plagijata (dalje u tekstu *alat*), tehnika predobrade (dalje u tekstu *tehnika*) i *točnost* (F-beta – izračunata na temelju preciznosti i odaziva) koja je ujedno i zavisna varijabla. Različite godine koriste se kao replikacije kod RSS skupa. Iste godine koriste se za svaku kombinaciju tehnike i alata.

Kako bi se testirala hipoteza H2 i djelomično odgovorilo na istraživačko pitanje IP1, ANOVA nije dovoljna jer testira samo cjelokupni utjecaj i ne daje specifične informacije o utjecaju pojedinih grupa. Kako bi se to riješilo, korištena je analiza kontrasta i metoda jednostavnih efekata (eng. *simple effect analysis*).

Kao kvalitativna analiza napravljena je varijanta analize sadržaja [166] kako bi se ostvario cilj C2 i dao kompletan odgovor na istraživačko pitanje IP1. Pronađeni parovi su ručno analizirani s ciljem otkrivanja uzoraka koji ukazuju na to zašto je neki par bio ili nije bio pronađen. To će biti izvedeno za interesantne kombinacije alata, zadatka i tehnike koje je dala statistička analiza.

Za potrebe ovog istraživanja dizajniran je i djelomično izrađen alat za višestruku provjeru plagijata (eng. *multiple plagiarsim checker* – MPC). Točnost je izračunata pomoću MPC alata i bazirana je na prikupljenim podacima. Svi statistički izračuni su automatizirani pomoću sustava R na temelju izlaza iz MPC alata (više u poglavlju 3.4). Kako bi se alati za otkrivanje plagijata postavili u jednaku poziciju, oni su kalibrirani (postavljanje njihovih parametara) korištenjem metode opisane u [138], a svi detalji kalibracije dani su u poglavlju 4.3. Kako se F-beta koristi kao mjera za točnost potrebno je izračunati preciznost i odaziv za koje je potrebno poznavati

točan broj stvarnih plagiranih parova, no kod RSS-a to nije moguće znati sa stopostotnom sigurnošću. Zbog toga je broj stvarnih plagiranih slučajeva zapravo procjena (slično kao u [178]) kao unija plagiranih slučajeva pronađenih pomoću svakog korištenog alata. Pronađene parove provjerio je ekspert (odgovorni nastavnik na kolegiju) koji je označio stvarne plagijate. Točna procedura za pronalazak plagiranih parova prikazana je u obliku dijagrama toka i opisana u poglavlju 6.3.1.

Analizom rezultata zadataka dobivenih iz SOCO skupa i zadatka iz RSS skupa potvrđena je hipoteza H1 i H2, što znači da tehnike predobrade imaju utjecaj na točnost detekcije plagijata i da različite tehnike imaju različite učinke. Podaci su prvo analizirani statistički nad četiri različita zadatka SOCO skupa. Tablica 8.17 daje pregled svih statistički značajnih rezultata za ta četiri zadatka. Plus i minus simbol označavaju smjer efekta, tj. pokazuje imaju li tehnike pozitivan ili negativan utjecaj. Sve usporedbe i nazivi rađeni su tako da se očekuje pozitivan utjecaj, na primjer kod usporedbe bez tehnika predobrade i s tehnikama predobrade očekuje se da usporedba s tehnikama predobrade daje pozitivan efekt. Ono što je vidljivo iz rezultata SOCO skupa je da su rezultati svih četiriju zadataka uglavnom usklađeni. Ti rezultati dodatno su potvrđeni sa ostalim zadacima iz SOCO skupa.

Može se reći da na temelju SOCO skupa efekt PPT-a nije značajan za SIM-java alat kod svih zadatka, dok su za ostale alate neki rezultati bili značajni. Za JPlag-java i Sherlock-text efekt je bio značajan no nažalost negativan, što znači da su se korištenjem tehnike predobrade dobili lošiji rezultati nego bez korištenja tehnika predobrade. S druge strane za alate Sherlock-java, SIM-text i JPlag-text dobiven je pozitivan efekt, što znači da su se rezultati značajno poboljšali korištenjem tehnika predobrade. Odgovor na pitanje o razlogu tog negativnog učinka kod nekih alata leži u preciznosti i odazivu. Iz preciznosti i odaziva se vidi da je primjerice za JPlag-java odaziv konstantan, što znači da se korištenjem tehnika predobrade uključuje sve više parova (tj. sve više parova se smatra plagijatom), a već je na početku većina stvarnih plagiranih parova bila uključena. Može se reći da preciznost brže pada nego odaziv raste ili drugim riječima sve je više lažno pozitivnih parova, dok broj lažno negativih ostaje isti. S druge strane, za alate koji imaju pozitivan efekt dešava se upravo suprotno: preciznost raste, dok odaziv ostaje uglavnom konstantan. Nadalje, tu se javlja pitanje o tome zašto tehnike kod nekih alata spuštaju, a kod nekih dižu preciznost. Odgovor na to pitanje je trenutno izvan fokusa ovog rada te je to tema nekih budućih istraživanja no sumnja je da odgovor leži u različitim algoritmima koje alati koriste.

Na pitanje o tome zašto različite tehnike različito utječu na točnost može se ugrubo odgovoriti da tehnike procesiraju (npr. brišu, mijenjaju i sl.) različite elemente koda. Na primjer RC tehnika miče komentare, dok CCR tehnika ne miče komentare, ali miče neke druge elemente te nije čudno da su rezultati različiti. Naravno ovisno o alatu postavlja se pitanje o tome koja će imati bolji učinak, pa ako alat već u svom algoritmu ne uzima u obzir komentare kod usporedbe, tada naravno RC tehnika neće imati nikakav učinak. Ono što je zanimljivo jest da kompleksnost programa izgleda nema nekakvi značajan utjecaj na krajnje rezultate.

Kod RSS skupa podataka dobiveni su slični rezultati, a najbolji rezultati dobiveni su za SIM-text i JPlag-text kao i kod SOCO skupa. Generalno, može se reći da na temelju rezultata zadataka iz oba skupa tehnike predobrade uglavnom imaju pozitivan utjecaj, kao u slučaju alata JPlag-text, SIM-text i Sherlock-java, ali mogu imati i negativan učinak kao što je to slučaj kod Sherlock-text alata. Naravno u nekim situacijama nije bilo učinka koji bi bio dovoljno jak da bude značajan kao što je to slučaj kod SIM-java i JPlag-java alata. Prema rezultatima najbolji učinak kod alata SIM-text i JPlag-text postignut je korištenjem tehnike AllnoNOR, dok je kod Sherlock-java alata najbolji učinak postignut AllNOR tehnikom. Važno je napomenuti da iako u nekim situacijama nije bilo statistički značajnog učinka, ne znači da ga nema, a jedan razlog je taj da su primjerice kod usporedbi bez tehnika i s tehnikama sve tehnike stavljene u jednu grupu, što znači da nije svaka tehnika zasebno uspoređivana s rezultatima kada se tehnike ne primjenjuju.

Kako statistička analiza ne uzima u obzir pojedine parove, rezultati jednog zadatka RSS skupa analizirani su kvalitativno na razini parova. Ono što je uočeno jest da su kod svih alata nekorištenjem ijedne tehnike neki parovi imali 90% sličnosti, no primjenom tehnike TE ona je pala na manje od 30%, a primjenom tehnike AllNOR ili AllnoNOR na 0%. Razlog toga je činjenica da je u tim zadacima bilo puno koda koji je studentima na korištenje dao nastavnik, a sve dok se on nije maknuo, sličnost je bila jako visoka. Kroz više godina pronađen je isti uzorak u situacijama s puno nastavničkog koda, što ne znači da tehnike ne mogu imati pozitivan utjecaj unatoč nedostatku statistički značajnog rezultata. U ovom konkretnom primjeru 19 parova se maknulo, što znači da je nastavniku uštađeno puno vremena. Korist nije međutim samo u vremenu, već i u tome da takvi parovi onda prikrivaju stvarne plagijate, pa je tako u jednom primjeru jedan par stvarnog plagijata koji je imao 72% sličnosti kada se ne koriste nikakve tehnike rangiran na 392. mjesto, no kad se primijenila tehnika AllnoNOR, on je pozicioniran na prvo mjesto.

Najzanimljiviji rezultat ovog istraživanja jest da je učinak bio najbolji kod tekstualnih alata, štoviše upotrebom tehnika u nekim situacijama točnost tekstualne varijante alata je nadmašila specijaliziranu varijantu. Zbog toga se postavlja pitanje o mogućnosti korištenja isključivo tehnika predobrade umjesto specijaliziranih alata. Naravno, odgovor na to pitanje je također tema budućih istraživanja. Detaljniji raspis analize rezultata nalazi se u poglavlju 8.

Ukratko u sklopu ovog istraživanja dani su sljedeći doprinosi:

- Detaljan pregled područja *otkrivanja plagijata izvornog koda u akademiji*. U kombinaciji s rezultatima prezentiranim u članku „Source-code Similarity Detection and Detection Tools Used in Academia: A Systematic Review" [140] ovo istraživanje daje najveći i kompletan pregled literature tog područja od 1980 do 2018 godine.
- Dana je nova definicija plagijata u programskim zadacima.
- Kreirana je nova metoda kalibracije alata za otkrivanje plagijata. Zajedno sa rezultatima prezentiranim u članku „Calibration of source-code similarity detection tools for objec-

tive comparisons" [138] ovo istraživanje daje kompletan opis i demonstraciju korištenja kalibracijske metode i daje razloge njezine važnosti. Također ta metoda se može koristiti i u drugim istraživanjima poput *detekcije duplikata*, *otkrivanja autorstva* i sl.

- Kreirane su i evaluirane dvije nove tehnike predobrade: uklanjanje uobičajenog koda i uklanjanje predloška. Ovo istraživanje u kombinaciji s rezultatima članka „Process Model Improvement for Source Code Plagiarism Detection in Student Programming Assignments" [95] daje puni opis tehnike uklanjanja predloška. Također, kreirane i evaluirane su dvije nove kombinacije tehnika AllNOR i AllnoNOR.

- Identificirane su razlike i dato je objašnjenje razloga pojave razlika u točnosti otkrivanja plagijata pomoću testiranih alata i zadataka, i to u slučajevima kad se koriste tehnike predobrade i kad tehnike predobrade nisu korištene.

- Identificirane su razlike i dato je objašnjenje razloga pojave razlika u točnosti otkrivanja plagijata pomoću testiranih alata i zadataka kad su korištene različite tehnike predobrade.

- Utvrđeno je kako tehnike predobrade utječu na točnost otkrivanja plagijata testiranih alata u različitim studentskim zadacima. Ovo istraživanje prikazuje da se značajan napredak u točnosti otkrivanja plagijata može postići korištenjem tehnika predobrade nad nekim alatima i objašnjava to na primjeru otvorenog SOCO skupa i privatnog RSS skupa podataka.

- Napravljena je studija slučaja nad stvarnim studentskim radovima koja pokazuje prednosti korištenja tehnika predobrade (posebno tehnika uklanjanja predloška). Analiziranjem jednog zadatka tijekom šest godina, ovo istraživanje većih je razmjera nego većina dosadašnjih istraživanja koja se provode nad stvarnim studentskim radovima.

- Napravljena je analiza rezultata na velikim datotekama stvarnih studentskih rješenja, a pod velikim misli se na rješenja s 500-3900 linija koda.

- Napravljena je studija slučaja usporedbe alata bez korištenja tehnika predobrade na novom skupu stvarnih studentskih radova. Ovo istraživanje pokazuje interesantne rezultate gdje su tekstualne varijante alata bile bolje od specijalizirane varijante za Java programski jezik.

- Primijenjen je dizajn eksperimenta u području otkrivanja plagijata uz korištenje više zadataka i više alata kroz više godina.

- Kreirana je nova metoda za određivanje praga pozitivih i negativnih parova (median +- 2/3 IQR) bazirano na sličnosti između parova.

- Kreiran je novi okvir za provođenje eksperimentalnih usporedbi alata za otkrivanje plagijata koji koristi statističke metode kako bi se dobili objektivni rezultati. Iako okvir nije

formalno evaluiran on daje niz koraka koji se mogu pratiti u provođenju budućih istraživanja i daje opis za rješavanje problema normalnosti (*bootstrap* metoda) i slične probleme.

Osim znanstvenih doprinosa u ovom radu dani su i praktični doprinosi:

- Implementiran je MPC sustav koji se može koristiti za testiranje i uspoređivanje alata za otkrivanje plagijata. Sustav trenutno podržava alate SIM, JPlag i Sherlock, no moguće ga je proširiti da radi i s drugim alatima. Dodatno, sustav omogućava korištenje različitih tehnika predobrade i njihovo kombiniranje s alatima, a moguće ga je proširiti i s drugim tehnikama.
- Sustav MPC može se koristiti kao alat za detekciju plagijata koji korisniku omogućuje da bira alate koje želi, a može odabrati da želi provjeru sličnosti sa svim alatima istovremeno. Dodatno, razvijeno je grafičko sučelje i konzolna varijata kako bi se omogućilo što jednostavnije korištenje MPC sustava na dnevnoj bazi.
- Kalibracijski modul integriran je u sam MPC sustav koji omogućuje kalibriranje alata SIM, JPlag i Sherlock što omogućuje da se svi alati postave u jednake pozicije prije provođenja usporedbi.
- Dane su upute za korištenje određenih tehnika predobrade s određenim alatima (SIM, JPlag i Sherlock) za otkrivanje plagijata.
- Dane su R skripte koje su korištene za provođenje statističkih analiza kod usporedbe alata kako bi se mogle koristiti u budućim istraživanjima.
- Dan je kompletan izvorni kod ovog rada koji sadrži R skripte, Latex i Sweave datoteke i koji se može koristiti kao primjer za buduća istraživanja za generiranje grafova, tablica ili čak cijelog rada u pdf formatu pomoću R studio alata.

Iako je tema ovog istraživanja bila otkrivanje plagijata, važno je naglasiti da je otkrivanje plagijata samo liječenje simptoma, a pravi je problem činjenica da je uopće došlo do plagijata. Zbog toga je važno prije svega primjenjivati tehnike sprečavanja plagijata, a u slučaju da one ne uspiju dobro je poznavati i tehnike otkrivanja plagijata.

CONTENTS

List of Figures **XVI**

List of Tables **XVIII**

List of Abbreviations **XX**

1 Introduction **1**

2 Related work **5**
2.1 Definition of plagiarism . . . 6
2.2 Plagiarism prevention . . . 8
2.3 Differences between source-code and textual plagiarism . . . 10
2.4 Plagiarism detection . . . 11
2.5 Obfuscation methods . . . 12

3 Methodology **15**
3.1 Systematic literature review . . . 16
3.1.1 Top authors . . . 18
3.2 Research constraints . . . 19
3.2.1 Detection tools . . . 19
3.2.2 Preprocessing techniques . . . 20
3.2.3 Evaluation measures . . . 20
3.2.4 Datasets . . . 21
3.2.5 Ethical issues . . . 22
3.3 Plagiarism detection process . . . 22
3.4 Process automation . . . 24
3.4.1 Isabella cluster . . . 26
3.4.2 Automation of analysis phase . . . 27

4 Similarity detection tools **28**
4.1 Related work . . . 28
4.1.1 Related areas to source-code plagiarism detection . . . 29
4.1.2 Comparison of plagiarism detection tools . . . 31
4.2 Selection of tools . . . 33
4.2.1 Changes on selected similarity detection tools . . . 35
4.2.2 Problems with JPlag-java and Sherlock . . . 36
4.3 Configuration parameters of similarity detection tools . . . 38
4.4 Calibration of similarity detection tools . . . 39
4.4.1 Comparison of SIM and JPlag . . . 41
4.4.2 Calibration of SIM and JPlag . . . 44
4.4.3 Calibration of Sherlock . . . 48

5 Evaluation measures **54**
5.1 Related work . . . 54
5.1.1 Sensitivity . . . 55
5.1.2 Performance Index . . . 56
5.2 Precision, Recall and F-beta . . . 57

6 Experimental datasets **60**
6.1 Related work . . . 60
6.2 Source Code Reuse dataset . . . 62
6.3 Real student solutions datasets . . . 63
6.3.1 Procedure for analysing student solutions . . . 65

7 Preprocessing techniques **67**
7.1 Related work . . . 67
7.2 Selection of preprocessing techniques . . . 70
7.3 Remove comments technique . . . 72
7.4 Common code remove technique . . . 72
7.5 Template exclusion technique . . . 73
7.6 Technique selection test . . . 75

8 Result analysis **80**
8.1 Preparation for analysis . . . 80
8.1.1 Threshold level selection . . . 80
8.1.2 Planned comparisons . . . 83
8.2 SOCO dataset analysis . . . 85
8.2.1 SOCO dataset preparation for analysis . . . 85
8.2.2 Results for D1 assignments . . . 89
8.2.3 Results for D2 assignments . . . 98
8.2.4 Results for D3 assignments . . . 105
8.2.5 Results for D4 assignments . . . 112
8.2.6 Discussion . . . 119
8.2.7 Guidelines verification . . . 122
8.3 RSS dataset analysis . . . 124
8.3.1 Results for A1 assignments . . . 125
8.3.2 Discussion . . . 132
8.3.3 Limitation of statistical analysis . . . 134
8.4 Java or Textual version . . . 135
8.5 Contributions . . . 136

9 Future work **139**

10 Conclusion **140**

Appendix A Example of calibration report **144**

Appendix B SIM's Licence **148**

Appendix C MPC system architecture details **149**

Appendix D MPC system coverage report **157**

Appendix E Contrast codings **160**

Appendix F Contrast codings for the simple effecs analysis **161**

Appendix G Shapiro-Wilk normality test **168**

Appendix H Model comparisons **173**

Appendix I Constrast effect sizes **176**

Appendix J Interaction graphs **186**

Appendix K Precision and Recall for RSS dataset **196**

Appendix L List of used packages in R **199**

Bibliography **202**

Curriculum Vitae **221**
Published Research 221

LIST OF FIGURES

3.1 Number of articles in databases per year . . . 17
3.2 Citation rate per year . . . 18
3.3 High level architecture of MPC system . . . 25

6.1 Number of articles in databases per year . . . 61
6.2 Procedure for confirming plagiarised matches . . . 66

8.1 F1 score for SOCO T1 assignment . . . 82
8.2 F1 score for SOCO C2 assignment . . . 82
8.3 F1 score for SOCO D1 assignment with 3*IQR . . . 90
8.4 D1 assignments - residuals . . . 90
8.5 F1 mean comparison for SOCO D1 . . . 92
8.6 F1 score for SOCO D2 assignment with 3*IQR . . . 99
8.7 D2 assignments - residuals . . . 99
8.8 F1 mean comparison for SOCO D2 . . . 101
8.9 F1 score for SOCO D3 assignment with 3*IQR . . . 106
8.10 D3 assignments - residuals . . . 106
8.11 F1 mean comparison for SOCO D3 . . . 107
8.12 F1 score for SOCO D4 assignment with 3*IQR . . . 113
8.13 D4 assignments - residuals . . . 113
8.14 F1 mean comparison for SOCO D4 . . . 114
8.15 False positives for SOCO C1 assignment with 3*IQR . . . 124
8.16 F1 score for RSS A1 assignment with 3*IQR . . . 126
8.17 A1 assignments - residuals . . . 126
8.18 F1 mean comparison for RSS A1 . . . 128
8.19 False positives for RSS A1 assignment in 2015-2016 with 3*IQR . . . 133
8.20 False positives for RSS A1 assignment in 2016-2017 with 3*IQR . . . 134

C.1 Summary report main input interface . . . 149
C.2 Summary report table . . . 150
C.3 Match side by side comparison . . . 150
C.4 Summary report class diagram - Web GUI module . . . 151
C.5 Summary report class diagram - core module . . . 151
C.6 Phase creation class diagram . . . 153
C.7 Prepare tools creation class diagram . . . 154
C.8 Detection tools creation class diagram . . . 155

C.9 Preprocessing techniques creation class diagram 156

D.1 Coverage report summary 157

J.1 Interaction graphs for SOCO D1 - part 1 186
J.2 Interaction graphs for SOCO D1 - part 2 187
J.3 Interaction graphs for SOCO D2 - part 1 188
J.4 Interaction graphs for SOCO D2 - part 2 189
J.5 Interaction graphs for SOCO D3 - part 1 190
J.6 Interaction graphs for SOCO D3 - part 2 191
J.7 Interaction graphs for SOCO D4 - part 1 192
J.8 Interaction graphs for SOCO D4 - part 2 193
J.9 Interaction graphs for RSS A1 - part 1 194
J.10 Interaction graphs for RSS A1 - part 2 195

K.1 Precision and Recall for RSS A1 assigment in academic year 2012-2013 196
K.2 Precision and Recall for RSS A1 assigment in academic year 2013-2014 197
K.3 Precision and Recall for RSS A1 assigment in academic year 2014-2015 197
K.4 Precision and Recall for RSS A1 assigment in academic year 2017-2018 198

LIST OF TABLES

2.1 Similaritites between textual and source-coude detection metrics [100] 10

3.1 Authors with most articles in SLR and most citations 19

4.1 Overview of plagiarism detection tools 32
4.2 Tools calibrated configuration 41
4.3 JPlag-java and SIM-java calibration dataset similarities 42
4.4 JPlag false similarity examples in SOCO 8 case 43
4.5 SIM-java calibrated with JPlag-java as base tool [a] 45
4.6 JPlag-java calibrated with SIM-java as base tool [a] 46
4.7 SIM-text calibrated with JPlag-text and SIM-java as base tools 47
4.8 JPlag-text calibrated with SIM-text and SIM-java as base tools 48
4.9 Sherlock default parameter values 49
4.10 Sherlock-java's first calibration with SIM-java and JPlag-java 51
4.11 Sherlock-java's second calibration SIM-java and JPlag-java 52
4.12 Sherlock-text's calibration with SIM-text and JPlag-text 53

6.1 SOCO dataset structure 63
6.2 Real Student Solution dataset structure 64

7.1 Mentioned preprocessing techniques in reviewed articles 68
7.2 Configuration comparison based on removed lines of code for template exclusion technique 74
7.3 PPTest similarities for JPlag-java and SIM-java 76
7.4 PPTest similarities for JPlag-text and SIM-text 77
7.5 PPTest similarities for Sherlock 77
7.6 Similarities for Sherlock's favour test 78

8.1 Example of SOCO T1 similarities near threshold based on number of plagiarised matches 81
8.2 Planed comparisons for tool factor 83
8.3 Planed comparisons for technique factor 84
8.4 SOCO dataset structure for experiment 86
8.5 ANOVA results for SOCO D1 91
8.6 Contrasts results for SOCO D1 94
8.7 Simple effect analysis result for SOCO D1 95
8.8 ANOVA results for SOCO D2 100

8.9 Contrasts results for SOCO D2 . . . 102
8.10 Simple effects analysis result for SOCO D2 . . . 103
8.11 ANOVA results for SOCO D3 . . . 107
8.12 Contrasts results for SOCO D3 . . . 109
8.13 Simple effect analysis result for SOCO D3 . . . 110
8.14 ANOVA results for SOCO D4 . . . 113
8.15 Contrasts results for SOCO D4 . . . 116
8.16 Simple effects analysis result for SOCO D4 . . . 117
8.17 Summary of SOCO assignements significant comparisons . . . 120
8.18 ANOVA results for RSS A1 . . . 127
8.19 Contrasts results for RSS A1 . . . 129
8.20 Simple effects analysis result for RSS A1 . . . 130

E.1 Tool contrast codings . . . 160
E.2 SOCO technique contrast codings . . . 160
E.3 Student technique contrast codings . . . 160

F.1 Simple effects analysis contrast codings - SOCO dataset - part1 . . . 162
F.2 Simple effects analysis contrast codings - SOCO dataset - part2 . . . 163
F.3 Simple effects analysis contrast codings - SOCO dataset - part3 . . . 164
F.4 Simple effects analysis contrast codings - student dataset - part1 . . . 165
F.5 Simple effects analysis contrast codings - student dataset - part2 . . . 166
F.6 Simple effects analysis contrast codings - student dataset - part3 . . . 167

H.1 MLM comparison for SOCO D1 . . . 174
H.2 MLM comparison for SOCO D2 . . . 174
H.3 MLM comparison for SOCO D3 . . . 174
H.4 MLM comparison for SOCO D4 . . . 175
H.5 Multi level linear model (MLM) comparison for RSS A1 . . . 175

I.1 Contrasts effect sizes for SOCO D1 . . . 176
I.2 Simple effect analysis effect sizes for SOCO D1 . . . 177
I.3 Contrasts effect sizes for SOCO D2 . . . 178
I.4 Simple effect analysis effect sizes for SOCO D2 . . . 179
I.5 Contrasts effect sizes for SOCO D3 . . . 180
I.6 Simple effect analysis effect sizes for SOCO D3 . . . 181
I.7 Contrasts effect sizes for SOCO D4 . . . 182
I.8 Simple effect analysis effect sizes for SOCO D4 . . . 183
I.9 Contrasts effect sizes for RSS A1 . . . 184
I.10 Simple effect analysis effect sizes for RSS A1 . . . 185

LIST OF ABBREVIATIONS

AllnoNOR All techniques without Normalisation.

AllNOR All techniques with Normalisation.

ANOVA Analysis of variance.

CCR Common Code Remove.

CD Calibration Dataset.

CDS Calibration Difference Sum.

GUI Graphical User Interface.

IDE Integrated Development Environment.

IQR inter-quartiles.

JSF Java Server Faces.

LOC Lines of code.

MBJ Maximum Backward Jump.

MFJ Maximum Forward Jump.

MJD Maximum Jump Difference.

MLM Multi level linear model.

MPC Multiple Plagiarism Checker.

MRL Minimum Run Length to Store.

MSL Minimum String Length to Store.

NOR Normalisation.

PPT PreProcessing Technique.

RC Remove Comments.

RSS Real Student Solution.

RWS Remove white spaces.

SLR Systematic Literature Review.

SOCO SOurce COde Reuse.

STR Strictness.

TDD Test Driven Development.

TE Template Exclusion.

CHAPTER 1

INTRODUCTION

Education enables a happier life, and according to [42] educated people are, on average, happier. So it is no wonder that concepts such as life long learning have emerged. "*But education can be seen as being also the means of establishing a protective social environment in which emotional stability is possible.*" [148, p. 57] [70, p. 766]. Young people recognize the importance of education, and this is visible from a large number of students that enrol in universities all over the world. Higher education helps them to get (hopefully) a better job and ensure a good life. There are other factors, except for education, that account for a good life but these are out of the scope of this thesis (a useful discussion on how to have a meaningful life is available in [149]).

It is the responsibility of universities and other educational institutions to provide students with knowledge, skills and competencies which they can use later in life. Companies acquire good students very fast and they expect that a person with a certain diploma has certain knowledge. But the diploma, unfortunately, does not state what exact knowledge was acquired. It is hard (or maybe impossible) to specify the exact amount of knowledge that one student has after finishing a particular course or university program. To solve that, learning outcomes are defined, which enable comparison of acquired knowledge between students after finishing a similar course and/or university program. That this is important is evident from the existence of qualification frameworks such as: The Croatian Qualifications Framework[1] on a national level or the European Qualifications Framework[2] on the European Union level.

But there is a problem, even with the qualification frameworks in place. It is possible that a student deceives a teacher by doing something that they did not really do (commonly known as cheating), and therefore did not achieve the expected learning outcome(s). During their study students must submit assignments and pass exams to pass a course, and the teacher's role is to evaluate the assignments and/or exams and grade them. Also, teachers should be capable and ready to find students that are cheating. For example, if a student "*copies from someone else and the teacher does not found that out, the student ends the course with insufficient knowledge and can later in life have problems because of that. More worrisome, studies show that students not only plagiarize regularly but also believe that it is okay to do so.*" [171] Various sessions held on the topic of academic dishonesty like [152] discuss ways students can cheat in exams and assignments, and one special form of cheating is plagiarism. The term plagiarism is usually defined as [43]: "*the act of taking the writings of another person and passing them off as one's own*".

The problem of plagiarism is very well known and many studies have been conducted on

[1] `https://www.azvo.hr/en/enic-naric-office/the-croatian-qualifications-framework-croqf`

[2] `http://www.cedefop.europa.eu/cs/events-and-projects/projects/european-qualifications-framework-eqf`

how to deal with it, as presented in Chapter 2. There are two main approaches to deal with the problem: *plagiarism prevention* and *plagiarism detection*. Although the focus of this thesis is on plagiarism detection, one should always try to prevent plagiarism in the first place. Since prevention does not exclude detection, it is best to use both. In case prevention measures fail, detection techniques may indicate students who try to plagiarise. As the saying goes: "*Trust, but verify*"[3].

As already stated the focus of this thesis is plagiarism detection, more specifically source-code plagiarism detection in academia. The motivation for this comes from the fact that in large classrooms (with around 50 students or more) and various obfuscation methods it is hard to check and find plagiarized cases manually. In other words, it is hard to identify if some student has plagiarised his/her homework assignment. As a teacher, teaching programming courses, the author's interest is in detecting plagiarism in programming assignments (further referred to as *assignments*) which contain source-code, rather than any other kind of assignments, for example essays. To perform plagiarism detection on a large number of submitted assignments (further referred to as *submissions*) plagiarism detection engines (further referred to as *tools*) are used.

Note that the terms '*plagiarism detection engine*' or '*plagiarism detection system*' are imprecise [132] since they focus on the primary application rather than the real capabilities of the tool, which is similarity checking. Such "*simplification that is done literally by all plagiarism detection systems – reducing plagiarism to similarity – is usually not stated explicitly, though there is an obvious difference.*" [132] Tools that are available today are not able to distinguish between real plagiarism and coincidental similarity, and maybe never will. Because of this, it is important to remember [139] that tools only suggest potential plagiarism and that they can not be used to accuse somebody of plagiarism. It is up to the teacher to make the final decision as to whether it is a case of real plagiarism or not.

Plagiarism detection tools[4] usually rank pairs, consisting of two submissions (further referred to as *pairs* or *matches*), based on similarity (usually displayed in percentages 0% - 100%) and mark every pair with similarity above a predefined percentage (called *threshold*) as plagiarized. To be able to do that a detection tool needs to calculate the similarity first. Different tools are based on different approaches and implement different algorithms After that, the similarity between submissions. Differences in similarity calculations cause differences in rankings which leads to differences in indicated plagiarized matches and therefore a different detection accuracy (further referred to as *accuracy*). In an ideal situation, a tool would have perfect accuracy, meaning finding all plagiarized pairs and finding only plagiarized pairs.

Numerous researchers have attempted to improve the accuracy of plagiarism detection tools using new approaches [143], creating new methods for similarity calculation [170], building new techniques for plagiarism detection [134], and so on. One way of improving the accuracy is

[3] `https://www.leadergrow.com/articles/443-trust-but-verify`

[4] Although it is a convention in the literature to use the term *plagiarism detection tool* the term *similarity detection tool* is more accurate; in this research both terms are used as synonyms.

by using preprocessing techniques [6, 38, 95]. Some authors like to define preprocessing as the first step in the process of plagiarism detection [38]. In this research, a *PreProcessing Technique (PPT) is understood as a technique which results in modified source-code or similar that can be used with different detection tools instead of being limited to only one tool.*

This research focuses on the preprocessing step, more precisely on the effect of PPTs on the accuracy of plagiarism detection used in the preprocessing step. To preprocess source-code, various PPTs can be used, including [6, 30, 38, 95]: removal of white space and comments, removal of imports and common declaration statements, removal of access specifiers, changing data types to synonyms, changing synonym constructs with generic ones, source-code resort, exclusion of template code, exclusion of source-code given by teachers, etc.

Since there are various preprocessing techniques, it is worthwhile asking whether they really make any difference. The literature review seems to indicate that preprocessing techniques do make a difference (see [6, 96] for an example). But even if it is presumed that there is a difference, it has to be considered whether all preprocessing techniques are equally good, or whether some differences exist between them, and, if so, how large those differences are. Furthermore, it still needs to be established whether those techniques always yield identical results, or whether their effect depends on the programming language or the type of assignment, or something else. Also, the question arises of "*when*" or "*why*" some technique is better than another. These open problems and questions constitute the motivation for this research.

The overall goal of this research is to *identify the effect of PPTs on plagiarism detection accuracy in student programming assignments.* The overall goal is divided into two subgoals:

G1 : *Find out*[5] *whether the application of PPTs has an effect on plagiarism detection accuracy.*

G2 : *Find out how, when and why different PPTs affect plagiarism detection accuracy.*

From the goals the following hypotheses emerge:

H1 : *For tested tools and assignments there exists a difference in plagiarism detection accuracy between at least one PPT and when no PPT is used.*

H2 : *For tested tools and assignments there exists a difference in plagiarism detection accuracy between at least two different PPTs.*

Since G2 is partially fulfilled by H2 the rest is covered by the following research question:

Q1 : *How do different PPTs affect the plagiarism detection accuracy for tested tools and assignments?*

[5] In the original proposal the word 'Investigate' was used in the goals, which is replaced with 'Find out'. The reason is only to improve the language.

The rest of the thesis is structured as follows. In Chapter 2, related work is described including short sections describing plagiarism prevention, definitions of plagiarism, and obfuscation methods. Chapter 3 outlines the methodology and describes the used datasets, metrics, tools and techniques. A more detailed description of the metrics is given in Chapter 5, of the datasets in Chapter 6 and of the techniques in Chapter 7. To objectively compare the results from different tools calibration of the tools was performed and is outlined in Chapter 4. Chapter 8 describes and discusses the results of the main experiment, Chapter 9 presents future work and Chapter 10 concludes.

CHAPTER 2

RELATED WORK

Academic dishonesty is a substantial problem in high education which has been present for a long time, and one which can be present in any course. The earliest work dates back to 1904 [10]. Academic dishonesty "*typically includes acts of plagiarism, using concealed notes to cheat on tests, exchanging work with other students, buying essays or, in some extreme and notorious cases, asking others to sit examinations for you.*"[175]

With the increasing popularity of the Internet and advances in technology, this problem has been growing since it makes cheating easier [175]. That technology, like personal computers or computer networks, makes cheating easier, was already noticed in 1989 by Parker and Hamblen [147]. So, today there are various interesting ways how students can cheat. One categorization of cheating methods is LowTech cheating methods and HighTech cheating methods, as presented in [92]. LowTech methods are, for example, hiding notes inside a pencil or switching a test with another student. HighTech methods, on the other hand, are more sophisticated methods using technology, such as sending text messages during an exam using mobile phones or using small wireless earbuds with a microphone so another person can talk to the student during the exam.

A lot of responsibility lies on universities to ensure academic honesty [7] since students still have a lot to learn about what actions are appropriate in society. By taking a strong stand for honesty, universities can point students in the right direction while determining their values.

Two of the major issues in academic dishonesty are plagiarism and collusion [168], and definitions are presented in Section 2.1. What constitutes plagiarism / collusion is difficult to answer as different research studies have indicated [11, 34, 167, 168]. The issue with that is, as Lancaster stated in his PhD "*why students are expected to be able to understand what plagiarism is, if academics will not or cannot define it explicitly*" [100]. Also, as presented in [27, 86, 196], another problem is that students do not recognize some cases as plagiarism / collusion while the teacher does, for example, students working in teams exchange parts of the work with other teams, or students working on individual assignments collaborate and that results in similar solutions. A good overview of various research with students regarding plagiarism/collusion can be found in [167].

A further problem is that similarity can occur for various reasons. For example, in programming assignments, some similarity might come from the syntax dictated by the programming language or from reusing source-code [110], which is actively promoted in various programming languages. Another problem [167] with programming assignments is the missing standard that describes how to reference source-code parts which are not the authors'. In visual arts, for example, "*there are no clear guidelines to help distinguish between plagiarism on the one hand and homage, parody, visual referencing, and related practices on the other. Academics and*

students alike have difficulty knowing what is academically legitimate and what is not. Further, it is not possible to reference an external source in a way that makes the reference visible to all viewers of the work." [167] Because of all that it is important that teachers clearly define what is considered plagiarism / collusion in their course, otherwise students might not even realize that they have plagiarized / colluded.

2.1 Definition of plagiarism

Various definitions of plagiarism exist in the literature. In [140] an extensive review was performed analysing 150 papers dealing with source-code code plagiarism detection tools used in academia and one of the questions was [140]: "*What is meant by source-code plagiarism in academia?*". In [140] three categories of plagiarism are analysed: the general definition of plagiarism, the definition of source-code (program) plagiarism, and the definition of source-code plagiarism in academia.

The general definition of *plagiarism* is already given in the introduction cited from Encyclopaedia Britannica. In [140] the four definitions for the word plagiarism from Merriam Webster On-line Dictionary[1] are found as the most comprehensive. But the essence of all general definitions is *presenting someone else's work (which includes writing, ideas or similar) as one's own*.

More precise definitions of *source-code plagiarism* are more of interest to this research. According to [140] the most used definition is the one from Parker and Hamblen [147]: "*A plagiarized program can be defined as a program which has been produced from another program with a small number of routine transformations. Routine transformations, typically text substitutions, do not require a detailed understanding of the program.*" Other definitions are more or less variations of this definition.

The most valuable definitions found in the literature for this research are definitions focusing on *source-code plagiarism in academia*. According to [140] the most used and best suitable definition is the one from Cosma and Joy [34]: "*Source-code plagiarism in programming assignments can occur when a student reuses ... source-code authored by someone else and, intentionally or unintentionally, fails to acknowledge it adequately ..., thus submitting it as his/her own work. This involves obtaining ... the source-code, either with or without the permission of the original author, and reusing ... source-code produced as part of another assessment (in which academic credit was gained) without adequate acknowledgement .... The latter practice, self-plagiarism, may constitute another academic offense.*".

In [168] it is noted that some writers use the term *plagiarism* to mean both *plagiarism* and *collusion*. In this research this is also the case, but before the reason for that is stated, the difference between plagiarism and collusion needs to be clarified. "*Both plagiarism and collusion entail using the work of others without properly attributing that work.*" [168] The

[1] `https://www.merriam-webster.com/dictionary/plagiarize`

difference is that collusion can only occur when there is forbidden collaboration between two or more people.

Collusion happens when "*working together to produce assessed work in circumstances where this is forbidden.*"[11], on the other hand, plagiarism is "*representing another person's work as being your own, or the use of another person's work without acknowledgement.*" [11] In other words collusion addresses issues when work is taken from a person (or persons) known to the author and the other person is involved. Plagiarism occurs when the work is taken either from an author or from a public domain such as the web without the knowledge and involvement of the original author.

Some research [54] indicates that plagiarism is in general looked at as a bigger problem than collusion, but this can be open to debate, at least from a teacher's standpoint. Suppose students are given an assignment that they need to solve individually but two students cooperate and produce a solution together and one student plagiarises by taking the solution from some random student. Here is a brief discussion presenting four examples:

1. Suppose all of them are clever enough to modify the solution so they don't get caught. Why would the student who plagiarised be more of a problem than the two who collaborated, since all three have cheated and got a grade they did not deserve?

2. Suppose students get caught but they are honest and admit what they did. Should the student who plagiarised be punished more than the other two who colluded?

3. Here are three scenarios where the students get caught and do not admit what they did: scenario one – they claim that the similarity is all purely coincidental, scenario two – the student who plagiarised states he has collaborated with the innocent student, but the innocent student denies it, and scenario three – the students who colluded stated they don't know each other and that the other has plagiarised.

   The first and second scenario are different than the third since an innocent student is involved, but in all three scenarios the teacher has the same problem to determine who is lying and it might happen that they can't expose the liars. From the standpoint of the innocent student it is clear that plagiarism is worse than collusion, but from the teacher's standpoint he/she has the same problem of determining the liars in all three scenarios. In such scenarios is the problem plagiarism/collusion or is the lying the main problem?

4. Suppose that collusion is treated more lightly than plagiarism and then two students plagiarise from the web and they state they colluded and maybe even did. Is it okay that they get a softer punishment?

Analysing the stated questions is out of the scope of this research, but teachers should take their time to think about such things.

With the difference between plagiarism and conclusion in mind, it is clear that the definitions from Parker and Hamblen, and from Cosma and Joy, cover both collusion and plagiarism.

There are two reasons why also in this research plagiarism is used to mean both collusion and plagiarism. Firstly, since the focus of the research is on programming assignments, when a teacher finds a similarity between two submissions that are suspicious they do not know whether it is a case of collusion or plagiarism. It could be that those two have taken the same code part from the web without collaborating with each other. This would mean that it is a case of plagiarism but since the only comparison that was done was comparing one student submission to another it may look like collusion. The second reason, in the course on which this research is preliminarily based on, the punishment is the same regardless of the type of cheating (plagiarism or collusion). It makes no difference to the teacher if a student copies something from the web (plagiarism) or from another student (collusion). Note that copying from another student can be plagiarism or collusion. Which one it is depends whether the other student participated in the process (for example by giving their solution to the first student) or not. Since there is always the problem of lying as described above, only if one student admits plagiarism from another student the innocent student is not punished on the same level as the one who admitted plagiarising, but the student who plagiarised is punished the same as the once that colluded.

The definition from Cosma and Joy is good, but to be used in this research it is missing information about how it is decided if something is or is not plagiarism. As there is no unique agreement on what constitutes plagiarism, in this research plagiarism in programming assignments in academia is defined as follows. Note that the word 'real' is quoted, since there is always a small possibility, even with high similarity, that the similarity is coincidental.

> *Plagiarism, in programming assignments, is the act of taking a significant amount of source-code parts (up to the entire source-code) from other students or from the Internet and using it without noting its origin. A 'significant amount' means that the similarity between two solutions of a programming assignment is high enough that an expert (teacher, ethical board, etc.) considers specific student work as sufficiently 'real' plagiarism to accuse the student of plagiarism.*

2.2 Plagiarism prevention

To efficiently prevent plagiarism one should know why students cheat. There are various reasons why students plagiarise [27, 85, 171, 180]: time pressure, an uninteresting course, a poor attitude from the teacher, poor motivation, lack of knowledge, fear of failure, perceived irrelevance of assignments, inadequate resources, procrastination, underestimation of required time and effort, etc. In [180] it was found out that "*the most probable reason for plagiarising, from a student's point of view, is not being able to successfully complete an assignment.*"

Some of the reasons to plagiarise can relatively easy be reduced, or maybe even eliminated, by teachers and/or universities. For example, to reduce the lack of motivation and deal with an uninteresting course, a teacher can try to use gamification methods [93, 94] as one way of making the course more appealing to students. Maybe gamification will not work for everyone

and some may think it is childish, but if it helps some of the students it is worth trying.

Another problem like time pressure may be caused by the amount of work that a student needs to do in one term. To evaluate and control the amount of work that needs to be done is partially controlled through systems like the European Credit Transfer System (ECTS). In short, the idea of ECTS is to have each course assign ECTS points which reflect how much time does it should take an average student to complete the course. For example, if one course has 6 ECTS points it means that to complete the course a students needs to work approximately 180h (1 ECTS = 30 hours). But when using systems like ECTS, the teachers define how hard one course is and how many ECTS points it should have. From the student's point of view this might not be correct. Because of that, every few years the ECTS points are re-evaluated to make them more precise. But universities can use other ways "*for calculating course difficulty and producing appropriate learning strategies for students*" [142].

As can be seen from the above examples some cases of plagiarism can be prevented relatively easy. There are other methods that can be tried to prevent plagiarism that has been reported in the literature: generating a unique assignment for each student, and making students evaluate each other's work [195], creating new assignments every year to prevent plagiarism across generations [171], requesting students to make a statement (like a formal cover sheet) that the work is their own [72], requesting incremental submissions [177], reducing class sizes [88], using assignments where students need to reuse already written code in a controlled environment [161], tracking students' input while developing assignments with systems like the APE (anti plagiarism editor) or Gorilla software [177], offering many tutoring or help sessions [88], introducing a mandatory ethics course [88], developing dishonesty policies [132], and similar methods.

For some of the mentioned prevention methods there is no assurance how effective they are and if they will work at all in a specific situation, so additional research is required. In addition to the above mentioned methods, educating students and teachers about plagiarism and other related elements [171], like authorship, intellectual rights, proper referencing and similar, is a crucial factor.

As already stated in Section 2.1, students can cheat by plagiarising or colluding, but it is also possible that a student hires a professional programmer to do his/her assignment. This kind of cheating is not usually possible to detect by any tool, but it can be found out by questioning the student about their source-code. From personal experience, usually a student that did not write the program hirself/herself will have problems answering questions about the source code they wrote, even the most trivial matters like where some variable is initialized or questions where the answer is already on the screen. This is a good example why plagiarism prevention should be used together with plagiarism detection.

Results from [4] "*suggest that preventing may be more effective than detecting at dealing with the problem of plagiarism. Although detection is an important issue, the use of carefully designed teaching methodologies and assessment strategies may make it unnecessary. In particular, assessment techniques that indirectly reward individual contributions and naturally*

avoid plagiarism constitute an interesting alternative to others that simply control and punish plagiarism."

Because prevention can eliminate plagiarism and make detection unnecessary, and because in some cases the detection is not possible, prevention should always be used together with detection when dealing with plagiarism. As indicated in [89] sometimes just the fact that the students are told that plagiarism detection is performed already prevents some plagiarism. In some sense, plagiarism detection is then used as a prevention method.

2.3 Differences between source-code and textual plagiarism

Many things can be plagiarised: music, visual art, source-code, literature text, etc. Each plagiarism type has its own problems and different methods are used to prevent and detect it. Plagiarism analysed in this research is source-code plagiarism and since source-code is basically text there is a logical question: *can the same principles and methods be applied to source-code plagiarism as for textual plagiarism?* To answer that question one needs to look at similarities and differences between source-code and text.

Similarities between text and source regarding metrics that can be used to detect similarity in assignments written in source-code vs. assignments written in text according to [100] are presented in Table 2.1.

Table 2.1: Similaritites between textual and source-coude detection metrics [100]

Source-code	**Text**
the number of lines that contain comments	the number of nouns used in the free text submission
the proportion of 'while' loops to 'for' loops	the proportion of uses of 'their' compared with 'there'
the number of compilation errors when the code is compiled	the proportion of words that are not found in a dictionary
the number of keywords that occur in both source code submissions	the number of capitalised words that occur in both free text submissions
the number of functions in the first submission that can be paired with a function with the same number of lines in the second	the number of paragraphs in the first submission that can be paired with a paragraph with the same number of sentences in the second

Regardless of the similarities, some studies indicate that [30] there are clear differences between textual plagiarism and source-code plagiarism and that academic integrity issues [168] differ between text based and computing assessments. "*Probably the most important difference is that source-code is more structured than natural language, which makes it more likely that two lines are similar.*"[139]. Some other differences in the nature and perception of source-code and textual plagiarism are:

- In source-code there is no standard for referencing [139, 167];

- Discussing details of work in progress is found more problematic in source code [1, 168];
- "*Plagiarism and collusion are more difficult to define in computing and the boundaries between acceptable and unacceptable practices are more blurred than with text.*" [168];
- It is less acceptable to copy part of a textual assignment like an essay than it is in a programming assignment. [1];
- Writing computer programs is more difficult than writing text [167] and the reason for that is that, even if someone is bad at writing, they can produce something that makes sense, on the other hand, if you are a bad programmer it is much more difficult to write a program that works.

Because of the differences, specialized tools have been built to detect source-code plagiarism. This does not mean that detection tools built for textual plagiarism can not be used, especially with the mentioned similarities in possible metrics, but it is expected that a specialized source-code plagiarism detection tool is better to detect source-code plagiarism. This is supported by [157] where "*experimental results show that highly specialised source code similarity detection techniques and tools can perform better than more general, textual similarity measures.*" Because of this, the research focus here is on those specialized source-code plagiarism detection tools, but since there are similarities between source-code and text in the experiment the textual version of the chosen tools are also compared and discussed.

2.4 Plagiarism detection

When plagiarism prevention methods fail, plagiarism detection methods can be helpful to find those students who have plagiarised. Detecting plagiarism is not an easy task, especially when the number of students in a classroom goes above 100. Also, students use different obfuscation methods to hide plagiarism, which makes detection more difficult (more details in Section 2.5). There are two approaches to detecting plagiarism: manually or with the use of similarity detection tools.

One approach to detect plagiarism manually is presented in [128], where it was "*found that plagiarized groups score higher in programming and spend less time in developing programs compare to non-plagiarized group.*" This indicates that examining times spent on program development can help detect plagiarism. For homework assignments, systems like Git or SVN can maybe do the same trick.

Sometimes plagiarism can be detected manually just by monitoring a student's development across time [182]. Any rapid improvements in submitted assignments may be an indication that plagiarism might be taking place. The teacher can question such a student about the assignment and find out if this is really the case.

Few prevention methods, like tracking students' input while developing assignments with

systems like APE/Gorilla software [177], can also be used to detect plagiarism. This method can be combined with the method that suggests examining development times.

Students sometimes plagiarise with the help of Google [174], they search for the solution and then they copy-paste the found solution or part of the solution into their work. Teachers, on the other hand, can use Google as a method to find plagiarism. The idea is just to copy a small suspicious part of the assignment at paste it into Google and see if something suspicious comes up. [8]. This method is more suitable for text with no obfuscation attempts, but it is worth a try in other cases too since it is very easy to perform the method.

When the classrooms are small (with around 10 students) a teacher can find plagiarism just by remembering the previous assignments that they graded. There is a high probability that he/she will remember if two students have submitted a similar solution. But as the number of students grows this becomes more difficult. Usually, when the number of students grows, multiple teachers are involved, each teaching different groups of students, and this makes it even more difficult to detect plagiarism manually.

To overcome these problems similarity detection tools have been built. The basic idea behind such tools is to compare student submissions, calculate the similarity between each pair of students, and rank the pairs by descending similarities, which presents the most suspicious pairs on the top. All pairs with similarity above some predefined threshold are marked as plagiarised. Detection would be quite easy if students would just copy-paste when they plagiarise, but usually they use obfuscation methods to hide plagiarism. With obfuscation methods present, some detection tools might calculate a lower similarity between two submissions than it actually is, this can lead that some pairs not marked as plagiarised even though they are. Because of that, detection tools are constantly improved and new tools have been built to improve the accuracy of the detection. A review of detection tools is given in Chapter 4.

2.5 Obfuscation methods

Obfuscation methods are modifications of the assignment solution with the intention to hide plagiarism. Different obfuscation methods are used for text than for source-code. In this research only source-code obfuscation methods are discussed, although some of them can be used for text.

These obfuscation methods vary in complexity as stated in [44] whereby "*novice, student plagiarism mainly utilizes certain stylistic and syntactic changes, while expert programmers may introduce semantic changes (e.g. changing the data structures used, changing an iterative process to a recursive process, etc.) as well*". Faidhi and Robinson [44] introduce six levels representing different obfuscation methods. Each level represents obfuscation methods from all previous levels and adds one new obfuscation method. For example, level 1 represents changes in comments and indentation while level 6 includes the changes of previous levels and changes in the decision logic (i.e., changes in expressions). There is also level zero which represents no

change. In [23] an improvement variation of the Faidhi and Robinson classification is presented called *"Pyramid of Program Modification Levels"* which has eight levels.

Another more common classification introduced by Joy and Luck [85] is to divide the methods into two categories [38, 85]: *structural* and *lexical*. According to [139] all methods from Faidhi and Robinson's classification can be mapped to the Joy and Luck classification which was first done by Lancaster [100, p. 27]. The lower level methods up to level three are classified as lexical, representing simpler ways of obfuscation, and the methods from level three are classified as structural, representing the more sophisticated ways of obfuscation. However, Lancaster states that level three is questionable in which category it should go.

The review in [140] analysed 72 papers that mention some kind of obfuscation methods (OM) from which 16 distinct obfuscation methods where identified and described. According to [140] by looking at [5, 38, 44, 60, 185] all methods can be found. In [140] obfuscation methods are classified into four categories extending the classification of Joy and Luck. The categories are [140]:

- *Lexical changes (label L)* are "*changes which could, in principle, be performed by a text editor. They do not require knowledge of the language sufficient to parse a program.*"[85]
  This category includes the following methods [140]: *Visual code formatting*, *Comments modification*, *Translation of program parts*, *Modifying program output*, *Identifier rename*, and *Changing constant values*.

- *Structural changes (label S)* are changes that "*require the sort of knowledge of a program that would be necessary to parse it. It is highly language-dependent.*"[85]
  This category includes the flowing methods [140]: *Reordering independent lines of code*, *Adding redundant lines of code*, *Splitting up lines of code*, and *Merging lines of code* (includes: *Merging lines of code* and *Replacing the procedure call by the procedure body*).

- *Advanced structural changes (label AS)* are defined as "*a subcategory of structural changes that require more knowledge of program possibilities and relations between equivalent statements in a specific programming language.*"[140]
  This category includes the flowing methods: [140] *Changing of statement specification* (includes: *Changing the operations and operand*, *Altering modifiers*, *Datatype changes*) and *Replacing control structures with equivalents*.

- *Logical changes (label LG)* are defined as "*changes that except for structural changes also change the logic (flow) of a program and require a certain amount of programming skills and knowledge about the application being developed to be performed correctly. They are very unlikely to be performed by total beginners.*"[140]
  This category includes the flowing methods [140]: *Simplifying the code*, *Translation of program from other programming language*, *Changing the logic*, and *Combining copied and original code*.

From all of the aforementioned methods, according to [140] the most researched are: *Identifier rename*, *Reordering independent lines of code* and *Comments modification*. "*When looking at the aforementioned modifications to the different obfuscation methods one can say that many of them are actually code refactoring methods. Experienced programmers could use the knowledge about refactoring to help them hide plagiarism.*"[140] Students can also plagiarize without using any obfuscation method (simple copy-paste), so one special category *No changes* can be added to those four mentioned.

With the obfuscation methods identified, one can start improving detection tools so that obfuscations will have no effect. It is no easy task to reduce or eliminate the effect of some obfuscation methods. For some obfuscation methods it might be even impossible. The fact that new researches are constantly being carried out trying to improve the accuracy of detection tools, shows how difficult it is to break the various obfuscation methods.

CHAPTER 3

METHODOLOGY

In this research the overall goal, as stated in the introduction, is to identify the effect of PreProcessing Techniques (PPTs) on plagiarism detection accuracy in student programming assignments. In other words, the idea is to reduce or eliminate the effect of obfuscation methods by using PPTs. As a consequence, an increase in the accuracy of detection tools should be observed. In order to achieve that goal, and test the hypothesis given in the introduction, a methodology is used which consists of two parts: *quantitative* and *qualitative* analyses.

The quantitative analysis uses experimental design [46, 124] to test the stated hypotheses statistically. To do the comparison, fulfil the first goal (G1) and test the first hypothesis (H1), multifactor Analysis of variance (ANOVA) [46, 124] was planned, or if this was not possible a suitable non-parametric test was used instead. Variables in the analysis for the Real Student Solution (RSS) dataset are: tool, preprocessing technique (further referred to as a *technique* or *PPTs*), assignment solutions (further referred as *assignment*) and *accuracy* (F-beta calculated from precision and recall) which is the dependent variable. Different academic years (further refereed as *year*) were used as replications for the RSS dataset. The same years were used for each combination of technique, tool and assignment.

To test the second hypotheses (H2) and partially answer the research question (RQ1), ANOVA is not enough since it tests for overall effect and does not provide specific information about affected groups. To solve that, post-hoc tests or contrasts (for greater test strength) were planned to be used. During the design of contrasts, they were designed as orthogonal contrasts if necessary (more details in Chapter 8). As a qualitative analysis, a version of content analysis [166] was planned to fulfil the second goal (G2) and complete the answer to the first research question (RQ1). Found pairs were analysed manually for patterns indicating why some pairs were or weren't found. This was performed for interesting combinations of the tool, assignment and technique based on the result of the statistical analysis and the top ranked matches.

To the author's knowledge, no experimental design has been previously performed to test the effect of preprocessing techniques, so every aspect needs to be documented. There are studies that use preprocessing techniques (Chapter 7), studies that compare tools (Chapter 4) and studies that use quantitative measures for comparisons (Chapter 5), but in most cases no statistical test was used to see if there was any statistically significant difference between the compared groups. Papers that were found, while analysing 150 papers during Systematic Literature Review (SLR), that use any statistical evaluation are: [32] where multivariate ANOVA was used, [2] where a Wilcoxon rank sum test was applied, [36, 38, 91] where a simple t-test was used, and [40] where Pearson's chi-square test was used. During the SLR outside of the 150 papers there were two papers [18, 69] found from the related domain of authorship attribution where ANOVA was

used, but the focus was different than in the plagiarism detection domain.

There are four main components used in this research: detection tools, preprocessing techniques, datasets and evaluation measures. Each component is described in a separate chapter and has its own related work section and detailed descriptions of how elements of the components are selected. To extract all of the relevant related work SLR was performed as described in Section 3.1. In Section 3.2 for each component the research constraints are described and a short description of the ethical issues for this research. Section 3.3 presents the plagiarism detection process that was used and Section 3.4 describes how the process was automated.

3.1 Systematic literature review

SLR was performed to analyse the related work for this research and answer ten questions. Eight questions, with detailed analysis and answers, are presented in [140], together with a detailed description of SLR protocol. The eight questions are: "*What is meant by source-code plagiarism in academia?, What is meant by source-code plagiarism detection in academia?, What obfuscation methods do students use to hide source-code plagiarism?, What detection tools are used and which are the best?, What algorithms are used in these detection tools?, What measures are used to compare the tools?, What datasets are used to compare the tools?, Where should one search for articles dealing with source-code plagiarism detection in academia?*".

Two other questions are: *Which are the most used preprocessing techniques?* answered in Chapter 7 and *Who are the top authors in the field of source-code plagiarism detection in academia?* answered in Section 3.1.1. The reason why SLR is published as a separate paper is because it goes beyond the scope of this study. Only the most related findings to the topic of this research are presented across the related work sections.

Articles that were extracted in SLR are the primary but not exclusive basis for the related work sections of each component. These sections include papers that were extracted from the searched databases for the purpose of SLR but also papers that were found ad-hoc over Google Scholar [1], Research Gate [2] and other sources at different dates.

SLR was performed on four databases: Scopus[3], ACM Digital Library[4], IEEE Xplore[5], ScienceDirect[6] and ISI Web of Science[7]. Databases were queried on three dates: 28.2.2015, 20.8.2015 and 29.8.2016. In Figure 3.1 number of articles found in each database per year is presented. In total 3069 distinct articles where extracted, after filtering the number of papers was reduced to the final 150 articles which were then analysed to answer the research questions as part of the SLR. For the purpose of this research, an additional 152 papers were analysed

[1] `https://scholar.google.hr/`
[2] `https://www.researchgate.net`
[3] `http://www.scopus.com`
[4] `http://portal.acm.org`
[5] `http://ieeexplore.ieee.org/`
[6] `http://www.sciencedirect.com`
[7] `http://www.isiknowledge.com/WOS`

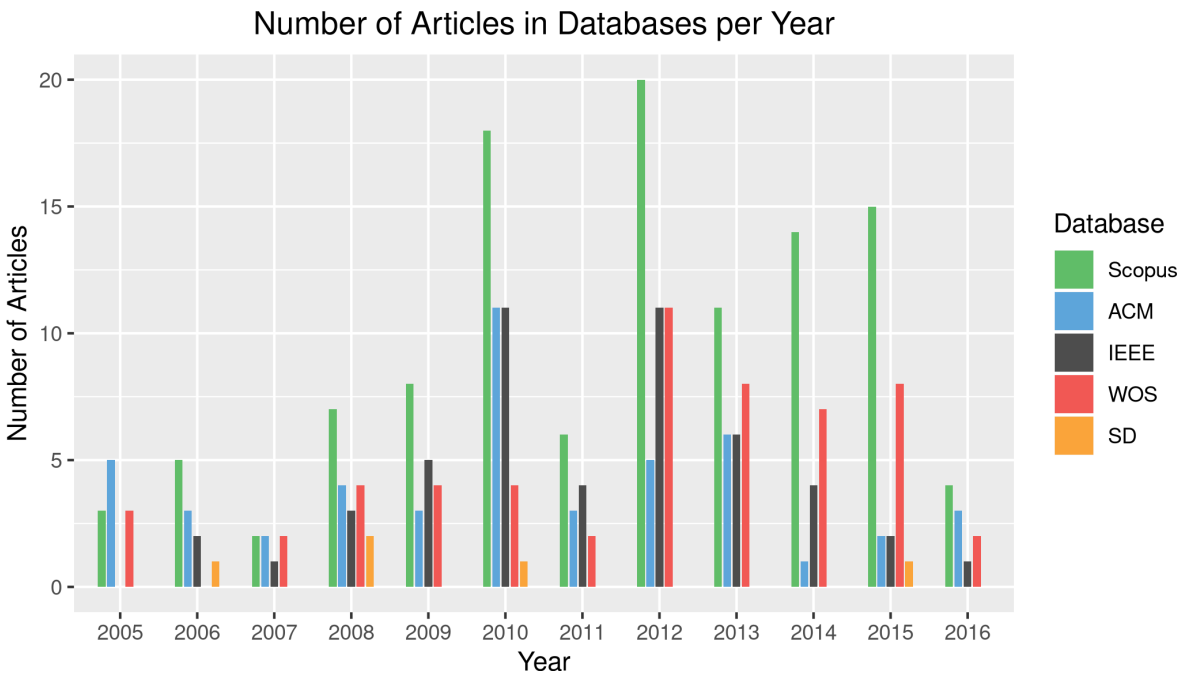


Figure 3.1: Number of articles in databases per year

from the 3069 which were extracted during SLR.

In the SLR part where the bibliographic analysis was performed, it was found that there were only 12 articles [24, 41, 44, 59, 60, 85, 147, 154, 178, 185, 187, 188] which had over 100 citations in Google Scholar. In Figure 3.2 the citation rate per year is presented, whereby each dot presents one article. Note that the y-axis in Figure 3.2 has a logarithmic scale. From Figure 3.2 one can see that real interest in the field of source-code plagiarism detection in academia started in 2005 and increased in 2010, but all 12 top cited papers where published before 2005, so it can be stated these are the ground papers for this field. More details are available in [140].

To have a good overview of the field for this research, in addition to the papers extracted during SLR a search was performed for top papers (citation rate above 100 in Google Scholar and in the top 5 at Google Scholar) which cite the 12 papers mentioned above. With this approach interesting papers with high citations (above 100) were found like [15, 30, 146, 163, 179] and some interesting PhD theses like [33, 100].

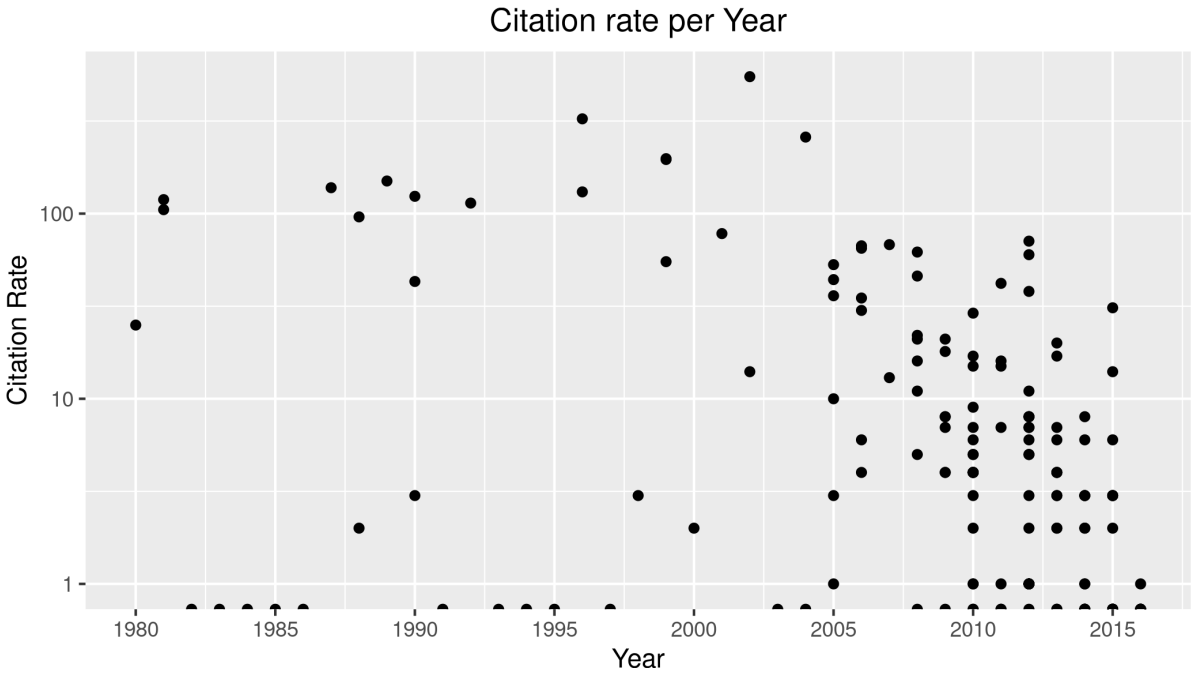


Figure 3.2: Citation rate per year

3.1.1 Top authors

Another way of finding relevant papers was to search for the main authors in the field. As part of the SLR the question was stated: "*Who are the top authors in the selected articles?*" In Table 3.1 the authors that have more than 2 articles published in the final 150 analysed articles are listed, with the corresponding articles and how many of those are the top cited with more than 50 citations.

The top three authors that have most articles are Joy, Cosma and Liu, but none of the articles of Liu has more than 6 citations. The same is for Wang, while Cho and Woo have only 1 article with more than 50 citations. Based on citation rate, the top authors are Wise and Joy followed by Cosma and Tahaghoghi. Based on both criteria and the citation sum for all articles listed in Table 3.1 it can be concluded that the top three authors are Joy, Cosma and Wise followed by Tahaghoghi. Although Wise has the most citations all three articles listed are before 1997, in comparison, the latest articles from Cosma and Joy are from 2015 and 2014.

Table 3.1: Authors with most articles in SLR and most citations

Author	Number of articles	Number of top cited articles	Articles
Joy, Mike	7	3	[31, 32, 34, 85, 86, 133, 196]
Cosma, Georgina	6	2	[2, 31, 32, 34, 86, 196]
Liu, DongSheng	5	0	[81, 120, 121, 182, 197]
Wang, Xin	3	0	[23, 192, 193]
Woo, Gyun	3	1	[79, 80, 102]
Wise, Michael	3	3	[178, 187, 188]
Tahaghoghi, Seyed	3	2	[5, 20, 28]
Cho, Hwan-Gue	3	1	[79, 80, 102]

3.2 Research constraints

Since there are four main components to this research constraints are also divided into four categories: detection tools, preprocessing techniques, evaluation measures and datasets.

3.2.1 Detection tools

There are many available detection tools as described in Chapter 4. To be able to use them as planned the tools must be open source or free tools that can be downloaded and used offline. Tools that operate as on-line services are excluded since the student solutions are confidential and cannot be shared according to our university's data protection regulations. This is more rigorous with the European data protection legislation[8] in place.

The criteria for selecting the tools is that they are commonly used and considered successful based on the results in the scientific literature. This kind of selection excludes some new tools like Spector [117] which are open source but have not yet been compared much to other tools, so the quality is questionable.

All internal preprocessing steps in a tool are disabled if possible, and if not, tools are tested with them. In such a case, this preprocessing technique is considered to be part of the tool. No changes to the tool are allowed that would change the algorithm or in some other way improve or reduce the quality of the tools. It is the point of preprocessing techniques to improve the tools so tools need to be in their original state.

To put all plagiarism detection tools in an equal position they are calibrated (by setting their parameters) according to a calibration test (described in Section 4.4 using the calibration method described in [138]) on a manually prepared dataset consisting of pairs extracted from a standardized dataset and manually created pairs for which similarity is approximately known. Also, to eliminate the effect of visualization, since different tools have different visualizations possibilities, all results need to be presented in a unified form for all used tools.

Because similarities between source-code and text exist, as noted in Section 2.3, and since

[8] `https://ec.europa.eu/info/law/law-topic/data-protection/data-protection-eu_en`

evidence exists in [157] that textual comparison can outperform specialized source-code comparison, in the experiment the textual versions of the tools are also compared and discussed. Because of that, tools that have the possibility to run textual comparisons are desirable.

Only tools that support the Java programming language are used because RSS dataset contains programs written in Java. Another reason is that Java is one of the most supported languages by detection tools, as it was found out in the SLR according to [140].

3.2.2 Preprocessing techniques

Preprocessing is recognized as an important step done before the actual detection. There are various preprocessing techniques, but there is very little research that would focus on the preprocessing itself (more details in Chapter 7). Because of that, selection is based on the availability of the currently most relevant techniques according to the scientific literature (Section 7.1).

Since no direct comparison of techniques is available it may be possible that the selected techniques are not the best in the field.

3.2.3 Evaluation measures

Evaluation measures are discussed in detail in Chapter 5 and the selected measure is F-beta. The F-beta measure is used as the accuracy measure since it is commonly used in other quantitative research. In this research F-beta measure is used to do comparisons of tools and techniques, because of that the term *comparison measures* is used as a synonym for the term *evaluation measures*.

F-beta is based on Precision and Recall, but to calculate Precision and Recall the exact knowledge of real plagiarized pairs is needed, but the nature of RSS dataset is such that this cannot be known with 100% certainty. Because of that, the number of real plagiarized cases will be estimated (similar to [178]) as the union of plagiarism cases found by at least one tool used in combination with every tested technique. This union of found plagiarized pairs will be examined by an expert (responsible teacher of the course) who will tell which ones are considered plagiarism. Also, since the expert examined all the assignment solutions before, all manually found plagiarised matches will be added to the union.

The exact procedure for finding plagiarized cases is defined in a flow chart diagram form in Chapter 6 under Section 6.3.1.

Threshold level

Another issue regarding F-beta is how to decide where to draw the threshold line (also known as a threshold level, cut-off threshold or threshold), some studies take the top n matches, others use all matches with similarities above some percentage (for example everything above 90%). Using a fixed number, whether it is the top n matches or a percentage, is not the best solution.

The problem with fixed top n matches is that the number of plagiarised matches varies from assignment to assignment which makes it difficult to decide what number to chose. Also, using a fixed top n matches could cause that, for some assignments, the perfect F-beta can not be reached, not even theoretically. Instead of using fixed top *n* matches for all assignments, it would be better to use flexible top n matches where n is the number of plagiarised matches in one assignment. In this way there is no bias in choosing the number, also theoretically it is possible for every tool to reach a perfect F-beta.

The problem with a fixed percentage number is that tools sometimes report quite different percentages for the matches over the whole dataset, which again makes it difficult to decide what number to chose. Choosing the percentage is even more difficult since it depends not only on the assignment but also on the tools that are used. The difference is especially big between the textual versions of the tool and the version of the tool that was primarily built for source-code detection. Because of that if a low percentage is chosen the tools that report higher percentages (source-code version) are at a disadvantage, if a high percentage is chosen it puts tools that report lower percentages (textual version) at a disadvantage.

The solution to that problem is to use a flexible threshold. The idea is based on the statement of suspects given by Brixtel et al. [17]: "*We consider two or more documents to be suspects if they are much more similar than the average similarity between documents.*" The same idea was also used by Freire [56] where the AntiCopias (AC) system was built and the plagiarised cases were marked based on the position on a histogram representing all matches. The underlying idea is the same, it is not important how high the similarity of a match is, but how it is in relation to the rest of the matches.

Following these ideas, a new method for determining the threshold level is created. The method does not use a fixed percentage rather it looks at all pairs, calculates the mean similarity and then takes all matches for which the similarity percentage is (for example) 3 standard deviations from the mean. Since there is a high chance that the data are not normally distributed it would be better to use 3 inter-quartiles (IQR) from the median, since it eliminates the extremes in the threshold percentage calculation. This method would make the threshold level flexible not only for each assignment but also for each combination of tool and technique in one assignment.

3.2.4 Datasets

In this research real data from student solutions alongside some available open source-code dataset were tested. As the standard open dataset, the SOurce COde Reuse (SOCO) dataset from the PAN@FIRE2014 competition described in [52] is used. The RSS dataset, a collection of personal students' solutions gathered over several years, is used. More details about datasets are given in Chapter 6.

The amount of data in the RSS dataset that needs to be analysed must be limited since the marking of plagiarised matches must be checked and confirmed manually. This must be done for each combination of tool, technique, year and assignment in the RSS dataset. So restrictions

on RSS are: Java programming language, assignments from three academic years up to six academic years, minimum of two assignments and maximum of four assignments per year. In addition, a maximum four tools and up to six preprocessing techniques (or combinations of techniques) were used. The author manually examined the pairs in the qualitative analysis and to ensure objectivity, an expert (as stated in Section 3.2.3), was included to mark the real plagiarized matches.

Since the RSS datasets is a collection containing solutions of assignments for multiple years, in the rest of this document the individual groups are marked as 'year-assignment' (for example 2013-2014-A1). The main difference between the RSS dataset and datasets in other studies, in the area of source-code plagiarism detection in academia, is that they use solutions which have an average with up to 500 Lines of code (LOC) from introductory programming courses. This research deals with source-code files with an average larger than 1,000 LOC and up to 6,000 LOC. In [134] the authors used source-code files with many LOC that, however, were not obtained in the 'academic' environment.

3.2.5 Ethical issues

In the research data preparation process, anonymization was performed on real student assignments with intent to protect all personal information. During publication only summary data are presented. When it is necessary to show specific similarity cases which include source-code parts from individuals one should not be able to identify the author.

3.3 Plagiarism detection process

Plagiarism detection can be performed using four steps [139]: 1. Preprocessing of input data, 2. Similarity detection, 3. Visualization of results, and 4. Confirmation of plagiarized cases. To get to the fourth step where the calculation of F-beta is possible in this research, a process is used which is based on the twelve step process described in [95].

Since the process in [95] was made for one tool, focusing only on the template exclusion preprocessing technique it was necessary to make some changes. The detection process that was used in this research has seven phases (some containing several steps) and goes beyond simple confirmation of plagiarized cases. Each phase is described below with one sentence stating the overall goal of the phase followed by a description of how the phase is used in the context of this research. The first six phases are part of the general plagiarism detection process, meaning they can be used by teachers to find plagiarism, while the last phase is specific for this research. The phases of this research are:

1. Download phase — The goal of this phase is to download and create the datasets.

    - For the RSS datasets the submissions were downloaded as zip archives form the learning management system Moodle. The submissions are grouped by assignment, and assignments are grouped by years.

- The SOCO dataset was downloaded from the official website of PAN@FIRE2014 competition [9] together with the files that contain information which matches are real plagiarised matches. [10] Detailed information about the SOCO dataset was extracted from [52]. The SOCO dataset contains files which are grouped into assignments, and assignments are grouped into collections. One file in the SOCO dataset is the same as one submission in the RSS dataset.

2. Preparation phase — The goal of this phase is to prepare the datasets so that they can be preprocessed. In the case of SOCO datasets the whole preparation is a simple extraction of two archives, one for each collection. In the case of the RSS dataset this phase is divided into three steps:

   (a) Archive extract — All archives are extracted and as a result one directory is created for each student submission containing all files and directories of the submission;

   (b) Directory rename — Each submission directory is renamed so that it contains the id of the student who submitted the solution;

   (c) File merge — All files of one submission are merged into one file, at this point files which were not of interest like images are deleted.

3. Preprocessing phase — The goal of this phase is to run the preprocessing techniques, but it can be skipped if no preprocessing is performed. In this phase selected preprocessing techniques are executed on each merged file for every assignment in the RSS dataset and SOCO dataset. In case of the template exclusion technique multiple steps are involved as described in Section 7.5 based on the process described in [95]. This phase at the end has, as a result, multiple instances of each assignment whereby each instance contains files processed by one technique.

4. Detection phase — The goal of this phase is to calculate the similarities between each file in one assignment instance. Detection is performed on each instance of the preprocessed assignment with each selected tool. At the end of this phase the number of instances of the preprocessed assignments is multiplied by the number of tools which are used.

5. Visualization phase — The goal of this phase is to visualize the data in a unified way regardless of the tool or technique used. The research data are presented in table format and with all necessary information needed in the following phases, but it is possible to present the data in graph format. Since presenting the data as a graph and discussing the possibilities of such visualization is out of the scope of this research.

6. Confirmation phase — The goal of this phase is to confirm the real plagiarised matches, and in this research to have the possibility of calculating F-beta. For the SOCO dataset this

[9] `http://users.dsic.upv.es/grupos/nle/soco/`

[10] Thanks to Paolo Rosso (prosso@dsic.upv.es) for sharing the password for the datasets.

was already done and as stated in download phase this information was downloaded from the official website. For the RSS dataset this phase was done by an expert as explained in Section 3.2.3.

7. Analysis phase — The goal of this phase is to analyse in depth the results quantitatively and/or qualitatively to find new things. The results of this phase are described in Chapter 8.

From the phase descriptions, it is evident that performing such research takes a lot of time if performed manually. Because of that a system was developed, called Multiple Plagiarism Checker (MPC), to help automate a bigger part of the process. While the download and confirmation phases were explicitly performed manually, other phases were fully or partially automated. Because an automated system was used in the preparation phase a set-up step was needed where the configuration files for the system were prepared and the individual collections were put in corresponding directories as needed by the system. Except for the MPC system, a smaller part of the process was automated using the system 'R'. The MPC and 'R' systems [11] are described in more detail in Section 3.4.

3.4 Process automation

The idea of automating the process for the evaluation of detection tools is not new. Cebrian et al. [22] created a benchmarking procedure for plagiarism detection engines. They presented a procedure where the test cases are automatically generated for the programming language APL2. They tested their procedure on the AC tool [56]. While the idea is interesting the tool that they built is not very useful because of the choice of the programming language. Most detection tools are made for Java of C language so it would be better to have this kind of benchmarking tool for these languages.

With the same basic purpose, to ease the evaluation of detection tools, the MPC system was built. The main difference between the tool described in [22] and the MPC system, is that the MPC system does not generate the test cases automatically, but it has other benefits. The MPC system eases the evaluation of detection tools so that it automates the underlying process of plagiarism detection.

The MPC system supports the process by preparing the data for the detection, giving the possibility to use preprocessing techniques, enabling the usage of multiple tools, visualizing the results in a unified form for all tools, and calculating the F-beta and other statistics needed for analysis phase based on the detection results. In addition, the MPC system has integrated the calibration method, described in detail in Section 4.4, it supports the process of manual marking of plagiarised matches, enables the creation of new combinations of preprocessing techniques, and it has the possibility for simple evaluation of preprocessing techniques. The MPC system

[11] The complete source-code of the MPC system and all used R scripts are available at `https://github.com/matnovak-foi`

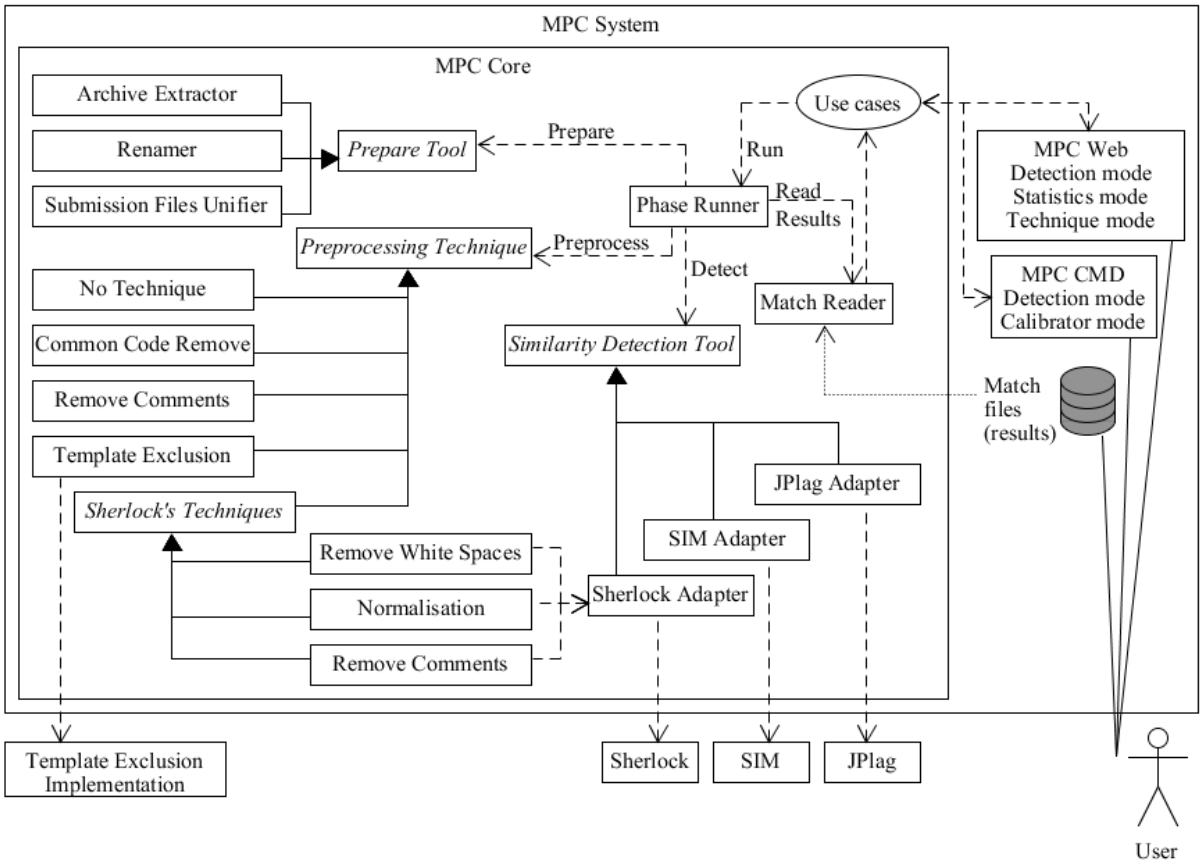


Figure 3.3: High level architecture of MPC system

can also be used as a simple plagiarism detection system by teachers which will run the detection on multiple tools and perform the desired preprocessing techniques.

The MPC system has four working modes where the main mode is the detection mode which goes through all phases of the plagiarism detection process. The other three modes are: 1) calibration mode – used for calibration in Section 4.4, 2) technique selection mode – used for technique selection test in Section 7.6, 3) statistics mode – used for calculating statistics that are input for the analysis phase (Chapter 8). The overall architecture of the system is presented in Figure 3.3, and in Appendix C a detailed description of the architecture is given.

The MPC system has been developed in the programming language Java [12] using NetBeans [13] and ItelliJ IDEA [14] Integrated Development Environment (IDE). The main mode can be run as a command line application or as a web application. Calibration mode can only be run over the command line, while the other two modes can only be run over the web interface. The web interface was designed using the PrimeFaces [15] framework on top of basic Java Server Faces (JSF). An Apache Tomcat server [16] was used to run the web application.

[12] `https://www.oracle.com/technetwork/java/javase/downloads/jdk8-downloads-2133151.html` - used version jdk1.8.0u144

[13] `https://netbeans.org/` - used version 8.2

[14] `https://www.jetbrains.com/idea/` - used versions from 2017.2.5 to version 2018.2.4

[15] `https://www.primefaces.org/` - used version 6.1

[16] `http://tomcat.apache.org/` - used version 8.0.27

To ensure validity of the MPC system, Test Driven Development (TDD) was used [13, 55], as invented by Kent Beck. TDD consists of three steps where in the first step a test is written until it fails, in the second production code is written only to pass the failing tests, and thirdly code is refactored to keep it flexible. The three steps are repeated which ensures that every part of the code works as it is supposed to. That TDD works is confirmed by various research like [25]. Also during development of the MPC system there were many occasions where tests broke and prevented a potentially big bug which probably would be gone unnoticed. To refactor the code guidelines described by Fowler et al. [53] were followed. To keep the code clean and easy to maintain principles described in [113, 114], this includes using best practices and design patterns [57]. The basis of TDD are unit tests, and to keep the tests clean and to get ideas for writing tests the suggestions presented by Meszaros in [122] are used.

Except for unit tests, to test larger parts of the system acceptance tests were written which once a module was completed served as regression tests. With tests it was ensured that the system will not have bugs which would influence the outcome of the research results, at least the probability of such case is minimized since the majority of the production code lines have at least one test to ensure it works as expected. The code coverage based on the Clover's coverage report is presented in Appendix D. The Graphical User Interface (GUI) was tested manually and finally several test runs were performed before the application was put into production, in other words before the main experiment was executed.

One factor while developing a larger system like MPC is the developer itself, and to keep focus the Pomodoro Technique was used [29] and also principles of professional programmers were followed described in [77, 115, 141]. Since the chosen programming language to develop the MPC system was Java, coding conventions and ideas for Java where used as suggestion from books like [16, 45].

3.4.1 Isabella cluster

Automation of the detection process using the MPC system saves a lot of time, but in this research to run all combinations of techniques, tools and assignments only for RSS dataset still takes a long time. To save time the various detections and preprocessing were run using an Isabella Cluster [17]. The Isabella Cluster consists of 104 computer nodes which contain 208 CPU processors with 2496 processing cores, 12,5TB of RAM and 200TB of disk space.

The benefit of using a cluster is twofold. First, the execution time is reduced compared to a laptop or classic server, and secondly, some processes can be run in parallel. The preprocessing of source-code was run first, and since every preprocessing technique operates on a different directory they can be run in parallel. Once all preprocessing is done the detections can be run. Detections with different tools on assignment instances, preprocessed with different techniques, also operate on different directories so detections can be run in parallel.

To give an example, the detection for SIM and JPlag on the Isabella Cluster took up to 4

[17] `https://www.srce.unizg.hr/isabella/`

minutes. On average the execution was 1 minute, in comparison on a laptop of 8GB RAM with Intel Core i5 CPU at 2.6GHz the average execution time for several test runs was approximately 10 minutes. This is already 10 times faster, but since there were 144 runs that needed to be done with one tool (6 techniques * 4 assignments * 6 years), which on a normal laptop can not be executed in parallel, is an even bigger time save. On the cluster on average up to 20 of these runs were run in parallel, so instead of 200 minutes (20 runs * 10 minutes per run), the execution time was 1 minute (20 parallel runs, 1 minute per run). The biggest save was with the tool Sherlock, where an average run took from a few hours up to 14 days on the cluster.

Detailed analysis of the runs is out of the scope of this thesis but one is for sure that Sherlock is definitely less optimized in terms of speed in comparison to SIM and JPlag.

3.4.2 Automation of analysis phase

All statistical calculations were performed automatically using open-source system R [18] based on the MPC output. The development environment RStudio [19] was used. Since R is a programming language there is the possibility to use TDD by installing the *testthat* package [20]. Using unit tests, the most newly created functions for processing the quantitative data (like creating frequency tables) were checked for correctness. Graphs, as the interfaces in the MPC system, were examined manually and checked for correctness. In addition to the R system, Microsoft Excel was used to do some simple calculations or to prepare tables that were input to the R system. Full list of packages in R used for statistical analysis is presented in Appendix L.

Since the analysis phase was also automated, it could be easily checked against what would happen if there were, for example, ten more false negative pairs as discussed in Section 5.2. Suppose ten plagiarized pairs were not found using all combinations of tested tools and techniques, then the Recall can be easy recalculated with ten more pairs added to the total number of plagiarized pairs. Finally, the whole calculation can be repeated and results compared with the previous results.

[18] `https://www.r-project.org/` - used version 3.3.3
[19] `https://www.rstudio.com/` - used version 1.1.453
[20] `http://testthat.r-lib.org/`

CHAPTER 4

SIMILARITY DETECTION TOOLS

The first components needed to perform this research were similarity detection tools. To select detection tools that matched the mentioned research constraints (Section 3.2) the literature was consulted, and related work is presented in Section 4.1. Once the tools were selected (Section 4.2), in order to have an objective comparison of the tools, the tools' parameters were calibrated (Section 4.4).

4.1 Related work

Using software to automatically detect plagiarism[1] in student programming assignments is not a new idea, and already in 1976 Ottenstein [146] dealt with this problem in his article "*An algorithmic approach to the detection and prevention of plagiarism.*" To calculate similarities, Ottenstein used Halstead's metrics [65] which are based on counting operands and operators. For each submission four numbers are calculated (unlike the single number usually used today): the number of unique operators, the number of unique operands, the total number of occurrences of operators, and the total number of occurrences of operands. Those four numbers present a single vector describing one code document. To get a feeling for similarity and to identify potential plagiarism he compared a submission vector to a mean vector (containing mean values of all submissions) using a multivariate normal density function and covariance matrix. Halstead's metrics [65], described in detail in the book "*Elements of software science*"[66], were not invented for similarity detection, rather they were software metrics to be used to predict durations and error rates of computer programs.

It was not long after that the first plagiarism detection systems where built. In 1980 a system called Instructional Tool for Program ADvising (ITPAD) was developed [158] to detect plagiarism in programs written in FORTRAN. ITPAD extended the four metrics used by Ottenstein with 10 new metrics. In 1981 Donaldson et al. published an article [41] describing a system which was able to detect plagiarism in programs written in FORTRAN, COBOL and BASIC. Donaldson also pointed out the main problem with plagiarism detection tools [41]: "*It is certainly safe to say that neither the detection system described in this paper nor any other detection system will find all occurrences of plagiarism. There is an inherent tradeoff between a highly discriminatory system, which overlooks some instances of cheating, and a less discriminatory one which flags many dissimilar programs.*" In the same year, Grier published his paper [60] about a system called Accuse developed for plagiarism detection in programs written in Pascal. These first programs used algorithms that are today known as attribute counting (or fingerprint)

[1] for reminder, more accurate it would be to say detect similarity

algorithms, because they count the various occurrences of variables, loops, etc.

Already in these early days of automatic plagiarism detection various obfuscation methods were identified [41]. Because the systems were fingerprint based one can easily think of how they can be tricked. Researchers started to compare the tools [147, 185] to see how good they are performing and how they deal with the obfuscation methods.

Since then, numerous researchers have attempted to improve the accuracy of plagiarism detection using new approaches [31, 143, 201], creating new methods for similarity calculation [151, 155, 170], building new techniques for plagiarism detection [81, 125, 165], using preprocessing techniques [6, 38, 95], improving presentation of the data [56, 123, 127], and so on.

Classification of the various algorithms used in the tools is presented in [140], identifying 16 categories. What stands out is that old algorithms like attribute counting methods were mostly used at the beginning but can still be found in new tools, usually in combination with algorithms from other categories. Another interesting thing is that most algorithms use tokenization. To be more precise, 53 articles from 120 articles which report developing a new tool use tokenization. Tokenization is a way of converting source-code into predefined tokens to represent some construct of the programming language like a variable. The idea is that in such a way the simple obfuscation methods like renaming will have no effect since all variables are replaced with a token (for example *int a=10;* would be converted to something like *varType varName = varValue;*). While tokenization can be seen as a preprocessing technique [132], in this research it is viewed as a part of an algorithm inside of a tool because it is common in many tools.

Some source-code plagiarism detection tools that have been developed include YAP3 [188], MOSS [163], JPlag [154], and Sherlock Warwick [85]. There are many tools not directly built for source-code plagiarism detection in academia but that could eventually be used, such as tools built for textual plagiarism.

4.1.1 Related areas to source-code plagiarism detection

Because of a missing standard for referencing in source code, among other reasons, most tools or services built to find plagiarism in text do poorly on source-code. For example, an experiment described in [183] shows that the popular Turnitin system which works well with text (as shown in [184]) performs poorly on code. On the other hand, the same experiment [183] also shows that the SIM tool [61] textual version did a good job. Similarly, experimental results from Ragkhitwetsagul et al. [157] show that textual plagiarism detection can outperform some specialized source-code detection. Note that in the same way some tools built primarily for source-code have an option to detect textual plagiarism, such as [186], but the comparison is needed to see how well they perform in comparison to tools primarily built for identifying textual plagiarism.

Apart for tools from the area of textual plagiarism, there are other related areas which produce tools that could be used for source-code plagiarism detection. Some of the related areas are:

- Authorship attribution — The idea of authorship attribution [172] is to determine whether two unrelated programs are written by the same author. This is done by checking the similarity usually in a stylistic sense. Authorship attribution and plagiarism detection areas overlap [19] in some points, but primarily authorship attribution has a different focus to plagiarism detection. Authorship attribution complements [19] plagiarism detection well. "*Plagiarism detection software is good at identifying works with matching parts . . . but they are not suited to discovering the true author of works.*" [19] Combining plagiarism detection and authorship attribution has been done for example in [144, 199].

- Code clone detection — Code clone detection is very similar to plagiarism detection since it also searches for copied code, but there are major differences in perspectives. According to [134] in clone detection the focus is to find clones inside of a single system to improve the system's maintainability, while in plagiarism detection, multiple systems are compared to find copied code between them to avoid legal issues. Also, in clone detection the copied code is slightly changed or not changed at all, while in plagiarism detection, the copied code is modified and refactored to hide the copied parts. That clone detection techniques can be used for plagiarism detection is presented in [17], and an experiment that compared 6 systems for clone detection on industrial programs was reported in [15].

- Code reuse detection — Programmers are often encouraged to reuse code for multiple purposes. The idea of code reuse similarity calculation is to find similar code that was used in different situations primarily for the purpose of improving the program and removing redundant code (for example [103]). The biggest difference between plagiarism detection and code reuse detection is that in code reuse there is no intention to hide the copying. However, it must be stated that some techniques from code reuse could be used to hide plagiarism and therefore some detection techniques to find code reuse can be useful for plagiarism detection like [48, 75]. Specifically in [48] cross-language source code re-use detection is described which could be very useful for plagiarism detection.

- Industrial plagiarism detection — There is a difference between detecting plagiarism in the academic environment and in industry. The main difference is that programs in the industry are much larger than student assignments so the techniques could be different. Also in industry the issues are more about copyright, and finding stolen code and proving its provenance. In academia source-code from a student's homework is available to teachers, who know exactly who submitted what. As stated by Whale [185]: "*Commercial plagiarism detection is largely a matter of establishing a link between a pair of products that perform identically. Similarity detection amongst student programs compounds this problem with the more difficult task of firstly identifying the dependent pairs or groups from a population of several hundred.*" For example, tools like Code Plagiarism Detection Tool (CPDP) [134], CodeMatch [194] or GPLAG [106] were tested in an industrial envi-

ronment but, as with every other related field, there is a certain probability that these tools could be effective in an academic environment.

- Assignment evaluation — There are tools which are built for automatic evaluation of assignments but sometimes they integrate plagiarism detection mechanisms. For example in [160] the main focus is on the automatic assessment of programming exercises but the paper mentioned that the tool can be used also for plagiarism detection.

Another related plagiarism detection area is the area dealing with specialized tools that try to detect plagiarism in software from compiled code. Some of them have been used in an academic environment like [5, 80, 145, 155]. Also, there are specialized tools for plagiarism detection in Microsoft Access [119], Excel [169], SQL [191], and similar software packages.

4.1.2 Comparison of plagiarism detection tools

According to [140], from the domain of source-code plagiarism detection in the academic environment, there are 120 articles describing the development of a new tool. Unfortunately, most tools are not available even though most of them were developed after 2010. These tools differ in many elements like availability off-line or on-line, generic or programming language specific, open-source or commercial, etc. Articles that review and compare the tools by features [101, 116, 126, 139, 167] and/or perform experiments [52, 63, 71, 157, 178, 183, 185] are a big help when looking for a tool. To the author's knowledge, the largest experimental statistical comparison of detection tools was done by Ragkhitwetsagul et al. in 2016 [157], who compared 30 tools which they divided into four areas: clone detection tools, plagiarism detection tools, compression tools and other tools. They did pervasive modifications in code using tools for source code bytecode obfuscation and source code normalisation using decompilers.

To decide which tools to use one can first look at the feature comparisons that are gathered from the literature. One such feature comparison is presented in Table 4.1. Such comparisons are good to quickly filter the tools by some feature, for example, if looking for a tool that supports Java programming language then a quick look at such a table can eliminate some of them. Next, one can look at other features that might be of interest, but once all tools have been filtered based on features it can happen that multiple tools are left over.

More precise statistical or qualitative comparisons using experiments can help to make a final decision. The problem as identified in [140] is that out of 120 articles describing a new tool only 55 compared the tool to others, and of those 55 only 27 were compared using some kind of quantitative measure. The quality of such tools is therefore questionable. Another problem is that two tools can have the same name, for example, SIM from Githcel and Tran from 1999 [59] (further refereed as SIM-GT) and SIM from Grune 1989 [61] (further refereed as SIM), or Sherlock from the University of Warwick (further refereed as Sherlock) [85] and Sherlock from the University of Sydney (further refereed as Sherlock-Sydney) [176], which can be confusing and lead to using a wrong reference.

Table 4.1: Overview of plagiarism detection tools

Tool	Open Source	Year	Supported Languages	Local/Web
Code Match [194]		2004	36 languages supported: Java, C++, Python, etc.	Local
GATE [135]	Yes	2009	Java, UML	Local
GPlag [106]		2006	C, C++, Java	Local
JPlag [154]	Yes	1996	Java, C, C++, Scheme, text	Local
Marble [62]		2007	Java, C#	Local
MOSS [163]		1994	23 languages supported: C++, Java, JavaScript, Python, etc.	Web
PDE4Java [78]		2007	Java	Local
PIY [143]		2015	Visual Basic tested	Local
PlaGATE [33]		2008	All	Local
Plaggie [3]	Yes	2002	Java	Local
SCSDS [38]		2013	C#, C++, Java	Local
SIM Grune [61]	Yes	1989	C, Java, Pascal, Modula-2, Miranda, text	Local
Sherlock Warwick[85]	Yes	1999	All, specialized for Java & C++	Local
Sherlock Sydney [176]	Yes		text	Local
Spector [118]	Yes	2015	Java	Local
SSID [150]	Yes	2012	C, Java	Local
YAP3 [188]		1996	Pascal, C, LISP	Local

When analysing Table 4.1 it can be noticed that most source-code plagiarism detection tools are built for specific programming languages. This is confirmed by the Systematic Literature Review (SLR) [140] were from 150 analysed articles most subsequent research deals with only three programming languages: C (49 articles), Java (48 articles) and C++ (33 articles). Other languages have been present in 5 or fewer articles, except for Pascal which was used in 12 articles but mostly in articles before 2000, while the top three languages are in articles after 2000. Interestingly, only four articles use PHP or the .NET/C# programming language. These numbers show that for most languages there is only a limited number of tools available, but new research is constantly being undertaken. For example, in 2017 there was at least one research paper dealing specifically with PHP [181] and one with .NET [156], and similar observations can be made about other popular programming languages.

Tools can be also be classified based on where they search for plagiarism [100]: intra-corporal or extra-corporal (corpora is the plural of corpus, which represents the set of all student submissions). In this research, a corpus refers to a dataset. Most tools found in the literature regarding source-code plagiarism detection work are intra-corporal, which means they search for plagiarism between submissions in the same dataset. Some tools have the ability for extra-

corporal searches like [143], where a large repository of previous submissions was used. This extra-corporal search is not the same as searching the Internet for textual plagiarism [21, 108] but it is going in that direction.

4.2 Selection of tools

When doing research which measures the effect of preprocessing on detection tools, it makes sense to use the best tools in the comparison. The problem is that it is difficult to say which tools are best since there has been no research comparing all tools. This is not surprising since most tools are not available for researchers (excluding the original authors) to be analysed. The next best thing is to use tools that are available and that were mostly compared. From the most compared tools the idea is to select tools which have the best ranking.

The top five tools which were compared most according to SLR presented in [140], also which where most mentioned in the 150 articles analysed in the SLR are: JPlag[2] – 37 comparisons and mentioned 43 times, MOSS[3] – 29 comparisons and mentioned 37 times, Plaggie[4] – 6 comparisons and mentioned 7 times, SIM[5] – 4 comparisons and mentioned 6 times, Sherlock[6] – 4 comparisons and mentioned 9 times. That those tools are the most compared is expected since when looking at Table 4.1 one can see that all of them except for MOSS are open-source and available for off-line usage, and MOSS is free to use as a web service. Some of the tools have the possibility to work in two modes (subsequently referred to as *versions*) — specialized source-code comparison or textual comparison. In this research, each version of a tool is viewed as a separate tool. Since in this thesis the focus is on the Java programming language the specialized source-code comparison mode is referred to as the *java version* or *source-code version*[7] and the textual comparison is referred to as the *textual version*[8].

All five mentioned tools appear to be good choices in experimental comparisons, whereby Sherlock is the least compared experimentally from those five. In [157] the SIM Java version ranks second and the JPlag textual version ranks fifth out of 30 compared tools. More importantly, both are the top two in the category of plagiarism detection tools, and after that come Sherlock-Sydney (rank eight), Plaggie (rank twelve), JPlag Java version (rank twenty one), and last the SIM textual version (rank thirty). MOSS and Sherlock were not compared in this research. On the other hand, in [183] JPlag ranked first, SIM second and MOSS third out of 18 compared tools, although Sherlock and Plaggie were not compared, and it is not clearly stated which version of SIM and JPlag was used. In [71] MOSS was marked as the top tool outperforming JPlag, SIM, Plaggie and five other tools while Sherlock was not compared. In [33] Sherlock outperformed

[2] `https://jplag.ipd.kit.edu`
[3] `http://theory.stanford.edu/~aiken/moss`
[4] `https://www.cs.hut.fi/Software/Plaggie`
[5] `https://dickgrune.com/Programs/similarity_tester`
[6] `http://warwick.ac.uk/iasgroup/software/sherlock`
[7] To recognize the source-code version of the tool, the label *.java* is added to the tool name.
[8] To recognize the textual version of the tool the label *.text* is added.

JPlag.

These research results can not be compared exactly since they used different datasets (more details in chapter 6) and it is possible that they used different parameter configurations (more details in section 4.3) of the tools and so on. The only thing they have in common is the statistical F-beta measure (details available in Chapter 5) that they used. Although the research results are not comparable, it can be concluded that each tool is a good choice and that, depending on the dataset and the situation, one tool can outperform another.

Those five tools all meet the most research constraints mentioned in Section 3.2. In this research, unfortunately, MOSS is not an option since according to our university's data protection regulations it is not allowed to upload student submissions to a third party public server. Since experimental evidence exists [157] that textual versions can outperform the Java version it was decided to use only tools that have the Java version and textual version according to the research constraints. Because of that, Plaggie needs to be excluded. This is not a big issue since Plaggie is based on a similar algorithm to JPlag so it does not make sense to use two quite similar tools. Since JPlag is better developed, has been the subject of more comparative studies, and outperforms Plaggie, it was decided to use JPlag.

To limit the number of runs, and since also textual versions are used in the experiment, it was decided to use only three tools. Since the three tools *JPlag*[9], *SIM*[10], and *Sherlock*[11] meet the research criteria and have the appropriate licence, they were selected for the experiment in this research. In the rest of the text, if a specific version of the tool needs to be discussed the appropriate label is added (java or text) otherwise the discussion is generally about the tool regardless the version.

It can be difficult to compare three tools where every tool has its own way of running with a different interface. To have a unified interface for all tools, and to eliminate the impact of the interface on the results and decision making, also to ease the process of running multiple tools, the Multiple Plagiarism Checker (MPC) system was built (described in Section 3.4). The MPC system is used to run the detection on all tools, or to perform detection with multiple tools, with one click. That handling of multiple tools is a problem was also noticed in [153] where the authors built a system called LOUPE that "*manages the execution of different plagiarism tools and generates a consolidated comparative visualization of their results*". LOUPE as MPC system supports three tools: JPlag, SIM and Sherlock. Unfortunately, the LOUPE system was not available for use at the beginning of performing this research, and LOUPE did not support the different preprocessing techniques, even though it had the preprocessing stage but with a different focus.

[9] Version 2.11.9 with GPL v3 licence

[10] Version 3.0.1 with specific licence given in appendix B

[11] Sherlock 2003 V5 with GPL v2 licence

4.2.1 Changes on selected similarity detection tools

As already stated the selected tools were all integrated into the MPC system. To be able to integrate the tools it was necessary to make some changes to the tools themselves. The changes were not allowed to improve or decrease the tools' capability of detection, or to cause any modifications on the original algorithm. When changing the tool it was important to keep the tool's capabilities as is. The reason for that is if one tool is changed and this improves the tool's capabilities, then it is necessary to ask why the other tools were not changed also. These tools are treated as black boxes which get an input and give an output which is then analysed. The changes that were done on the tools are as follows.

- No changes were made to SIM, except that it needed to be compiled for Linux and that it needed to be run twice. SIM has two modes of running, one gives the side by side comparison and one gives the percentages. Since both items of information were needed SIM was always executed in both modes sequentially.

- Two changes were made to JPlag.

  – First, the access to classes, functions and variables was changed from private to public. JPlag has a very complicated report in HTML format which was hard to parse and it does not give all the needed information about one match, so the method for generating the report was overridden so that the MPC system was able to get the raw results.

  – Second, JPlag initially generated only results for the top 1000 matches, but to have the complete set of matches that JPlag finds this number was set to the maximum integer value that is possible. Usually, in the experiment, there were up to 2000 matches. This change is not a problem since the algorithm already finds this matches anyway, the lowest similarities were just pushed down once 1000 matches were reached.

- Multiple changes were made to Sherlock.

  – The most changes were related to changing the access to classes, functions and variables from private to public to be able to read configuration values used in testing to ensure everything was set up correctly, and to use some functions to ease the preparation of configuration files.

  – The second change was done because Sherlock had the tendency to break down when operating on comparisons with around 50 files. The reason for that was some statements were missing in the code to close the used files, so this was added but this did not change the algorithm at all.

- The third change was regarding the preprocessing possibilities of Sherlock. To be able to test the effect of preprocessing, Sherlock needs to run without any preprocessing. This was not a problem for the textual version but for the Java version Sherlock always runs normalisation before the actual detection so this was turned off (more precisely, it was changed to use the original file rather than the normalised file), again this did not change the detection algorithm. More details about the preprocessing options of Sherlock are given is given in Chapter 7.
- The fourth change was to remove '#Line' printing in the report. '#Line' was a mark from Sherlock when it removed a part of code when preprocessing. This has nothing to do with the detection itself but with Sherlock's preprocessing techniques which were used. This line would cause problems for other tools to parse the code with that line. Sherlock had in the detection algorithm specified to ignore such lines, but the other tools didn't and because of that this '#Line' printing was disabled.
- The fifth change was to add to Sherlock the ability to process files that have extensions different than *.java* like *.txt*. This change has no effect on this research since only Java files are used, but it is mentioned to cover all changes made in Sherlock's source-code.

4.2.2 Problems with JPlag-java and Sherlock

Whilst using JPlag and Sherlock some problems were noticed in their functionality. This is not a big issue since the tools are treated as black boxes and no improvement is allowed. Also, the literature suggests that the tools are successful even with these problems, but it is for sure that these problems can help or hinder the tools in their work and therefore have some hidden impact on the experimental results. In this section, these problems are presented and the possible impact on the results of the experiment discussed.

The JPlag Java version has the problem of not supporting features of Java 8 and later, meaning JPlag-java just skipped these lines and did not parse them, although an in depth analysis of JPlag-java code was not done, so it might have other influences. But one should notice that if a correct parser for Java 8 would be implemented for JPlag-java, probably the results when Java 8 is used would be better. The problems with JPlag-java were:

- lambda expressions — which may cause errors on some lines after the lambda expression if they are closed on a new line like in the next example:

```
parking.forEach((p) -> {
List<MeteoData> meteoData = Arrays.asList(
    meteoClientBean.getMeteoData(p.getId(), p.getAdress()
    ));
meteoData.stream().forEach((mp) -> this.meteoDataList.
    add(new ParkingMeteoData(p, mp)));
```

```
}); //this causes the error to continue
```

- static imports ;
- annotations inside a function the argument definition for params with Java 8 ;
- lines that include the '::' operator like: *Arrays.stream(args).anyMatch("–kreni"::equals)*;
- annotations and functions that define the method with *@interface* and the default state for columns.

These problems with Java 8 are not a big concern because this issue was not present in most years (mainly for 2016/2017 (caused 284 errors) and 2017/2018 (caused 403 errors)). This number of errors might seem big but if we suppose that this represents the number of Lines of code (LOC) affected (although it is not a number of lines because sometimes one line causes up to 10 errors) and in 2016/2017 there were 4 assignments with 44 submissions and every submission had more than 800 LOC, this makes then 140,800 lines, which means the error appears in only 0.2% of the lines. This percentage is a big overestimate since most submissions have on average 1500 LOC which makes this percentage drop to 0.1%. The same percentage comes out for 2017/2018, even though there are more errors, since there were also more submissions. Another reason why this issue is not much of a problem for this research is that most students did not use the Java 8 features, so it is a small probability that plagiarism was present between two solutions that used the Java 8 features. Also, even if the Java 8 parts of code are not detected as similar there is a high probability, if this is a case of plagiarism, that the rest of the code is also plagiarised and this is then detected by JPlag-java.

In some rare cases, errors occur, because of the preprocessing technique which removed the *finally* statement and leaves only the *try* statement. This caused JPlag-java to report a problem that the *catch* statement is missing. Whether this has some effect on the similarity is unknown but since it happened rarely it is not investigated further. To be more concrete, the problem with missing *catch* statement occurred only 37 times in all the detections, when looking by year it was caused mostly in: 2016-2017 (10 errors) and 2017-2018 (10 errors). If the same estimation is done as before, supposing a submission has 800 LOC, this is then 0.007% of the LOC.

Similarly, it happened with another preprocessing technique where some leftover annotations or *catch* statements were present and sometimes an expression was partially deleted, like '*Message message = new Message(messId,.*', which was an expression in two lines and the second part was removed. But also this is not a concern since for all years and all assignments it happened only 20 times.

The biggest issue with JPlag-java is the problem with *import* statements or *package* statements which are not at the beginning of the file. This is caused by merging all files of a submission before the detection. This problem gives JPlag-java an advantage on the detections in cases where these lines are not removed previously by preprocessing techniques, in comparison

to the other tools which in some cases indicate some of these lines as similar. Such lines make up around 17% of the code, if calculated with 800 LOC per submission, but a more realistic approximation would be 9% since there are on average 1500 LOC in one submission. How big this impact is it is hard to tell since it is unknown if JPlag-java would have found these lines as similar if it didn't have the parsing problem. This issue should be taken into account if JPlag-java would perform much better than other tools in cases when such statements are not removed in preprocessing.

Sherlock has only one problem with the calculation of similarity whereby it may mark two files with a similarity larger than 100%. This problem is caused by the fact that Sherlock calculates the similarity for parts of code, called a match, and then some lines appear in two (or in some rare cases in more) matches. Sherlock then just sums up all the similarities as found matches. Since Sherlock is not using LOC as a basis for the calculation of similarity there is no way of correcting that without modifying the calculation algorithm. Since changing this would mean improving Sherlock it is not allowed, also as stated before Sherlock as is can be considered a successful tool even with this problem. But if this would be corrected Sherlock probably would report other similarities which then would change the configuration parameters during calibration and the detections would also report different similarities. Analysing this is out of the scope of this research but it is an interesting question for the future.

From the analysis, one can say that most of the mentioned issues probably do not have a big impact on the experimental results, but regardless of that one should keep them in mind when reading the final analysis of the experiment.

4.3 Configuration parameters of similarity detection tools

Most detection tools have configuration parameters which can be set before the detection. The idea is to give the user the possibility to configure how strict (or sensitive) a tool is when calculating similarity. By changing the configuration the tool one can go from considering everything as similar, to marking nothing as similar. To objectively compare tools they need to be executed with their best parameter configuration to eliminate the effect of the parameters. It would be biased to compare tools where one has the best parameter configuration and the other has the worst parameter configuration. The problem is that the best parameter configuration for each tool is not known and depends to some extent on the dataset used. Because of that, finding the best parameter configuration could take too much time or it might be impossible to find in some cases.

One way of solving this problem is to look at existing research and see what parameter configuration was used. Ragkhitwetsagul et al. [157] gave a useful overview of 30 tools and what was the best configuration in each case. But often configuration is not reported like in [26, 40], which brings into question whether the results are credible, but even when the configuration is known, as in [157], this does not guarantee that such configuration will be good for other

research. It can happen that on a different dataset another configuration is better.

The solution to the configuration problem is to perform a calibration of the tools using the calibration method which was developed as part of this research and presented in [138]. Instead of searching for the *best* parameter configuration for each tool, the idea is to find an *optimal* configuration which puts all tools in an equal position, as much as possible. This approach then ensures that the chosen configuration is not biased. Even though the calibration method is presented in [138] it does not give all details about the calibration of the selected tools which are used in the experiment, so more details are given in Section 4.4.

4.4 Calibration of similarity detection tools

When comparing tools it is expected that the similarities they report are as close as possible to the real similarity. For example, if there are two files which have 50% similarity it is expected that all tools report 50% similarity. In reality, often these similarities differ and since the real similarity is not known one can only relay the similarity that the tools report. If all tools report approximately the same similarity we can trust to a certain degree that this reflects the real similarity. However, if all tools differ, the problem is which one to trust.

In addition to this, as already stated, most tools have some configuration parameters which can be changed and this changes the similarity. Now the question is what parameter combination to choose to put them in a fair position for comparisons. Since the best parameter combination is not known, and since the real similarity is not known, the idea is to calibrate the tools. In the best case scenario, all tools will report the same similarity after calibration. In this way even though the real similarity is still unknown at least all tools are equally off from the real similarity.

The calibration method described in [138] uses the tools to calibrate each other using a Calibration Dataset (CD) and allows that the CD can be any dataset for which the similarities don't need to be known, which is a big benefit since the similarities vary from tool to tool and it is difficult or maybe impossible to establish what the exact similarity is. Another benefit is that the calibration can be done by one person without including experts and still have an objective calibration of the configuration parameters. It is important to note that this method, even though it puts tools in an equal position, is never 100% fair.

The primary component to enable such calibration is the metric called Calibration Difference Sum (CDS) described in detail in [138]. The formal specification of the CDS metric is [138]:

$$CDS_{ab}^{AB} = \sum_{i=1}^{n} |sim(c_i, t_a^A) - sim(c_i, t_b^B)|, a = p_{A1}...p_{An}, b = p_{B1}...p_{Bn} \tag{4.1}$$

where 'A' is the *base tool* used to calibrate tool 'B' (called the *calibrated tool*), 't_a^A' and 't_b^B' represent one configuration parameter combination of the tools, 'a' and 'b' are sets of allowed parameter values 'p' in the tools, and 'c_i' is one case in the CD representing a pair of files for which similarity is calculated. So, $sim(c_i, t_a^A)$ is the similarity of tool 'A' with configuration 'a'

on case '*i*'. The CDS is therefore a sum of absolute differences in similarity for each case in the calibration dataset for one configuration of tool A and tool B.

The idea of the method is to find a parameter combination where CDS is minimal. In an ideal situation CDS would be 0, which means all tools report exactly the same similarities. But one needs to be careful since it is possible to configure a tool to not find any similarity and always report 0%. Because of this, a CDS value of 0 is suspicious and one should always perform the calibration on a CD where at least for some cases the approximate similarity is known.

In this research mutual calibration is used, meaning each tool is used to calibrate the other, so the goal is to find the *optimal CDS* value (CDS_{opt}). Optimal CDS is achieved when calibrating tool B with tool A and calibrating tool A with tool B gives the minimal CDS for the same parameter configuration. Sometimes the optimal CDS value cannot be found, as happened when textual versions of SIM and JPlag were configured. In such situation one can use another tool to settle the argument, or look which configuration suggests the minimal CDS, or summing the CDS values in both directions and look which configuration has a minimal CDS total, or find some other way make a decision.

Since from three selected tools most information about parameter configuration exists for SIM and JPlag they were calibrated first. Once this was done Sherlock was calibrated with SIM and JPlag as base tools. The calibration dataset that was used consisted of 10 cases extracted from the SOurce COde Reuse (SOCO) training dataset (details about the dataset are presented in Section 6.2.) for which it was known if the case was plagiarised or not and 8 manually created cases (with different obfuscations) for which the similarity was approximately known to control the problem if CDS value would be 0. Each case had only two files which were compared. The description of the 8 manual cases can be found in [138].

To automatize the calibration, or rather to automatize the calculations for the CDS values for different configurations for each tool, a command line tool called MPC Calibrator was developed. MPC Calibrator is part of the MPC System and was designed to generate a textual report as shown in Appendix A. Input data for the MPC calibrator were: name of the base tool, configuration for the base tool, name of the configured tool and the path to a directory containing the calibration dataset. MPC Calibrator calculates CDS values for the given configuration of the base tool and the different configurations of the calibrated tool. In the report (Appendix A) the MPC Calibrator marks the optimal configuration and for this configuration it gives the corresponding similarities and absolute similarity differences between the base tool and the calibrated tool for all cases. Also, it states the best configuration and the corresponding similarity for a specific case. In the second part of the report, MPC Calibrator presents the total difference of similarities between the base tool and the calibrated tool (known as the CDS value) for each tested configuration of the calibrated tool.

The calibrated configuration is presented in Table 4.2. The detailed description of the individual calibrations is given in Section 4.4.2 and Section 4.4.3.

Table 4.2: Tools calibrated configuration

Tool Name	Java	Text
SIM-Grune	r=22	r=10
JPlag	t=9	t=7
Sherlock*	MFJ = 10, MRL = 13, MBJ = 1, MJD = 2, MSL = 1, STR = 1	MFJ = 10, MRL = 5, MBJ = 1, MJD = 1, MSL = 1, STR = 1

Note:
r - minimum run length of tokens
t - minimum token length
* - parameters explained in Section 4.4.3

4.4.1 Comparison of SIM and JPlag

SIM and JPlag have only one parameter that can be modified that influences the detection outcome. For SIM this is parameter *minimum run length of tokens (r))*, for JPlag this is parameter *minimum token length (t)*. In the article [157] the authors tested 30 similarity detection tools among which were SIM and JPlag. They found out the optimal parameters (in their scenario) for SIM and JPlag for both text and Java versions of the tools.

According to [157] the optimal parameter value for SIM is r=22 for the Java version and r=4 for the text version. For JPlag the optimal parameter value is t=3 for the Java version and t=8 for the text version. One could ask why these configuration parameters should not be used, and the answer is that, as explained before in Section 4.4, there is no guarantee that such configuration will be good for this research.

Before the actual calibration was done, a simple test was performed using the optimal configuration reported by [157]. It was expected that JPlag and SIM with their optimal parameters would give approximately equal similarities for each case. But this did not happen — in fact, there were big differences between similarities for some cases in the Java version (for example SOCO 8 case) and for some cases in the text version (for example SOCO 4 case) as shown in Table 4.3.

After qualitatively examining the results in the Java version it became clear that JPlag-java recognized non-similar parts as similar. For example, in SOCO 8 case (Table 4.4), one can see some lines (File 144 - A vs. File 192 - A and File 144 - B vs. File 192 - B) that JPlag-java matched as similar were at all similar (numbers 144 and 192 are the names of the files in the SOCO dataset). The reason why JPlag matched those lines is because the t value was too small and made JPlag very sensitive to small similarities. SIM-java did a better job of not identifying such lines. This is not a surprise since in [157] SIM-java ranked second while JPlag-java ranked twenty-first.

In the text version the situation was a bit more complicated. In SOCO 4 case, which is an exact copy with few slight changes, both SIM and JPlag in the text version gave worse results

Table 4.3: JPlag-java and SIM-java calibration dataset similarities

Case name	Similarity			
	Java version		Text version	
	JPlag (t=3)	SIM (r=22)	JPlag (t=8)	SIM (r=4)
manual - 0% example A	10.0	0.0	0.0	6.0
manual - 0% example B	11.4	0.0	0.0	2.3
manual - 50% Copy	64.1	58.5	40.7	56.5
manual - 50% Simple Obfuscation	67.5	67.0	48.8	64.0
manual - 50% Complex Obfuscation	64.1	58.0	39.9	57.0
manual - 100% Copy	100.0	100.0	99.7	100.0
manual - 100% Simple Obfuscation	98.3	100.0	96.7	100.0
manual - 100% Complex Obfuscation	84.8	92.5	79.1	98.5
SOCO 0 - N - (084-258)	45.7	14.5	0.0	4.7
SOCO 1 - N - (000-001)	22.1	7.5	4.1	30.6
SOCO 2 - N - (002-003)	28.9	0.0	4.0	10.2
SOCO 3 - P - (003-004)	77.6	54.0	49.6	61.5
SOCO 4 - P - (107-112)	100.0	100.0	33.3	75.0
SOCO 5 - N - (052-077)	50.0	0.0	4.2	17.5
SOCO 6 - N - (011-037)	38.5	0.0	0.0	3.0
SOCO 7 - P - (062-064)	87.4	85.5	77.7	85.5
SOCO 8 - N - (144-192)	57.1	1.0	0.0	15.3
SOCO 9 - N - (037-093)	41.4	0.0	0.0	12.0

Note:
N in the case name marks a non-plagiarised case
P in the case name marks a plagiarised case

than with the Java version. This is not unexpected since the Java versions are much better for source-code similarity detection than textual versions. But what is interesting is that JPlag-text tends to be more confused than SIM-text for SOCO 4 case. On the other hand, in the SOCO 1 case SIM-text identifies too much as similar because of the low r value, so only 1 line with 4 identical words is needed to be matched as equal. An example of one such matched line is in SOCO 1 case: "*BufferedReader bf = new BufferedReader(new InputStreamReader(is));*". But the similarity is purely coincidental through common code and usage of convenient variable names. Because of the SOCO 1 case JPlag-text does a better job not identifying the line as similar since it is in isolation.

Based on the results it is evident that both tools are more strict if the value of their configuration parameter is lowered and more liberal if the value is increased. Also, from the results, it is clear that the tools are not configured to be in a fair position for the experiment, so calibration must be performed.

Table 4.4: JPlag false similarity examples in SOCO 8 case

Example	Code

File 144-A	`if (imageFileName.charAt(0) != '/') {`
	` imageFileName.insert(0, URLName.concat("/"));`
	`}`
	`else {`
	` imageFileName.insert(0, HostName);`
	`}}`

File 192-A	`if (getFileSize(strWatchDogDiffFile_01_02) > 0)`
	`{`
	` sendMailWithDetectedChanges();`
	` System.out.println("Text␣diff␣mail␣has␣been␣sent␣the␣''`
	`␣email␣address.");`
	`}`
	`else`
	`{`
	` System.out.println("The␣DIFF␣file␣has␣zero␣length␣-␣`
	`text␣diff␣mail␣has␣NOT␣been␣sent.");`
	`}}`

File 144-B	`generateChecksum(temp0000File.getName(), checksumFile);`
	`}`
	`else if (!diffImages.isEmpty()){`
	`reportDifferences(false, null, null, diffImages);`
	`}`

File 192-B	` (inputFile.equals(strWebPageOutputFile01))`
	` {`
	`  writeTextArrayToFile(strImageArray,`
	`strImageOutputFile01,`
	`intImageCounter);`
	` }`
	` else if (inputFile.equals(strWebPageOutputFile02))`
	` {`
	`  writeTextArrayToFile(strImageArray,`
	`strImageOutputFile02,`
	`intImageCounter);`
	` }`
	`}`
	`}`

4.4.2 Calibration of SIM and JPlag

One could argue about which tool to choose to calibrate the other. Based on the results from Section 4.4.1, SIM-java (with r=22) could be selected as the base tool to configure JPlag-java, but since JPlag-java is the most widely used tool based on the literature review, it would make sense to select JPlag-java as a base tool even though it ranked worse in [157]. Because of this problem the decision taken to do a mutual calibration of SIM and JPlag so no tool is preferred.

The calibration was done separately for the textual versions and for the Java versions since the two versions use different approaches and have different success rates, as noted in Section 4.1. Also, to be able to analyse in the experiment the difference between the text and Java approaches they need to be kept independent from each other as much as possible.

Calibration of SIM and JPlag java version

The calibration of SIM-java and JPlag-java was done by calculating CDS values for all variations of parameters of SIM-java and JPlag-java in the range from 1 to 30. The range 1-30 was selected based on some simple tests on a few cases and observing how the parameter affects the similarities. It was noticed that after parameter values reached 30 the tools tended to find only obvious cases of plagiarism and mostly reported only 0% similarity.

As already explained, for each case in the calibration dataset the difference between similarities that SIM-java and JPlag-java reported were calculated. For example, SOCO 8 case in Table 4.3 SIM-java reported 1% similarity while JPlag-java reported 57.1% similarity this makes an absolute difference of 56.1%. Differences for all 18 case are then summed up and this represents the CDS value. For example, in the report presented in Appendix A the minimal value for JPlag-java was t=9 when calibrated with SIM-java (r=22) with the minimal CDS of 84.1.

Table 4.5 shows JPlag-java parameter ‘t’ in the range 1 to 15 and the corresponding optimal value of parameter ‘r’ from SIM-java which gives the lowest CDS value. The range from 16–30 for parameter ‘t’ is not presented since the CDS value is only increasing. SIM-java was searched for the optimal parameter value with minimum CDS in the range 1 to 30. From Table 4.5 it is visible that the lowest CDS with the value 84.1 is achieved with configuration: *JPlag-java t=9* and *SIM-java r=22*.

In Table 4.6 results are presented where SIM-java is the base tool to calibrate JPlag-java. The range for JPlag-java is 1 to 30 as it was for SIM-java. The only difference in Table 4.6 in comparison to Table 4.5 is that it shows the whole range for ‘r’ with some values excluded. Some parameter values are excluded because the parameter values of SIM-java give no new information and they follow nicely the trend of the presented data in Table 4.5. The data in Table 4.6 indicate that the best combination of parameters is r=22 and t=9 with lowest CDS value of 84.1, which is the same parameter combination as indicated in Table 4.5. It should be mentioned that theoretically it is possible that some untested combination with values above 30 may give

Table 4.5: SIM-java calibrated with JPlag-java as base tool [a]

JPlag-java (t)	SIM -java (r - best)	CDS
1	5	170.1
2	6	126.2
3	9	158.7
4	9	143.4
5	11	121.6
6	14	106.0
7	21	105.5
8	22	93.5
9	**22**	**84.1**
10	26	89.8
11	26	95.4
12	26	112.6
13	29	118.4
14	29	118.4
15	29	125.2

[a] Part of the table was presented in [138]

better results, but this is very unlikely and according to the noticed trend the CDS values are probably only growing.

With the configuration for SIM-java (r=22) and JPlag-java (t=9) the calibration of Sherlock-java is performed. This result can be considered good since it matches SIM-java parameter value 22 in the article [157] which indicates that the calibrated data is valid. But calibration shows that the reported value t=3 in [157] is not optimal when talking about fair comparisons and confirms the suspicion that there can be a better option for other datasets that have been reported elsewhere. Also, one needs to remember that SIM-java ranked second and JPlag-java ranked twenty-first so it was expected that the calibration would align more with the better ranked tool. Another important note is that these results and the results in [157] suggest that using the default configuration is not always good. For example, the SIM-java default configuration is r=24 which is different than the calibrated value and the value reported in [157], for JPlag-java, on the other hand, the default value is t=9 which is the same as value obtained by the calibration.

Table 4.6: JPlag-java calibrated with SIM-java as base tool [a]

SIM-java (r)	JPlag-java (t - best)	CDS
3	2	328.1
5	2	170.0
7	2	147.2
9	4	143.4
11	5	121.6
13	6	107.5
15	7	108.2
17	7	108.2
18	9	106.8
19	9	106.5
20	9	102.1
21	9	94.6
22	**9**	**84.1**
23	9	86.4
24	9	86.4
25	9	86.4
26	9	84.9
27	9	84.9
28	9	87.5
29	9	92.0

[a] Part of the table was presented in [138]

Calibration of SIM and JPlag text version

For calibrating the text versions of SIM and JPlag the same approach with the same range was used as in the calibration of the Java versions. Table 4.7 presents the CDS values for SIM-text as the base tool and Table 4.8 presents the CDS values for JPlag-text as the base tool. The main problem when calibrating text versions is that the CDS value constantly gets lower as the values for the 't' and 'r' parameters rise. From this it can be concluded that CDS would get to zero at a point when both parameters ('r' and 't') are so large that they would give for all cases zero similarity. Such parameters are of no use, so to solve the problem the SIM-java version was used to help stabilize the CDS values. The idea was to calibrate SIM-text using JPlag-text and SIM-java with r=22, and to calibrate JPlag-text using SIM-text and SIM-java with r=22.

In Table 4.7 and Table 4.8 there is a column showing 'CDS to SIM-java' which shows the CDS value of SIM-java (r=22) for the best corresponding parameter value of SIM-text (Table 4.7) or JPlag-text (Table 4.8). Since the idea was to keep the CDS between SIM-text and JPlag-text as close as possible, but also to eliminate the continuous decrease of CDS values with every increase of the parameter values, a CDS total was calculated summing up the CDS value to the base tool and the CDS value to SIM-java (r=22).

In Table 4.7 the smallest CDS total is 174.4 for SIM-text r=10 in combination with JPlag-text

Table 4.7: SIM-text calibrated with JPlag-text and SIM-java as base tools

JPlag-text (t)	SIM-text (r - best)	CDS	CDS to SIM-java (r=22)	CDS TOTAL
1	2	137.3	282.6	420.0
2	4	119.8	250.8	370.7
3	4	112.0	250.8	362.9
4	10	97.4	98.6	196.1
5	10	92.2	98.6	190.8
6	10	91.8	98.6	190.4
7	**10**	**75.8**	**98.6**	**174.4**
8	11	77.6	109.1	186.7
9	15	68.4	135.6	204.0
10	19	76.9	186.6	263.5
11	19	69.3	186.6	255.9
12	19	64.7	186.6	251.3
13	19	65.9	186.6	252.5
14	19	64.7	186.6	251.3
15	19	65.7	186.6	252.3
16	19	65.0	186.6	251.6
17	19	62.7	186.6	249.3
18	26	60.5	203.5	264.0
19	27	45.8	237.5	283.3
20	27	45.8	237.5	283.3

t=7. In Table 4.8 the smallest CDS total value is 247.8 for JPlag-text t=7 in combination with SIM-text r=10. Since in both directions the same parameter combination was suggested these were the values to be used in the experiment. Note that in Table (Table 4.7) the 'CDS to SIM-java' is not the smallest value, so if only SIM-java (r=22) would be used to calibrate the textual version of JPlag-text it would choose t=4, but this would mean a higher difference between SIM-text and JPlag-text. Similarly, with SIM-text the best value is r=7 when calibrated with SIM-java (r=22), which also has the same effect of increasing the difference between SIM-text and JPlag text.

With the configuration for *SIM-text (r=10)* and *JPlag-text (t=7)* the calibration of Sherlock-text was performed. The calibrated configuration can be considered good since the optimal value for JPlag-text in the article [157] is t=8 and which is just one value higher than calibrated. For SIM-text the difference is higher but this is not unexpected since SIM-text ranked last (rank thirty) and JPlag-text ranked fifth. As with the Java versions it was expected to more align with the better ranked tool. Regarding the default configuration, one can conclude the same as with Java version, that it is not a good choice to use the default values. For example, the default value for SIM-text is r=8 and for JPlag-text is t=5 which is different from the calibrated values and the values reported in [157].

Table 4.8: JPlag-text calibrated with SIM-text and SIM-java as base tools

SIM-text (r)	JPlag-text(t - best)	CDS	CDS to SIM-java (r=22)	CDS TOTAL
2	1	137.3	276.9	414.2
4	3	112.0	170.5	282.5
5	4	99.7	164.7	264.4
6	4	105.3	164.7	270.0
7	4	101.0	164.7	265.7
8	7	96.2	172.0	268.2
9	7	89.7	172.0	261.7
10	**7**	**75.8**	**172.0**	**247.8**
11	9	75.0	184.1	259.1
12	9	74.9	184.1	259.0
13	9	72.2	184.1	256.2
14	9	73.2	184.1	257.2
15	9	68.4	184.1	252.5
17	9	76.4	184.1	260.5
19	17	62.7	232.8	295.5
21	18	64.5	235.3	299.7
22	18	64.0	235.3	299.3
23	18	63.5	235.3	298.8
26	18	60.5	235.3	295.8
29	29	47.3	298.1	345.4

4.4.3 Calibration of Sherlock

The calibration of SIM and JPlag is easy in comparison to Sherlock, and the reason is that Sherlock has six parameters that can be configured. Let suppose that those parameter values range from 1 to 10 this gives 1,000,000 variations [12], so to calibrate Sherlock a limit needs to be placed on the number of parameter variations. The six parameters are [68]:

- Minimum String Length to Store (MSL): lines which have fewer characters than this number will be ignored;
- Minimum Run Length to Store (MRL): runs shorter than this number will be ignored;
- Maximum Forward Jump (MFJ): how far to look forward for a matching line when one pair of lines has matched;
- Maximum Backward Jump (MBJ): how far to look backward for a matching line when one pair of lines has matched;
- Maximum Jump Difference (MJD): how much the jumps in the two different files can vary in length;

[12] Statistically these are variations with repetition, but to simplify the discussion in this Section they are just called variations.

- Strictness (STR): how strict the algorithm is.

Sherlock has two more parameters which can have the values true or false. First is *Amalgamate Nearby Runs* which tells whether overlapping runs should be made into one bigger run. Second is *Concatenate Nearby Runs* which tells whether runs which are close together should be concatenated. In all experiments *Concatenate* was set to false, and *Amalgamate* was set to true. The default values of the six parameters that needed to be calibrated are presented in Table 4.9. Also, Sherlock can be used in Java mode or C++ mode, but since in this research only Java is used, only the Java mode in Sherlock was used.

Table 4.9: Sherlock default parameter values

Parameter	Value
MSL	8
MRL	6
MFJ	3
MBJ	1
MJD	3
STR	2

Calibration of Sherlock's Java version

Based on four manually created cases for which the similarity was approximately known, Sherlock's configuration parameters were tested for their effect on the similarity in four steps. The reason was to limit the number of parameter variations to a reasonable value of around 500 variations. The four steps were the following.

Step 1 Each parameter was tested alone in the value range from 1 to 20 while others were on default values (Table 4.9). The results were:

- MRL (interesting range [1-20]) - lowers the similarity with higher value;
- MFJ (interesting range [1-20]) - increases the similarity with higher value;
- MBJ (interesting value [1]) - made no changes in similarity;
- MJD (interesting range [2-4]) - values 1 and 2 gave the same similarity, values 4 and higher gave the same similarity;
- MSL (interesting range 1-2) - value 2 and higher gave the same similarity;
- STR (interesting range [2-3]) - values 1 and 2 gave the same similarity, values 3 and higher gave the same similarity.

Step 2 Since MRL and MFJ had the highest impact they were tested together in combination with others at the default value (Table 4.9). For both the range was from 1 to 20, which gives 400 variations. After analysing the results manually, the results were:

- MRL (interesting range [3-9,10,14]) - Similar similarities were for values [10-13] and [14-20]; MRL with value 1 or 2 is very small so it was decided to exclude them in this step;
- MFJ (interesting range [3,4,11,12,15,17,18,19]) - Similar similarities were for values [4-10] and [12-14], and equal similarities for values [15,16] and [19,20], while values 1 and 2 gave small similarities (20% or less) for cases where the similarity should have been approximately 50%.

Step 3 Step 2 left 9 MRL values and 8 MFJ values in a total of 72 variations. In this step, those two were combined with others in the range from 1 to 5 (since step 1 indicated that for the four parameters this covers all had interesting values) one at a time, and others were kept default (Table 4.9):

- MBJ (interesting value [1])- gave no changes, as already indicated in step 1, so it was fixed at 1 as it was in the default settings;
- MJD (interesting range [1-5])- there were slight differences in similarity over all five values;
- MSL (interesting range [1-2])- value 2 and higher gave the same similarity as indicated in step 1;
- STR (interesting range 1-2)- there were slight differences in similarity over all five values but strictness with value 3 or more gave low similarities (20% or less) on cases where the similarity was approximately 50%.

Step 4 Step 3 gives a total of 1440 variations, which is still a bit too many. Since most values contain MFJ and MRL they were further limited to 5 values. For MFJ the values [12,17,19] were removed since there was a 'representative' value close by. For MRL the values [4,6,8,10] were removed which left a nice range of 'representative' values. Once the best variations were known after the first calibration, the calibration was repeated with a new set of values including the removed ones.

Based on the results in the simple test the parameter values for the first calibration were (500 variations): MFJ - [3, 4, 11, 15, 18], MRL - [3, 5, 7, 9, 14], MBJ - [1], MJD - [1,2,3,4,5], MSL - [1,2], STR - [1,2]. The calibration was done on the same calibration dataset as it was done for SIM and JPlag. First Sherlock-java was calibrated with JPlag-java as the base tool and then with SIM-java as the base tool. The optimal configuration for base tool JPlag-java (t=9) with CDS=133,2 is: MFJ = 4, MRL = 7, MBJ = 1, MJD = 2, MSL = 2, STR = 1. The optimal configuration for base tool SIM(r=22) with CDS=164,5 is: MFJ = 11, MRL = 14, MBJ = 1, MJD = 2, MSL = 5, STR = 1.

Since the two configurations differed, the top six configurations from each calibration were taken and mapped together. These top six configurations from both calibrations are presented

Table 4.10: Sherlock-java's first calibration with SIM-java and JPlag-java

						CDS	CDS	CDS
MBJ	**MFJ**	**MRL**	**MSL**	**MJD**	**STR**	**SIM-java**	**JPlag-java**	**Total**
1	**11**	**7**	**2**	**1**	**1**	**184.5**	**140.0**	**324.5**
1	4	7	1	2	1	197.5	139.2	336.7
1	***4***	***7***	***2***	***2***	***1***	***205.5***	***133.2***	***338.7***
1	***11***	***14***	***2***	***5***	***1***	***164.5***	***182.6***	***347.1***
1	4	9	1	2	1	210.5	144.2	354.7
1	4	7	2	2	2	214.5	142.2	356.7
1	4	9	2	2	1	219.5	143.1	362.6
1	4	9	2	2	2	220.5	144.1	364.6
1	11	9	2	2	2	205.5	166.4	371.9
1	11	14	2	2	2	205.5	166.4	371.9
1	15	14	2	5	1	186.5	202.6	389.1

in Table 4.10, together with the CDS value between SIM-java and Sherlock-java (marked CDS SIM-java) and the CDS value between JPlag-java and Sherlock-java (marked CDS JPlag-java). Those two CDS values were summed up (marked CDS Total) which ensured that Sherlock-java had minimal differences from JPlag-java and SIM-java.

Now since some of the configuration values were excluded in Step 4, the idea was to repeat the calibration and see whether the CDS total can get even smaller. From Table 4.10 the parameter values from the top four combinations were taken for the second calibration since they included the top configurations got with SIM-java and JPlag-java. For MRL and MFJ parameters, the removed values from Step 4 were used for the second calibration, also values +/- 1 from the current top values were used if they were not included in the first calibration. This gives the following values for parameters (640 variations): MFJ [4, 5, 7, 10, 11, 12, 17, 19], MRL [4, 6, 7, 8, 10, 13, 14, 15], MBJ [1], MJD [1, 2, 5, 6, 7], MSL [1,2], STR [1].

In the second calibration, configuration with the minimal CDS=120.8 for base tool JPlag-java(t=9) is: MFJ = 10, MRL = 13, MBJ = 1, MJD = 2, MSL = 1, STR = 1. The configuration with the minimal CDS=146,5 for base tool SIM-java (r=22) is: MFJ = 11, MRL = 15, MBJ = 1, MJD = 5, MSL = 2, STR = 1. Since again the optimal Sherlock-java variations got with SIM-java and JPlag-java did not match, the top six variations were taken from each and mapped together as for the first calibration. The results are presented in Table 4.11. From this the chosen configuration parameter combination for Sherlock's Java version is therefore: *MFJ = 10, MRL = 13, MBJ = 1, MJD = 2, MSL = 1, STR = 1*.

The calibration could be repeated a few more times, but since the CDS total value did not decrease drastically in the second run there is reason to believe that further runs would not make much of a difference. Also, since the calibration is just a way to ensure an objective selection of parameters for the main experiment, the top configuration from the second calibration was not used.

Table 4.11: Sherlock-java's second calibration SIM-java and JPlag-java

						CDS	CDS	CDS
MBJ	**MFJ**	**MRL**	**MSL**	**MJD**	**STR**	**JPlag-java**	**SIM-java**	**Total**
1	**10**	**13**	**1**	**2**	**1**	**120.8**	**178.5**	**299.3**
1	10	15	2	5	1	153.6	150.5	304.1
1	***11***	***15***	***2***	***5***	***1***	***164.6***	***146.5***	***311.1***
1	7	15	2	6	1	162.5	152.5	315.0
1	7	13	2	6	1	162.5	152.5	315.0
1	7	14	2	6	1	162.5	152.5	315.0
1	4	6	2	2	1	129.3	190.5	319.8
1	10	13	2	5	1	165.5	158.5	324.0
1	10	14	2	5	1	165.5	158.5	324.0
1	10	14	1	2	1	133.8	191.5	325.3
1	7	6	2	1	1	134.4	192.5	326.9
1	4	7	2	2	1	133.2	205.5	338.7
1	4	8	2	2	1	133.2	205.5	338.7

Calibration of Sherlock's text version

The calibration of Sherlock-text was done in the same way as the calibration of Sherlock-java. Again, the four manually created cases were used to limit the parameter variations. But the difference was that for the textual version already at the first step very small changes of similarity were observed. In the first step, as for Sherlock-java, each parameter was tested alone in the value range from 1 to 20, while others were set to default values. Default values for the textual version were the same as for the Java version (Table 4.9). The results of step one were:

- MRL (interesting range [1,3,4,8,10])- lowers the similarity with higher value but with less effect than in Java version values; equal similarities were got for parameter values: [1,2], [4-7], [8,9], and [10-20];
- MFJ (interesting range [1,2,3,5,8,14])- increases the similarity with higher values but with less effect than in the Java version; equal similarities were obtained for parameter values: [3,4],[5-7],[8-13],[14-20];
- MBJ (interesting value [1]) - made no changes in similarity;
- MJD (interesting range [1,2]) - values 2 and higher gave the same similarities;
- MSL (interesting range [1,14,17]) - values [1-13],[14-16], and [17-20] gave the same similarities;
- STR (interesting range [1,3,7]) - values [1,2],[3-6], and [7-20] gave the same similarities.

In comparison to the Java version the number of variations was not too big (540 variations) so all could be run at once. But since MRL and MFJ made the most changes they were run

Table 4.12: Sherlock-text's calibration with SIM-text and JPlag-text

						CDS	CDS	CDS
MBJ	**MFJ**	**MRL**	**MSL**	**MJD**	**STR**	**SIM-text**	**JPlag-text**	**Total**
1	**10**	**5**	**1**	**1**	**1**	**160.9**	**149.9**	**310.8**
1	10	1	1	1	3	163.9	156.9	320.8
1	10	2	1	1	3	163.9	159.9	323.8
1	8	2	1	1	3	166.9	159.9	326.8
1	8	1	1	1	3	166.9	159.9	326.8

together in variations to see are there any other parameter values that would make a difference. Since no such value was found the ranges from step one were used.

Calibration was run on the calibration dataset as all the other tools with the ranges for parameter values: MFJ - [1,3,4,8,10], MRL - [1,2,3,5,8,14], MBJ - [1], MJD - [1,2], MSL - [1,14,17], STR - [1,3,7]. The results are presented in Table 4.12. CDS values for Sherlock-text were calibrated with SIM-text (r=10) are marked as *CDS SIM-text* and CDS values for Sherlock-text calibrated with JPlag-text (r=7) are marked as *CDS JPlag-text*. Again the sum is marked as *CDS Total* and the lowest CDS total was for the configuration: MFJ = 10, MRL = 5, MBJ = 1, MJD = 1, MSL = 1, STR = 1.

The most interesting thing in this calibration is that the same configuration was indicated by the calibration with SIM-text and JPlag-text. Since the configuration matched up, the configuration that will be used for all further experiments for Sherlock's text version is therefore: *MFJ = 10, MRL = 5, MBJ = 1, MJD = 1, MSL = 1, STR = 1*.

CHAPTER 5

EVALUATION MEASURES

In Chapter 4 it was mentioned that there are many tools available that can help find plagiarism in source-code programming assignments. In order to find out which tool to use, researchers compare the tools. Feature comparisons are a good way to make an initial selection, but to find out which tool is better experiments and qualitative analysis are performed. When doing experiments, various metrics are used to do the comparisons and the statistical analysis.

In this Chapter these metrics that are used for comparisons are analysed. To find out which metrics are used literature is reviewed in Section 5.1. After that the metrics which are used in this research are discussed in more detail in Section 5.2.

5.1 Related work

One of the first comparisons of algorithms used for plagiarism detection was done in 1989 by Parker and Hamblen and presented in "*Computer algorithms for plagiarism detection*"[147]. Soon after that in 1990 Whale published "*Identification of program similarity in large populations*"[185] comparing tools that had been developed at that time. Whale calculated the *Performance Index* and used it as a comparison metric, later on Verco and Wise used the same metric in [178]. More detailed explanation of the metric is given in Section 5.1.2.

The Performance Index metric depends on two measures called *Precision* and *Recall*. That this is an important fact becomes evident from Mozgovoy's article [131] where he declares that using Precision and Recall for evaluation is one of two main principles. The second principle that Mozgovoy describes is the evaluation based on *Highest False Match* and *Separation* (difference between the Lowest Correct Result and the Highest False Match) metrics used by Hoad and Zobel in [74].

The first approach using Precision and Recall is used more often than the second approach, in fact Precision and Recall are the two most used metrics for comparisons according to the SLR [140]. Precision was used 23 times for comparisons and Recall 24 times according to [140]. The next in line is *F-beta* which is a metric calculated based on Precision and Recall and was used 12 times. If one would count articles that use this metric for evaluation of their tool without comparison these numbers would be slightly higher.

As already stated in Chapter 4 in the performed SLR [140] from 120 developed tools only 55 were compared, and from those only 27 used a quantitative approach. For comparison, a qualitative approach was used 47 times. In the 27 articles that use a quantitative approach only 6 different measures are used, that are mentioned in at least two articles. Next to Precision, Recall and F-beta the other three measures are: *Performance Index* (used 2 times), *Sensitivity* (used

5 times), and *speed* (used 5 times). There are some other measures but conclusions in these articles are primarily based on one of the six mentioned measures.

From the six measures, speed is the least interesting since it is not a direct measure of quality. While speed is important it is not a primary concern. Most tools are able to produce results in a few minutes which will satisfy most users. Once it is known that a tool has good Precision and Recall does it make sense talking about speed as done in [143].

In order to explain the other five measures, one needs to establish some basic terminology. In Chapter 1 it was stated that *match* or *pair* represents two submissions which were compared. If a match is a case of real plagiarism it is referred to as *plagiarized match* and if a match is not plagiarised it referred to as *non-plagiarized match*. Tools mark matches as potentially plagiarized if the similarity of the match is higher than some *threshold* (sometimes also called *cut-off threshold*). Based on that, if a plagiarized match is also marked by a tool it is categorized as *true positive (tp)* otherwise it is categorized as *false negative (fn)*. If a non-plagiarized match is marked by a tool as potentially plagiarised it is categorized as *false positive (fp)* otherwise it is categorized as *true negative (tn)*.

Basically, every match can be classified into one of the four categories. Some combinations of these categories are known under specific names. Three terms for these combinations were already mentioned, but here they are explained in a different context: *plagiarized match* – which represents true positives and false negatives, *non-plagiarized match* – which represents false positives and true negatives and *match* – which covers all four categories. Except for those there is the term *indicated match* which represents true positives and false positives, or in other words everything that a tool marks as potential plagiarism.

Different measures, simply formulated, present different ratios of these categories using the number of elements in some category. Measures that will be used in this research (Section 5.2) are Precision, Recall and F-beta (which is calculated based on Precision and Recall) since they have been used in various existing studies [31, 134, 154, 157, 170].

5.1.1 Sensitivity

Sensitivity can mean three things, first as stated in [2] it can mean Recall, and the term Sensitivity for Recall was used by Lee at al. 2012 [102]. But Sensitivity mentioned previously actually represents two different measures and not Recall.

First, Sensitivity (further referred to as Sensitivity-A) as used in [63, 98, 130] presents how the system will react if more and more obfuscation is done. Example of that was done in [63] where one program is put through a series of refactoring modifications (simulating obfuscation methods) and if it gives 100% similarity then it is insensitive to modifications. If the modifications lower the similarity of a tool the tool, is considered sensitive, and the more modifications lower the similarity the more sensitive a tool is. The idea is to modify the program bit by bit and see how the system becomes more sensitive. For example in [98] lines of code are inserted to see how the similarity will lower. Sensitivity-A is easier presented with a line graph

where on the y axis is the similarity and on the x axis the obfuscations starting from the simplest to the most complicated. The line in the graph usually starts at 100% similarity for the original program (at least that is expected) and then lowers towards 0% similarity.

Second, Sensitivity that is used by [185] and [178] where it is explained in detail, is further referred to as Sensitivity-B. To explain Sensitivity-B, even in short, the terms *redundant matches* and *essential matches* are needed. Redundant matches are matches that already have another match where the two submissions exist. So if submissions A, B and C exist and there are three matches A-B, B-C, A-C then one of them is redundant and the others are essential matches. The redundant match is the one that has the lowest similarity. Sensitivity-B, as explained in [185], tells us how many of the possible redundant matches are found in a group of n submissions (calculated by $0.5 * (n-1) * (n-2)$). Sensitivity-B increases if the threshold increases. Since Sensitivity-B can be modified by changing the threshold it is normally not used directly, instead, it is used to calculate Performance Index at a constant value of Sensitivity-B. There is also the term *Limiting Sensitivity* which represents the Sensitivity-B that will include at least all true positive essential matches, at this point there is little sense to change Sensitivity-B since Precision is the maximum it can be.

One can see that the term Sensitivity in both cases means the same thing, the only difference is how it is measured. In general Sensitivity means if a system is more sensitive it introduces more matches that are not plagiarized matches.

5.1.2 Performance Index

The Performance Index metric depends on three measures: Sensitivity-B (explained in 5.1.1), Selectivity and Excess Detection. All are introduced by Whale in [185] and were used by [178] where detailed descriptions can be found and how to calculate them. Sensitivity-B was already explained in the previous Section so here is a short explanation of the other two dependent measures [178, 185]:

- Selectivity – is the ability of a plagiarism detection tool to have high Precision with increasing Sensitivity-B. Zero Selectivity is present when true positive essential matches are equally spread throughout the Sensitivity-B range. The Precision at such a linear response is always constant. Since the true number of plagiarized matches is not known this Precision is calculated with estimated parameters. Precision of the linear response (P_L) based on estimated parameters is calculated as a number of plagiarized essential matches found by all systems compared divided by half the number of submissions (N)), Although in [178] authors use N number instead of $N/2$. The formula to calculate P_L is: $P_L = (tp + fn)/(N/2)$

- Excess-detections (D) - "*are detections made by a real system over an idealized system with maximum Recall and zero Selectivity.*"[185] "*It represents the Excess Detection made by the detector over pure chance.*" [178] Maximum D is needed to calculate the Per-

formance Index and it is measured only when Precision is high enough, usually 60%. To calculate D the theoretical number of true positive essential matches of the linear response is subtracted from the measured number of true positive essential matches at the same number of essential matches (esm). This is done for all values of essential matches from 1 to n where Precision is above 60% (or 0.6). Then the maximum D needs to be found to calculate Performance Index. The formula to find maximum excess-detections is: $D_{max} = \max_{esm=1}^{n}(tp_{@esm} - esm * P_L)$ where: $tp@esm$ are the true positives essential matches at the number of essential matches, esm is the number of essential matches, P_L is the precision of the linear response and n is the number of checked essential matches usually estimated as the number of submissions divided by two according to [185].

Now that it has been explained what excess-detections are, one can define Performance Index. Performance Index is the normalized maximum Excess Detection. It is calculated by multiplying maximum Excess Detection by the normalization value (also called normalization factor). The normalization value is the reciprocal value of the number of real plagiarized matches multiplied by $1 - P_L$. The Performance Index is calculated as

$$Performance_index = D_{max} * \frac{1}{(tp + fn) * (1 - P_L)} \tag{5.1}$$

or

$$Performance_Index = \max_{esm=1}^{n}\left(tp_{@esm} - esm * \frac{tp + fn}{\frac{N}{2}}\right) * \frac{1}{(tp + fn) * (1 - \frac{tp+fn}{\frac{N}{2}})} \tag{5.2}$$

where: N is number of submissions; tp – true positive essential matches, fn – false negative essential matches; esm is number of essential matches; $tp@esm$ – true positives essential matches at the number of essential matches; n is the number of checked essential matches usually estimated as $n = N/2$.

Usually, in a real case scenario, the number of real plagiarised matches is not known so the fn value is actually an estimated value, mostly consisting of marked real plagiarised matches by multiple tools. While Performance Index seems to be a very good measure for comparison it is difficult for calculation since it uses only essential matches for calculation. Probably because of this reason today a more acceptable measure is F-beta.

5.2 Precision, Recall and F-beta

In an ideal situation, everything that a tool marks as plagiarized is really plagiarized and all plagiarized cases are marked. But this is in most situations impossible, so two measures called Precision (P) and Recall (R) are used to describe how good a tool is.

Precision and Recall are two measures that are very dependent ranging from 0 to 1. Usually one can not increase one without decreasing the other. Recall shows how many plagiarised cases a tool has marked, while Precision shows from the matches that are marked as potentially

plagiarized how many matches are really plagiarized. Because the tools are not perfect, to find more plagiarized matches (higher Recall) one will also have more false positive matches (lower Precision). Precision and Recall are calculated as follows:

$$P = \frac{tp}{tp+fp}, \in [0,1] \quad R = \frac{tp}{tp+fn}, \in [0,1] \tag{5.3}$$

Since the real plagiarized matches need to be known, as stated before, the number of false negatives is usually estimated. This has the implication that Recall is also an estimated value. To estimate the false negatives in this research the number of real plagiarized matches is a union of real plagiarized matches that were marked by every tested tool, the same approach was used by Verco and Wise in [178]. In this way, if there is a plagiarized match that is not found by any tool it has the same impact on all tools. Meaning, if an extra false negative would be added it would be added to all calculations for every tool, this would change the Recall, but the ratio would still be the same and that is more important when doing comparisons.

To be able to calculate Precision and Recall one needs to set the cut-off threshold. The threshold can be *fixed* or *optimal* as stated in [154]. The fixed threshold is chosen in advance and it is the same for all tools and datasets. The fixed threshold is usually set to a certain percentage as done in [154] or it can be set to the first n matches as explained in [63]. The optimal cut-off threshold is a threshold where the sum of Precision and Recall is maximal, whereby one sets the importance of Recall in comparison to Precision. This can be represented [130] as $P + mR$ where m is the importance of Recall. If m is less then 1 it gives more importance to Precision and if m is bigger than 1 it gives more importance to Recall.

Sometimes interesting information can be obtained when presenting Precision and Recall in a graph form. For example, in [154] graphs presenting Precision or Recall at different thresholds show how fast the two values change with a lower threshold. With a lower threshold Precision decreases but the Recall increases. Similar information can be obtained with a graph presenting Precision at different levels of Recall as used in [5, 31]. Such a graph shows how Precision lowers as the Recall increases. An overview of other graphs and other measures that can be used for comparisons is presented in [140].

Based on Precision and Recall the measure *F-beta* can be calculated. F-beta (more formal representation is F_β) is a unified number of quality taking into account both Precision and Recall. F-beta is calculated as follows:

$$F_\beta = \frac{(\beta^2+1) * P * R}{(\beta^2 * P) + R}, \in [0,1] \tag{5.4}$$

The beta value has the same function as the m value in the optimal cut-off threshold. If beta is less then 1 (usually 0.5) it gives more importance to Precision and if beta is bigger than 1 (usually 2) it gives more importance to Recall. In most cases beta is set to 1 so it gives equal importance to Precision and Recall, this measure is called *F1 measure* (or F1 score or balanced

F-score). F1 is the harmonic mean of Precision and Recall. To calculate F1 measure the formula can be simplified to:

$$F_1 = 2 * \frac{P * R}{P + R}, \in [0, 1] \tag{5.5}$$

In this research, the term *accuracy* is used to describe quantitatively and qualitatively how well a tool performs. Quantitatively it is represented using the average measure F1 or the individual measures Precision and Recall. Note that the term *accuracy* can also mean a metric, as presented in [2], which tells us the rate number of true positives and false negatives divided by the number of matches.

CHAPTER 6

EXPERIMENTAL DATASETS

In [139] it is noted that often results from different studies are not completely comparable, even when the same metrics are used, because these studies use different datasets. Similarly, it was noted in [100] where a quote from Saxon was used [162]: "*it is difficult to compare detection engines in the same way since there are no standard source code corpora available*".

In this Chapter a literature review was performed to determine whether there are some standardized datasets which could be used for comparison of detection tools. In Section 6.1 the results of the literature review are presented, and in the rest of the sections the description of datasets that will be used in this research are given.

6.1 Related work

In the literature there seems to be an agreement that the F1 measure together with Precision and Recall is a standard measure to be used for detection tool comparisons. On the other hand, the same research does not use a standardized dataset. From the 24 articles that use Recall as a comparisons metric, in 19 of them, datasets were used that consist of personal students' datasets or datasets that were specifically generated for that research. Using personal datasets is important, especially when these datasets consists of real student solutions. For example, using real personal students' datasets has the advantage of showing how tools operate in a real environment, and since most of these studies use JPlag for comparisons these studies show how JPlag performs on different datasets.

The problem with using personal datasets in research is that results for tools which were used in only one study are not directly comparable to other studies. For example, in [38] a tool called source code similarity detector system (SCSDS) was developed and compared to JPlag. In the execution, the F1 measure was used with a personal dataset. The study showed how SCSDS performed better on different levels of cut-off thresholds than JPlag. The research itself is relevant, well executed, and published in a credible journal. The only problem is that, since a personal dataset was used, a direct comparison of SCSDS to tools reported in other studies is not possible, especially since SCSDS is not a publicly available tool.

To have the possibility of direct comparisons, standardized datasets need to be created and used in the studies. This does not mean that personal datasets containing real data should not be used, rather it means that standardized datasets should be used along with personal datasets. Until today according to the Systematic Literature Review (SLR) performed in [140] only two datasets were used at least two times in 150 analysed articles. To visualize, datasets types that are discovered in [140] are presented in Figure 6.1.

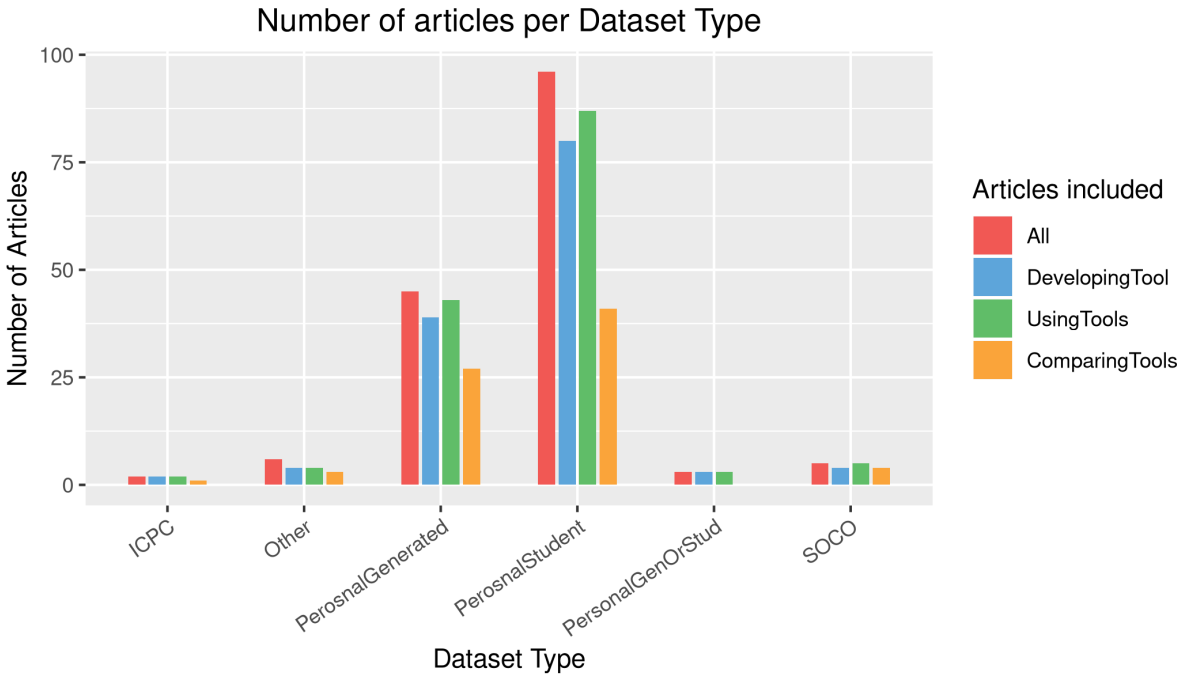


Figure 6.1: Number of articles in databases per year

There are six dataset types in studies as presented in [140], three types are personal dataset types and three are standardized dataset types. The personal dataset types are: datasets that use real student data (PersonalStudent), datasets that were manually generated for the purpose of the study (PersonalGenerated) and datasets in studies that use both real and generated datasets (PersonalGenOrStud). There are 6 standardized datasets but only two are used more than 2 times in the analysed articles (more details later in this Section), the other three are grouped into one category (Other) and are the least interesting. Figure 6.1 divides the number of articles per dataset type into four areas, which include articles that: 1) compare tools, 2) develop a new tool, 3) use a tool in their study which includes comparison and development of tools, and 4) all articles that were analysed in the SLR. Figure 6.1 clearly shows how personal datasets are used mostly in comparison to the standardized datasets in all four areas.

These standardized datasets are the SOurce COde Reuse (SOCO) dataset [1] and the ICPC (ACM International Collegiate Programming Contest) dataset [2]. The SOCO dataset was used 5 times in [35, 40, 52, 58, 164] and the ICPC dataset was used 2 times in [79, 102]. In articles that do comparisons, SOCO was used 4 times and ICPC only once. Of course, if one looks beyond the 150 articles there are other studies that also use these datasets like [71] or [157] as reported on their web page [3].

While both datasets are available to the public the SOCO dataset is more practical for detection tool comparisons in the domain of source-code plagiarism, and the reason is that SOCO

[1] `http://users.dsic.upv.es/grupos/nle/soco/`
[2] `https://icpc.baylor.edu/worldfnals/problems`
[3] `http://crest.cs.ucl.ac.uk/resources/cloplag/`

has reported the real plagiarised matches which are necessary to calculate Precision and Recall. There are also other datasets that are available and are used in the source-code plagiarism domain like the Google Code Jam Contest [4] in [50] or the BlueJ BlackBox dataset [5] in [125].

While these datasets are useful in some cases, for example in [125] authors report that the BlueJ BlackBox dataset is suitable for applying code style based plagiarism detection techniques, there is still the problem of knowing which are the real plagiarised cases. Because of that issue, and based on the usage of the dataset in the source-code plagiarism domain, one can conclude that the SOCO dataset is currently the dataset that has the best probability of becoming a standardized dataset to be used in detection tool comparisons in the source-code plagiarism domain. More details about the SOCO dataset are given Section 6.2. If one is interested in cross language plagiarism detection there is a different dataset called CL-SOCO [6] which is described in detail in [49].

6.2 Source Code Reuse dataset

The SOCO dataset has been created for the international PAN@FIRE 2014 competition, which is a competition that "*focuses on the detection of source codes that have been re-used in a monolingual context*"[52]. The dataset is first divided into Java and C collections. Since in this research Java is used when referring to the SOCO dataset, only the Java collection is used. The Java collection is further divided into a *training collection* and a *test collection*. Both collections have reported the real plagiarised cases [7] but there is a difference how they were labelled. The training collection was labelled manually by three experts and the test collection was labelled using tools whereby a "*pair needs to appear at least in 66% of the competition runs to be considered as a relevant judgement*" as reported in [52].

In Table 6.1 the structure of the SOCO dataset is given. The SOCO training collection was made from the corpus used in [5]. The training collection consists "*of university student assignments that contain source code re-use cases that were manually detected and reported by the professors. [51].*"

The test collection has been created from the Google Code Jam Contest source-codes (same collection as used in [50]) and marked automatically as explained before. The SOCO test collection is divided into 6 assignments containing different files. The main difference between the assignments is that they represent different scenarios (i.e., different themes/problems are considered) [52], other than that the differences between the assignments are the average line counts and the differences in the number of reused files and the total number of files.

The A assignments are solutions from Problem B (Dancing With the Googlers), the B

[4] `https://code.google.com/codejam/contest/1460488/dashboard`

[5] `https://bluej.org/blackbox/`

[6] `http://users.dsic.upv.es/grupos/nle/clsoco/`

[7] In the context of SOCO they call it re-use rather than plagiarism since the decision whether it is plagiarism depends on how much re-use is acceptable.

Table 6.1: SOCO dataset structure

Collection	Assignment	Number of Files	Plagiarized files	Plagiarised matches	Matches
Test	A1	3,241	86	54	5,250,420
Test	A2	3,093	75	47	4,781,778
Test	B1	3,268	124	73	5,338,278
Test	B2	2,266	62	34	2,566,245
Test	C1	124	0	0	7,626
Test	C2	88	28	14	3,828
Training	T1	259	115	84	33,411

assignments are solutions from Problem C (Recycled Numbers) and the C assignments are solutions from Problem D (Hall of Mirrors). Descriptions of the problems are available at the Google Code Jam Contest website mentioned before. "*Problems B and C are considered of medium difficulty, whereas problem D is the most difficult.*"[50] In the Google Code Jam Contest the problems B, C and D have two complexity levels — small and large — as noted in [50]. While it is not clearly stated in [52] it is suspected that the distinction between A1-A2, B1-B2, and C1-C2, represents the solutions submitted to solve the smaller or larger complexity of the corresponding problem the same as was done in [50]. If this is true then the A1, A2, and B1 problems had an 89–95% success rate (the number of correct solutions divided by the number of submitted solutions) and the B2, C1, and C2 problems had only 63–71% success rate as stated in [50].

In this research, the most interesting is the training collection (assignment T1) which has been manually labelled by experts so it was selected to be used in the experiment. Since in this research six tools are used in combination with preprocessing techniques it was decided to use the C1 and C2 assignments from the test collection "as is" and to create subsets of the other four assignments in the test collection for the analysis. The reason why only C assignments are used as provided is that the number of files in the A and B assignments is large and would take too long to process, and the number of files in a group is much larger in comparison to the dataset containing real student solutions. Additionally, the C1 assignment has no plagiarized matches which makes it impossible to calculate Precision and Recall so for C1 only a qualitative analysis can be performed. In summary, for the research assignments, T1, C1 and C2 are used "as is", where C2 is not part of the experiment that is verified using F1 measure and the A1, A2, B1 and B2 are used partially. More details about how the datasets are used are given in Chapter 8.

6.3 Real student solutions datasets

Why it is important to use datasets containing real data is evident from SLR [140], where from 150 papers 96 papers use in their studies real personal datasets, and that from 55 papers that do comparisons 41 are using real personal datasets. Most research projects working with

such datasets have created the datasets from solutions of one course at an undergraduate level according to [140]. Also, most research has on average up to 6 assignments, from 50 to 300 submissions per assignment and with solutions of 50 to 500 Lines of code (LOC). There are some articles that have a large number of submission [20, 36], or assignments [190] gathered from multiple years, but they are rare.

In this research, a Real Student Solution (RSS) dataset was created which contains submissions gathered in one course over six years. The course is teaching web programming using the programming language Java on the first year of a university graduate study program. In all six years, students have to submit four assignments which differ in complexity. While the first two assignments are relatively simple by using more basic Java, the last two assignments are more complex and use all sorts of features like web services, JSF, ORM, web filters, EJB and similar. Submissions can contain, in addition to Java source files, other files types such as pictures, text documents, XML files, PDF files, and SQL files. Also, submissions supporting different languages like HMTL, CSS, JavaScript, etc. are possible. In Table 6.2 the structure of the RSS dataset is presented. On average there are 50 submissions per one assignment and each submission ranges from 400 LOC to 6,000 LOC which makes 11.8KB file size up to 155KB file size.

In comparison to a typical study, this study has a larger dataset than most other studies, but since only the first assignment is verified by the experts and therefore only for the first assignment can the calculations be made, can one say that this study has the average number of assignments (6 assignments, one in each year). Since it was performed at the graduate level (which is relatively rare) it has larger solutions which mostly have more than 500 LOC. It has the average submission size of 50 submissions for one assignment and like most studies, it is performed on one course. In summary, this research is in some sense performed on a larger scale of real data than most of the previous studies.

Table 6.2: Real Student Solution dataset structure

	Assignment							
	A1		**A2**		**A3**		**A4**	
Year	**T**	**P**	**T**	**P**	**T**	**P**	**T**	**P**
2012-2013	61	8	59	NA	60	NA	61	NA
2013-2014	47	2	47	NA	49	NA	50	NA
2014-2015	50	9	52	NA	49	NA	47	NA
2015-2016	41	0	43	NA	43	NA	42	NA
2016-2017	44	0	44	NA	45	NA	44	NA
2017-2018	56	4	58	NA	58	NA	55	NA

Note:
T - Total number of submissions
P - Plagiarised number of submissions

The evaluation (described in Section 6.3.1) of which matches are cases of real plagiarism was done by an expert, more specifically the teacher who is responsible for the course. The rule to decide what is considered plagiarism (according to the definition in Chapter 2) was simple, if the teacher is ready to accuse the student of plagiarism it is considered a real case of plagiarism. Note that students who have been accused of plagiarism by this teacher have always admitted that this is the case. Note that in Table 6.2 for A2, A3 and A4 the number of plagiarised submissions is marked NA. This is because the number is not known. Because of the large number of manual checks that need to be made by the expert it was not possible to include all four assignments in this research and therefore only the first assignment A1 is used and the others are left out for future work.

In the course there is a clear statement that everything that is reused that was not given by the teacher is restricted. If a student takes something from the web or other students he/she can report it by filling out a word form which he needs to submit together with the solution. In this case when it is reported this part is not considered plagiarism. Students are also notified it makes no difference to the teacher if the student copies something from the web (plagiarism) or from another student (collusion). If a student gives their solution to another student they are punished the same as the student who copied, meaning their work is also considered plagiarism.

6.3.1 Procedure for analysing student solutions

A problem with analysing matches manually is that there are many of them. In this research on average there were approximately 1,800 matches per assignment in one year. One option would be to look at the top n matches, but this could lead to missing some if n is small, and maybe too much to analyse if n is big, also there is the problem to decide what number n should be. Another possibility is to analyse matches above a certain percentage, but this again leads to a high number of matches since the tools in combination with different techniques report quite different similarities.

The solution to this problem is the idea that it is not important how high the similarity of a match is, but how it is in relation to the rest of the matches. The same idea was used to solve the problem of selecting the threshold level needed for the calculation of F-beta, presented in Section 3.2.3. With this idea in mind, the matches were ranked by similarity and then marked in three colours. First are matches with a similarity that is 3*inter-quartiles (IQR) from the median, these are considered critical and must be analysed and are marked red. Second are matches with a similarity between 3*IQR and 2.5*IQR from the median, marked orange. Third are matches between 2.5*IQR and 2*IQR from the median, marked yellow. The procedure is presented in Figure 6.2.

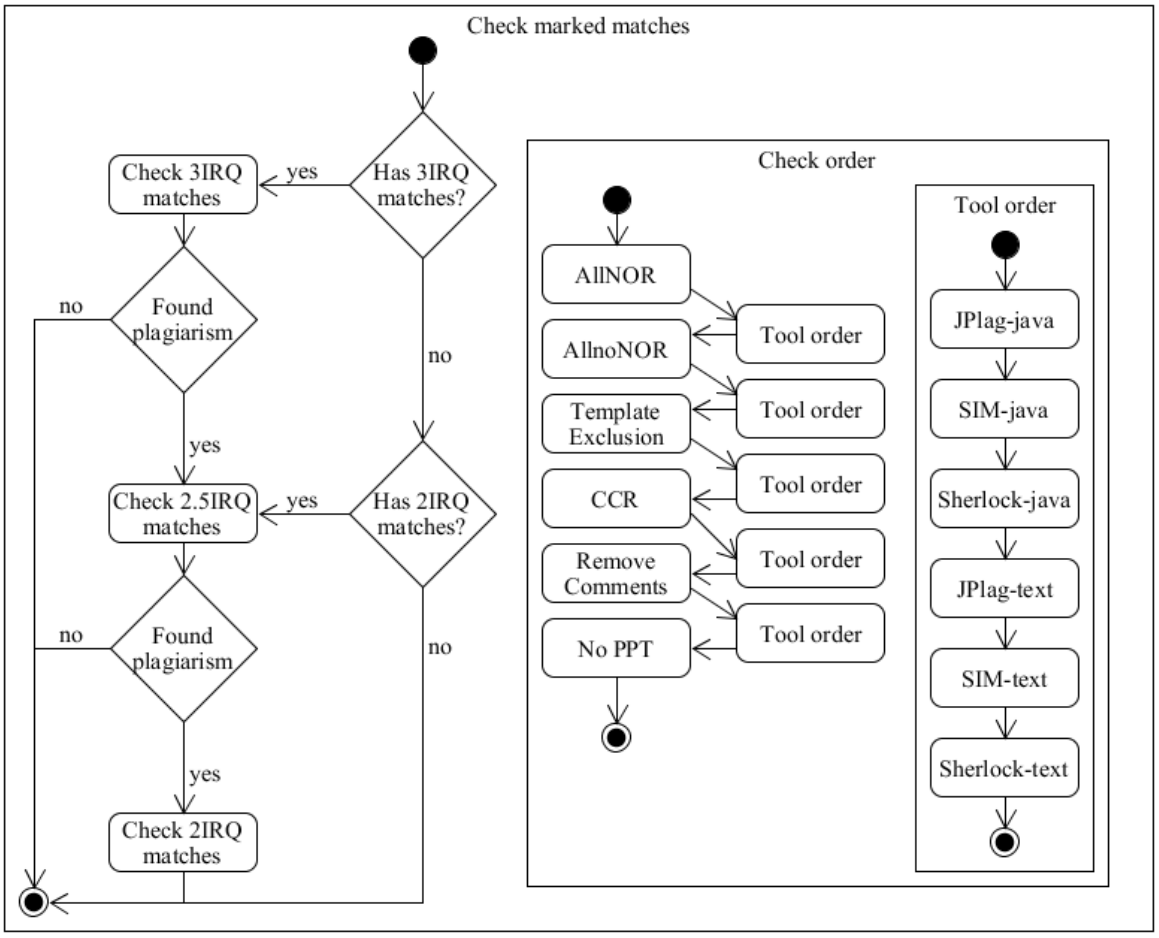


Figure 6.2: Procedure for confirming plagiarised matches

CHAPTER 7

PREPROCESSING TECHNIQUES

The main goal of this research is to analyse the effect of preprocessing so it is logical that preprocessing techniques are one of the main components for the experiment. In Section 7.1 previous researches that were done is analysed and discussed using Systematic Literature Review (SLR). The main reason for the SLR was to answer the question: "*Which are the most used preprocessing techniques?*". Answering this question is important to find out what techniques exist and to filter out the ones that are considered useful. After the review of related work in this Chapter the selected techniques are described together with a simple test that was performed to decide which techniques to use in the experiment.

7.1 Related work

As already noted in Chapter 1 researchers try constantly to improve the detection quality, and one way of doing so is to use the PreProcessing Techniques (PPTs). Sometimes the preprocessing techniques are called a different name for example in [159] they are called the *Transformation Rules* but the idea is the same as in preprocessing techniques. An indication that preprocessing is important is available in [96] where it was concluded that "*pre-processing may be more important than the comparison algorithm itself*". In [96] it was stated that the goal of preprocessing is "*to reduce the 'noise' in a given set of programs*" and it was indicated that their system which uses preprocessing can give better results in comparison to other systems.

To analyse which preprocessing techniques exist the same articles will be analysed that were used in the SLR [140]. From 150 processed articles there are 42 articles [2, 3, 6, 12, 14, 24, 26, 31, 32, 38, 40, 50, 59, 64, 67, 76, 82–85, 88, 90, 95, 97, 99, 104, 105, 107, 111, 112, 120, 136, 137, 143, 147, 154, 159, 174, 187, 188, 198, 200] describing some kind of preprocessing technique. In Table 7.1 the identified categories of preprocessing techniques are presented with the count of how many articles where mentioning them. PPT categories listed unify different techniques under the same name, this is especially true for the *Change Code Parts* and *Reorder Code* categories. These categories can be seen as one bigger technique, so further the categories are refereed to as technique.

PPTs can be divided into two categories according to how they can be used [139]: "*independent preprocessing – wherein results can be used with different tools or algorithms like removal of comments, and dependent preprocessing – wherein results can be used only on some tools like tokenization. Dependent preprocessing could be considered as a part of a new tool or algorithm.*" In this research the focus is on the *independent techniques*. The techniques listed in Table 7.1 are all considered independent, but this must not mean that the individual articles that mention

Table 7.1: Mentioned preprocessing techniques in reviewed articles

Preprocessing Technique	Article count
Remove comments	28
Remove white spaces	20
Remove code parts	13
Change code parts	11
Template exclusion	10
Reorder code	6
String to Upper/Lover case	6
Other PP techniques	3

some technique use them as independent. It is possible that a PPT was used as part of some algorithm, but it would be possible to extract or reimplement the technique to run it separately and combine it with other plagiarism detection tools. On the other hand, the tokenisation technique is a very specific one, even though it can be extracted and run separately, it is seen as dependent processing. This is because "*every tool has its own variation of tokenisation. As a result, it is not possible to use the tokenisation process results obtained by one tool in another tool.*" [139]

To make it clear what each PPT in table 7.1 represents here is a short description of each one:

- *Remove Comments (RC)* - PPT which will remove comments from the code. This includes: single line comments, block comments, trailing comments and end-of-line comments. This PPT is mentioned in [2, 12, 32, 38, 59, 67, 76, 82–85, 88, 90, 95, 105, 107, 111, 120, 136, 137, 143, 147, 154, 174, 187, 188, 198, 200].

- *Remove white spaces (RWS)* - PPT includes removal of redundant white spaces. This could be a new line, new line with carriage return or just multiple sequence of space or similar chars. This PPT is mentioned in [12, 50, 59, 67, 82, 83, 85, 90, 95, 97, 99, 105, 107, 120, 136, 137, 143, 147, 154, 200].

- *Remove Code Parts* - PPT which removes some code parts that are not needed for the logic, that are complicating the detection since the code part is too generic, or that in some other way obfuscates the detection. The articles that are mentioning some kind of a PPT that is removing code parts are [2, 31, 32, 38, 76, 88, 90, 97, 105, 112, 120, 187, 188]. Some examples of the techniques that are mentioned in this articles are: removing all tokens that are not from the lexicon of the target language [188], removing package declarations and import statements from the original source code [38], remove access specifiers (public, private, protected) [38], remove all declarations of variables or functions [97], removing the input and output statements and header file on the premise of not affecting the program semantics [120].

- *Change Code Parts* - PPT which changes the code to improve the detection. The techniques that are counted here are mentioned in [12, 26, 32, 38, 50, 97, 107, 111, 120, 159, 174]. Change code parts PPT includes: removal of symbols that are duplicated more than six times [50], extract reserved words and user definition symbols [174], stripping variable names and formatting [111], selection and iteration statements are transformed to standard forms [159], replace all variable names with a constant name and replace all function names with a constant name [97], renaming variables [26], unifying programming style [120], extract reserved words and user definition symbols [174], treat identifiers which consist of multiple terms separated by underscores as single terms [32], applying the same indentation style [12], normalize the statements and function names in the source code [107], and split of combined variable declarations into a sequence of individual declarations [38].

- *Template Exclusion (TE)* - or sometimes referred to as removal of professors code is a PPT which tries to remove all code that is given to students and which usage should not be considered plagiarism. This technique is mentioned in [3, 24, 32, 38, 40, 64, 95, 111, 112, 154].

- *Reorder Code* - PPT which performs code reorder to improve the detection. The techniques that are counted here are mentioned in [38, 97, 107, 120, 137, 188]. Reorder code PPT includes: reordering the functions into their calling order [188], ignoring order of the fields [137], divide program code into parts where one part is one function [97], unifying programming style [120], changing the order of variables in statements, according to the alphabetical order [38], and sort the user-defined functions so that the entry functions (e.g. main) are first, then the rest of the functions in calling ordered and last all functions that have never been called [107].

- *String to Upper/Lover Case* - a simple technique to change the whole code or parts of code from upper case to lower case letters or vice versa. The technique is mentioned in [2, 26, 32, 99, 187, 188].

- *Other PPTs* - The techniques that are counted under others are only mentioned in one article and could not be grouped in any of the previous categories. The techniques counted here are: integration of files into one file [26], ignoring programs with syntax error [137], and preprocessing based on design patterns (de-patterning) [6].

The techniques RC, RWS and TE could be seen as special cases of the PPT ‘Remove Code Parts’, but since many articles are mentioning the first two techniques and since some articles are mentioning explicitly the third one, the three techniques are separated. Similarly ‘String to Upper/Lover Case’ technique could be seen as special case of ‘Change Code Parts’ but again since multiple articles explicitly mention the ‘String to Upper/Lover Case’ it is listed separately.

The mentioned techniques can be divided into two groups according to how they operate: *removal techniques* and *modifier techniques*. Removal techniques try to remove code that complicates in some way the detection, while modifier techniques try to modify the code (with or without changing the logic) to get better performance.

When looking for ideas for new preprocessing techniques one should look at obfuscation methods and descriptions of to what some tools are resistant. For example, in the article [154] authors describe the successful and non-successful plagiarized attacks (like insertion, modification, or deletion of comments or change names of variables, methods, or classes) and from those, one can get ideas for new or recognize existing PPTs. Similarly, in [173] the list of redundant statements (like unreachable code, uncalled procedures and functions, extra semicolons, unused variables or parameters) which one could add to hide plagiarism can be used to get ideas for the PPTs.

The usefulness of some technique depends among other things on the tool and its underlying algorithm that the tool is using. This raises the question what is the effect of such techniques and should they be performed? For example, since most tools use tokenization the 'Change Code Parts' techniques that do renaming of variables or functions will be ineffective since tokenization transforms the code into a set of tokens which should ensure that modifications like renaming have no effect. Therefore the technique is unnecessary in combination with such tool.

7.2 Selection of preprocessing techniques

For the experiment, it was decided to use no more than six different techniques to limit the amount of analysis, as stated in Chapter 3. This number includes also combinations of techniques. During the literature review, there was no article that would compare the effectiveness of different preprocessing, and because of that, it is not possible to select the techniques based on their quality. The decision was to first select the techniques based on how often some technique is mentioned to be used in the literature. If a technique is used by more tools it is more probable that it has some effect on improving the detection, otherwise, why use the technique.

From the review, in Table 7.1, one can see that the two most used techniques are RC and RWS. Followed by 'Remove code parts', 'Change Code Parts' and TE. While multiple implementations exist all off them have the same goal and it is decided to use the most convenient one. Most implementations are integrated inside of a tool and it is difficult or impossible to extract the preprocessing technique. From the top rated systems (presented in Section 4.1) only Sherlock has explicit possibility to perform preprocessing without the detection. Sherlock offers implementations for the following PPTs: Remove White Spaces, Normalisation, and Remove Comments.

Based on the names one can conclude what RWS and RC do, also they are described in Section 7.1. On the other hand, the name Normalisation (NOR) does not say much. NOR technique in Sherlock does among others the following: removes redundant spaces in one line

between keywords and signs, splits every if statement in separate line on every '&&' or '||' character, it removes all indentation, etc. NOR technique includes in some way RWS technique and adds the mentioned modifications. In general, it changes the code so that it unifies the style to a single format and therefore it would go into the category 'Change code parts'. The three implementations can be used, but unfortunately the technique RC in Sherlock has a bug, so new implementation of the technique is developed described in Section 7.3.

No ready to use implementation was found for the 'Remove Code Parts' technique so it was decided to develop a new technique and use the suggestions from [38] with some additional removals. The new technique is called *Common Code Remove (CCR)*, further sometimes refereed to as *Remove Common Code*. A detailed description of the technique is given in Section 7.4.

TE is one technique that has most available implementations, it is implemented for example in JPlag as well in Sherlock. Problem is that even though the implementations exists it is not convenient to extract them for independent preprocessing technique and since the implementation differs it is not an option that each tool uses its own implementation. Therefore a new version of the template exclusion was implemented and presented in [95]. Results in [95] indicate that template exclusion has some positive effect on the detection at least in combination with the tool Sherlock. Since in [95] the focus was on the process model and not on the technique a detailed description is given in Section 7.5.

Regarding the combination of techniques, Sherlock offers combinations of Remove Comments with Remove White Spaces, Remove Comments with Normalisation. But since the Multiple Plagiarism Checker (MPC) system allows combining any techniques and allows to specify the order of execution, MPC is a better option. Since the individual effect of a technique is unknown it is hard to choose the combinations, so only one combination that will include all of the chosen techniques is considered useful to be used in the experiment. The reason why combining all techniques is considered useful is the following. If every technique has a positive effect, combining all of them should then have an even better effect than each of them individually. To not jump into conclusions to fast more inside will be given with the following tests.

In summary, to have a representative from each of the top five PPT categories the following techniques are selected: *RC, RWS, NOR, CCR, and TE*. These techniques are further analysed in a simple effect test (further referred to as PPTest) in Section 7.6. The reason for this PPTest is the following. From Table 7.1 one can see that remove comments and remove white spaces techniques are used most of the time. At the same time, these are also most primitive ones and the effect is questionable, so it is the idea of the PPTest to remove techniques that show no effect on a small dataset to lower the number of combinations in the main experiment.

7.3 Remove comments technique

RC is the most popular technique which is available in tool Sherlock, but as noted before it has a bug. The bug manifest itself by not removing only comments. For example, if there is a line like "*System.out.println("http://www.somepage.com");*" Sherlock's RC technique will delete it and this will cause some problems in the parsing of JPlag-java tool. Because of that, a new implantation of RC technique was developed.

The goal of the technique is simple, remove all comments from source-code. This includes multi line comments and single line comments. In Java multi line comments begins with slash sign followed by star sign and ends with start sign followed by slash sign (*/* multi line comment */*), and single line comments begins with two slash signs and ends with a new line (*// single line comment*).

Implementation of the technique is very simple the characters of the files are read one by one and if a beginning of a comment is recognized (single or multi line) the flowing chars are skipped until the end of the comment is reached.

7.4 Common code remove technique

CCR technique is a preprocessing technique to remove code that is common and usually can not be used to identify plagiarism. Common code mostly gives higher percentage rate and enforces false positives. The CCR technique includes in its implementation of multiple actions which can be seen as individual techniques. CCR is, therefore, a technique which combines multiple techniques. Elements that the CCR technique removes are:

- package and import statements;
- annotations - remove all annotations that exist with or without parameters, and with or without sub annotations;
- setter methods - methods that only set one variable and have no other logic, this includes all forms of spaces between words and parenthesise, and with or without keyword 'this';
- simple 'setter' methods - it is a 'setter' method which does not have the word set but the only thing the method does is to set a field. This method must use the word 'this' for setting up a field otherwise it is not removed, also if any statement is there that does not set a field the whole method is not removed;
- getter methods - methods that only get one variable and have no other logic, this includes all forms of spaces between words and parenthesise, and with or without keyword 'this';
- empty constructors - including constructors which only call the super method with or without parameters;

- simple constructors - constructors that only set up fields by assigning a value to the fields, the constructor must have a ‘this’ keyword, if the constructor has one element without ‘this’ keyword or does something else, it is not removed. The only exception is the call of the ‘super’ method;
- empty functions - functions that do nothing and which have any number of input arguments, but they have no statements or just have an empty return statement;
- empty classes - classes with no fields and methods. If the class has implement or extend statement is not important, this also includes classes which will become empty because of using the techniques described here;
- non initialized class fields - removing all fields that are not statically initialized and have specified access modifier. Fields that don’t have access modifiers or are statically initialized are not removed. Although fields can have some value for the detection they are in most cases more misleading and don’t implement real functionality;
- empty blocks - removes empty loops (like while loop, for loop, or do-while loop) blocks, empty if-else blocks or just else blocks, empty switch blocks, empty try-catch or try-catch-finally or finally blocks;
- leftover white spaces - removes all double white spaces. This is necessary because by removing all of the mentioned, big blocks of white space are generated. These white spaces are removed at the end which has the effect that the output for the analysis is nicer.

7.5 Template exclusion technique

TE is, as already stated, based on the implementation described in [95]. The technique uses Sherlock-text in the background where it performs similarity detection for every student submission to the template code. The template code can be any code that is given to the students by the teachers and that can be used in the assignments without considering it plagiarism.

TE technique from [95] can work with multiple templates. The technique first searches for the best suitable template, which means it has the most similarity with the submission. Once the best suitable template is found it removes the similar part from the submission. The detection is repeated until the similarity between the submission and the template is at zero percentage. The possibility of working with multiple templates is important since sometimes there are more teachers in one course and every teacher can give to his group modified versions of the examples. Combining all examples could lead to a huge template slowing down the detection, so it is more efficient to create multiple templates. For more detailed explanation read [95].

The TE technique [95] uses Sherlock-text so it is dependent on Sherlock’s configuration parameters. For the experiment the same parameters are used which were found during the calibration (Section 4.4.3). A simple test was performed to see how calibrated configuration

Table 7.2: Configuration comparison based on removed lines of code for template exclusion technique

Submission	Calibrated	Default	Differance
Calibrated configuration is better			
S4	455	11	444
S8	289	19	270
S3	226	46	180
S1	191	53	138
S5	147	125	22
Default configuration is better			
S6	19	52	-33
S2	53	89	-36
S7	0	57	-57

performs in comparison to the default configuration used in [95]. The test was to count how many lines of the template code were removed.

On eight randomly selected student submissions from the Real Student Solution (RSS) dataset the calibrated configuration removed on average more Lines of code (LOC). Table 7.2 presents the result of the test, whereby the first two numeric columns present the number of removed LOC that were not removed with the other configuration. The last column presents the difference of removed LOC, if the number is positive the calibrated configuration performed better, if the number is negative the default configuration performed better. One can see that only on 3 submissions the default configuration was better removing on average 42 LOC more than the calibrated configuration, while the calibrated configuration was better on 5 submissions removing on average 211 LOC more than the default configuration.

The TE technique in the article [95] is based on Sherlock text and Java version. Since in this research the technique needs to be the same for text and Java version, only the text version is used. This is necessary to be able to compare later the effect of the technique between Java and textual detection. The Java version is not an option since it produces an output that can not be used with other tools except Sherlock.

Problem with the implementation of the TE technique described in [95] is that it does not take care of keeping the code parsable. This is not a problem for Sherlock since it does not parse the code that is compared, but for JPlag-java that uses a parser, this means that most submissions are not ok. To solve that the implementation was modified so that only complete blocks can be removed. For example, if during detection some segment of a function needs to be deleted and if this segment contains an open or closed bracket of the function that bracket must not be deleted. Deleting this bracket would mean that the function would at the end miss the closing or the open bracket causing parse error in JPlag-java.

Because of that problem, the new implementation of the TE removes a bit less template code that the original implementation and it is more complicated since it takes care that nothing is

deleted that could cause parse problem. Otherwise, the new implementation flows the same process and idea as described in [95].

7.6 Technique selection test

In Section 7.2 five techniques were selected based on the literature and it was stated that one combination of the techniques should be added to the list. Just to clarify why doing all possible combinations is unrealistic let us suppose that five techniques are used, this makes 10 combinations of 2 techniques, 10 combinations of 3 techniques, 5 combinations of 4 techniques and one combination with all techniques. Doing all 31 combinations is not possible, especially when this is combined with 6 tools, 3 academic years (at least) and 2 assignments in a year this leaves 1116 results which need to be analysed by an expert. In addition, each result has multiple individual pairs that need to be examined. Because of that, the limit of maximum six techniques or their combinations is chosen.

On the five selected techniques, a PPTest of effectiveness was performed. This PPTest should help to select, for the main experiment, techniques which have the best predispositions to give interesting results. The PPTest was done on a dataset containing four randomly selected pairs from the RSS dataset. It was expected that the similarity should differ in comparison when no technique is used. It is not important at this point how big the difference is and if the change is positive or negative.

The PPTest was done on all three tools on both versions, in total six tools. All tools were configured with values selected in the calibration phase (Section 4.4).

The results are presented in three tables, for JPlag and SIM Java versions in Table 7.3, for JPlag and SIM text versions in Table 7.4, and for Sherlock in Table 7.5. From Table 7.3 it is visible that RC, RWS and NOR techniques have the same similarity for SIM and JPlag Java versions. For the textual versions of SIM and JPlag the situation is similar, with the difference that RC technique causes differences in similarity. Because of this, some combinations RC, RWS and NOR techniques where tried out.

The three tested combinations have no effect on Java versions of SIM and JPlag while there is some difference in similarity for the textual version, but the similarity is the same as with the RC technique. The reason for these results is that SIM and JPlag ignore comments in Java version and are already immune to some simple stylistic changes like adding white spaces and ignore them in textual and Java version.

The techniques CCR, TE and combinations of TE with CCR and combination of all techniques make some changes in similarity in comparison to when no preprocessing is used. The combination with all techniques and the combination of TE with CCR has the same effect in Java version while in the text version there is some difference.

For Sherlock, on the other hand, every technique and every combination that has been tested causes a difference in similarity in comparison to no preprocessing. This is no surprise since

Table 7.3: PPTest similarities for JPlag-java and SIM-java

Technique Name	JPlag Java				SIM Java			
	E1.	**E2.**	**E3.**	**E4.**	**E1.**	**E2.**	**E3.**	**E4.**
No Preprocessing	83.0	77.2	25.7	47.7	86.0	78.5	26.0	50.5
Techniques								
Remove Comments (RC)	83.0	77.2	25.7	47.7	86.0	78.5	26.0	50.5
Remove White Spaces (RWS)	83.0	77.2	25.7	47.7	86.0	78.5	26.0	50.5
Normalise (NOR)	83.0	77.2	25.7	47.7	86.0	78.5	26.0	50.5
Common Code Remove (CCR)	**81.5**	**61.2**	**17.9**	**36.6**	**85.0**	**61.5**	**16.0**	**39.5**
Template Exlusion (TE)	**72.7**	**18.5**	**9.1**	**18.4**	**78.5**	**20.5**	**4.5**	**19.0**
Combinations								
All (TE-RC-CCR-NOR-RWS)	**74.2**	**17.2**	**8.8**	**21.0**	**79.5**	**19.5**	**5.5**	**19.5**
RC-NOR-RWS	83.0	77.2	25.7	47.7	86.0	78.5	26.0	50.5
NOR-RC	83.0	77.2	25.7	47.7	86.0	78.5	26.0	50.5
RC-NOR	83.0	77.2	25.7	47.7	86.0	78.5	26.0	50.5
TE-CCR	**74.2**	**17.2**	**8.8**	**21.0**	**79.5**	**19.5**	**5.5**	**19.5**

Note:
All values that are different than the values when no PPT is used are bolded.

Sherlock was implemented to use the preprocessing to be more efficient. One unexpected result was that the combinations RC-NOR and NOR-RC gave different results, which shows that the order in which techniques are executed makes a difference for Sherlock. Another thing that was unexpected that combination RC-NOR and RC-NOR-RWS have the same effect. This indicates that RWS technique is unnecessary if NOR technique was used before. Also, it shows that NOR technique already removes all unnecessary white spaces.

Based on the results for all six tools it can be concluded that RWS technique is not a useful technique to be used and that it is covered by the NOR technique. Also, the results suggest that NOR technique is of no use for other tools except for Sherlock.

To be even more confident what techniques to chose another from the ones that had a low effect in the PPTest (RC, RWS, and NOR), a test was done to favour some technique (referred to as favour test). For favour test, four cases which are 100% plagiarised were created manually and some simple modifications were made to favour some technique. The four cases are: a case of plain copy, a case where only comments were removed (favour RC technique), a case where only new white spaces where added (favour RWS technique), and a case where modifications were made to favour NOR technique. All combinations with two of these techniques in both orders were also tested in the favour test.

Favour test confirmed that for JPlag and SIM Java versions there is no need for RC, RWS and NOR techniques. The textual versions, as before, only RC technique makes a difference in comparison to no technique. For JPlag-text the similarity jumps from 63,1% with no technique to 99,6% with RC technique, and for SIM-text the similarity jumps from 60% with not technique

Table 7.4: PPTest similarities for JPlag-text and SIM-text

Technique Name	JPlag Text				SIM Text			
	E1.	**E2.**	**E3.**	**E4.**	**E1.**	**E2.**	**E3.**	**E4.**
No Preprocessing	62.7	73.0	24.5	22.0	64.5	72.0	30.5	18.5
Techniques								
Remove Comments (RC)	**61.9**	**74.7**	**19.5**	**35.9**	**63.0**	**73.5**	**22.4**	**37.0**
Remove White Spaces (RWS)	62.7	73.0	24.5	22.0	64.5	72.0	30.5	18.5
Normalise (NOR)	62.7	73.0	24.5	22.0	64.5	72.0	30.5	18.5
Common Code Remove (CCR)	**60.0**	**47.0**	**19.1**	**11.7**	**62.0**	**61.0**	**27.0**	**12.1**
Template Exlusion (TE)	**47.3**	**9.0**	**12.3**	**7.5**	**52.5**	**8.5**	**26.5**	**7.7**
Combinations								
All (TE-RC-CCR-NOR-RWS)	**50.1**	**14.2**	**2.3**	**2.8**	**51.0**	**12.0**	**1.3**	**2.6**
RC-NOR-RWS	61.9	74.7	19.5	35.9	63.0	73.5	22.4	37.0
NOR-RC	61.9	74.7	19.5	35.9	63.0	73.5	22.4	37.0
RC-NOR	61.9	74.7	19.5	35.9	63.0	73.5	22.4	37.0
TE-CCR	**53.1**	**13.6**	**13.3**	**2.2**	**53.0**	**14.5**	**25.9**	**1.6**

Note:
All values that are different than the values when no PPT is used are bolded.

Table 7.5: PPTest similarities for Sherlock

Technique	Sherlock Java				Sherlock Text			
	E1.	**E2.**	**E3.**	**E4.**	**E1.**	**E2.**	**E3.**	**E4.**
No Preprocessing	47	87	15	19	37	60	3	6
Techniques								
Remove Comments (RC)	54	82	15	47	29	62	3	6
Remove White Spaces (RWS)	66	69	16	10	36	60	3	3
Normalise (NOR)	58	71	7	13	43	56	6	3
Common Code Remove (CCR)	62	59	22	26	25	31	6	4
Template Exlusion (TE)	48	22	9	3	20	6	1	0
Combinations								
All (TE-RC-CCR-NOR-RWS)	36	23	7	5	41	5	3	1
RC-NOR-RWS	63	82	19	31	52	55	10	29
NOR-RC	73	75	18	32	36	55	6	6
RC-NOR	63	82	19	31	52	55	10	29
TE-CCR	41	21	11	7	18	3	1	0

Table 7.6: Similarities for Sherlock's favour test

	Sherlock							
	PlainCopy		Favour RC		Favour NOR		Favour RWS	
Technique name	**Java.**	**Text.**	**Java.**	**Text.**	**Java.**	**Text.**	**Java.**	**Text.**
No Preprocessing	100	100	55	19	57	5	83	14
Techniques								
Normalise (NOR)	100	100	0	18	100	98	100	100
Remove White Spaces (RWS)	100	100	0	11	100	34	100	100
Remove Comments (RC)	100	100	69	50	57	5	83	14
Combinations								
RC-NOR	100	100	98	98	100	98	100	100
RC-RWS	100	100	38	79	100	34	100	100
NOR-RC	100	100	91	25	100	98	100	100

to 100% with RC technique.

Favour test on Sherlock gives the following results: for plain copy case no technique is needed, for favour RC case the remove comments gives better results than no technique but it is best when also NOR technique is used after RC technique, for favour NOR case the NOR technique is best option and combinations with this technique, and for favour RWS case the NOR and RWS techniques give the best results. In Table 7.6 the results for Sherlock are presented, only three combinations are presented the other are removed since they gave the same results as one of the presented combinations. For a remainder, the goal was that the technique should rise the similarity to 100%.

Based on the PPTest and the favour test it was decided to remove RWS technique from the experiment since it is covered by NOR technique. RC technique will be used since it shows some effect on textual versions of for all three tools. CCR technique and TE technique will be kept since they gave the biggest effect. The NOR technique gave only some effect on Sherlock so it was decided to use it only in a combination.

Most combinations are giving the same results as the individual techniques so they are not used. The TE-CCR combination gave some results but it was in some cases equal to the combination when all techniques are used and for others the numbers did not differ too much either form TE technique, CCR technique or the combination when all techniques where used so it was decided to not use it in the experiment.

The combination using all techniques is selected for the experiment but without the RWS technique. In addition, another combination technique which will use all techniques except for RWS and NOR is also selected for the experiment. The reason why these two combinations are chosen is that NOR technique is not tested alone but it is important for Sherlock it is included in the first combination. On the other hand, the technique did not affect JPlag and SIM so another

combination is used to have the possibility of comparison.

In summary, the following individual techniques are selected for the experiment: TE, CCR, and RC. In addition to the individual techniques two combinations are selected: combination including the TE, RC, CCR and NOR techniques (referred to as All techniques with Normalisation (AllNOR) technique or AllNOR combination) and combination including the TE, RC, and CCR techniques (referred to as All techniques without Normalisation (AllnoNOR) technique or AllnoNOR combination). The order of execution of the individual techniques in the combinations is as listed.

The order is important and it is probable that another order would give different results. The reason why this particular order is chosen is the following. TE technique needs to be first since the templates itself are not preprocessed in any way, so the exclusion of template needs to happen on the original file to have the best effect. RC technique and CCR technique should be independent, meaning they remove completely different things so the order should not make a difference, but since CCR technique has at the end removal of leftover white space it is put after RC technique. In this way if RC technique leaves some empty lines CCR technique can clear it up.

NOR technique is executed at last since it is only necessary for Sherlock. Also, in the favour test (favour RC case) it showed better results when it was put after RC technique. First reason why NOR technique was chosen to be after CCR technique is that CCR uses the regular expression in some parts of the implementation and NOR technique could make CCR technique less efficient. The second reason is that NOR technique reformats the code to have a unified style, so it is best to be used last after all removals are performed. A third reason is that NOR technique clears all white spaces and it makes sense to do that at the end.

In total there are five techniques with three selected individual techniques and two combinations of the techniques. Since the SOurce COde Reuse (SOCO) dataset does not have templates the TE technique was excluded and the selected combinations were used without TE technique.

CHAPTER 8

RESULT ANALYSIS

This Chapter provides the complete results of this research. Before the results are presented there is a description of some preparations that are important for the analysis.

The results are divided into two sections. First, the results from the SOurce COde Reuse (SOCO) dataset are described and then the results from the Real Student Solution (RSS) dataset are described. Next, a Section is given which describes the interesting findings regarding the textual versions of the tools. The Chapter finishes with a Section summarizing the contributions of this research which are based on the results from both datasets.

8.1 Preparation for analysis

This Section describes the selection of the threshold level (which is used to calculate the Precision, Recall and F1 measure) and the contrasts that are used in the statistical analysis.

8.1.1 Threshold level selection

In Chapter 5 it was stated that to calculate the F1 value one needs to set a threshold level, and in Section 3.2 it was discussed why using a flexible threshold level is better than using a fixed threshold level.

There are two main reasons for using the second option over the first option. The first reason is that it is more useful in practice, since one needs to know the exact number of plagiarised matches for the usage of the first option, while in the second option one has the possibility to calculate the threshold without knowing that number. Every time a teacher uses a similarity detection tool the exact number of plagiarised matches is not known but a tool needs to decide which cases it will mark as potentially plagiarised, in other words, it needs to set the threshold level. Because of this, it makes more sense to test the tools with the calculated threshold rather than base it on the number of plagiarised matches.

The second reason for choosing the calculated threshold is that it takes into account the similarity when calculating the threshold and therefore it can not happen that the similarity of the last marked match is the same as the similarity of the next match (the first match which is not marked as plagiarised). For example, in the T1 assignments in the SOCO dataset the number of plagiarised matches is 84, which means the cut-off is at the 84th match. In Table 8.1 the similarities for the six tested tools are given when no technique is used for the 84th and 85th match. One can see that the similarity is the same at 84th and 85th match for four tools, which makes it unreasonable to set the threshold at the 84th element. Sometimes only one match, but several matches (which are not marked as plagiarised) have the same similarity as the

Table 8.1: Example of SOCO T1 similarities near threshold based on number of plagiarised matches

	Match			
Tool	**1st**	**84th**	**85th**	**diff**
JPlag-java	100	47.4	47.2	0.2
JPlag-text	100	46.0	45.5	0.5
SIM-java	100	56.5	56.5	0.0
SIM-text	100	50.0	50.0	0.0
Sherlock-java	100	56.0	56.0	0.0
Sherlock-text	97	27.0	27.0	0.0

last marked match. This problem could be solved by including all matches that have the same similarity as the last marked match, but (for the first reason above) the decision remains to use the calculated threshold.

Similarity based threshold level calculation

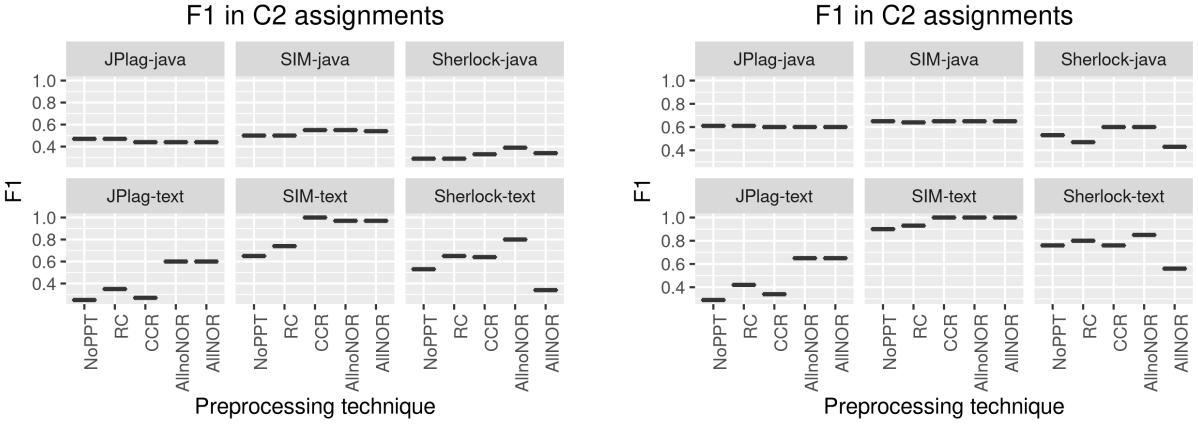
To calculate the threshold level based on similarities one needs to calculate the median and the inter-quartiles (IQR), and this raises the question as to what number to multiply the IQR by in the threshold level calculation. There is no universal answer to this question since it depends on the data distribution in the dataset. In the context of comparisons for some tools, 2*IQR would be better, and for other tools, 3*IQR would be better. The idea of using a calculated threshold based on similarities is to look for matches that come out as outliers in the distribution. From this perspective, 3*IQR is better than 2*IQR. This means the match is more suspicious when its similarity is 3*IQR from the median than 2*IQR. On the other hand, it could happen that there are no matches which have a similarity larger than 3*IQR from the median. It is out of the scope of this research to test the differences on various IQR multipliers, so a decision needed to be made.

To decide, a simple test on the T1 and C2 assignments from the SOCO dataset was performed where the F1 measure was calculated for 3*IQR, 2.5*IQR and 2*IQR. In Figure 8.1 presents the F1 values for all techniques and tools for the T1 assignments when 3*IQR and 2*IQR were used. Similarly, Figure 8.2 presents the F1 values for the C2 assignments when 3*IQR and 2*IQR were used. One can observe that the F1 value always increases with the higher multiplier with two exceptions. The first exception is for Sherlock-java in combination with All techniques with Normalisation (AllNOR) in the T1 assignment when the F1 value slightly decreases. The second exception is for SIM-text in combination with Common Code Remove (CCR) in the C2 assignments when it is at maximum value and cannot increase any more.

This simple test showed that if a lower IQR multiplier is chosen then F1 is also lower in most cases. The reason for that is simple, with a lower IQR multiplier more matches are included which then increase Recall but decrease Precision more. Since in every combination 3*IQR gave

(a) with 2*IQR **(b)** with 3*IQR

Figure 8.1: F1 score for SOCO T1 assignment



(a) with 2*IQR **(b)** with 3*IQR

Figure 8.2: F1 score for SOCO C2 assignment

no zero F1 values (meaning all calculations had matches with similarity above 3*IQR larger than the median), and since 3*IQR almost always gave a better F1 result than the 2.5*IQR and 2*IQR, it was decided to use 3*IQR to calculate the threshold.

Note that, while the increase is present in almost all F1 values, the increase for various combinations of tools and techniques is not constant. For example, for Sherlock-text in combination with CCR, the increase was 0.04 while for SIM-text in combination with CCR it was 0.13. For Sherlock-text in combination with AllNOR the increase was 0.3 and for SIM-text in combination with the same technique it was 0.09. This means that by choosing 3*IQR some combinations of the tools and techniques might have been put in a privileged position in comparison to others. In other words, different IQR multiplier could influence the final results. For this reason, the conclusions that follow are valid only when F1 is calculated using 3*IQR to calculate the threshold. While Figures 8.1 and 8.2 suggest that the conclusions might be valid for other multipliers of IQR, more research is needed to confirm that.

After calculating all F1 values for the SOCO dataset, in total 3780 calculations for all assignments with all combinations of tools and techniques, there were six zero values. The six values had F1=0 because there were no matches above 3*IQR larger than the median. Since there were only six such values it is confirmed that 3*IQR was a valid choice.

8.1.2 Planned comparisons

In Chapter 3 it was stated that Analysis of variance (ANOVA) would be used followed by planned comparisons (contrasts). To perform contrasts they need to be defined upfront. The contrasts are defined for each independent variable (factor). The contrasts for the tool factor are presented in Table 8.2 and for the technique factor in Table 8.3.

The first contrast for tool compares textual tools with Java tools. The second 'level' contrasts (contrasts 2 and 3) compare SIM and JPlag combined with Sherlock. The reason why SIM and JPlag were put together is that Sherlock is the only tool of the three that depends on preprocessing in the original implementation, and because the Normalisation (NOR) technique comes from Sherlock. The third level contrasts (3 and 4) compare SIM and JPlag.

The first contrast for the technique is comparing all techniques to when no PPT is used. The

Table 8.2: Planed comparisons for tool factor

Contrast			
Number	**Name**	**Partition 1**	**Partition 2**
1	TextvsJava	Sherlock-text, SIM-text, JPlag-java	Sherlock-java, SIM-java, JPlag-java
2	Text.SherlockvsOthers	Sherlock-text	SIM-text, JPlag-text
3	Java.SherlockvsOthers	Sherlock-java	SIM-java, JPlag-java
4	Text.SIMvsJPlag	SIM-text	JPlag-text
5	Java.SIMvsJPlag	SIM-java	JPlag-java

Table 8.3: Planed comparisons for technique factor

Contrast			
Number	**Name**	**Partition 1**	**Partition 2**
1	NoPPTvsPPT	No PreProcessing Technique (PPT)	Template Exclusion (TE), Remove Comments (RC), Common Code Remove (CCR), All techniques without Normalisation (AllnoNOR), All techniques with Normalisation (AllNOR)
2	SinglevsCombo	RC, CCR, TE	AllnoNOR, AllNOR
3	RCvsCCRandTE	RC	CCR, TE
4	CCRvsTE	CCR	TE
5	AllnoNORvsAllNOR	AllnoNOR	AllNOR

second contrast is comparing single techniques to combo techniques (techniques that consist of two or more techniques). Contrasts 3 and 4 compare the single techniques (i.e.,RC, CCR and TE), and contrast 5 compares the combo techniques (i.e.,AllnoNOR with AllNOR). Since the SOCO dataset does not have templates the TE technique was not used, so contrasts 3 and 4 are merged to just compare RC with CCR.

The first contrast in technique is very important since it is used to test the first hypothesis (H1) and the other contrasts are important to check the second hypothesis (H2) and partially answer the first research question (RQ1). While the contrasts for the tool factor are important, the contrasts for the technique variable are more important since they are directly used to answer the question of this research and confirm the hypothesis.

The coding of the contrasts for tools and techniques is presented in Appendix E from which one can see that they are all coded as orthogonal contrasts, in accordance with the suggestions in [46, pp. 414-425]. Making the contrasts orthogonal it is easier to interpret the results since the comparisons are not correlated and the p-values are also not correlated [46, p. 426]. In addition, by using contrasts (and not post-hoc tests) the family wise error rate is controlled [46, p. 423-424], meaning rather than using the absolute difference (for the number of the regression coefficient) of the means, the absolute difference is divided by the number of groups used in the contrast.

Simple effects analysis

While the planned contrasts yield good statistics they are not enough to check the hypothesis. Planned contrasts give the possibility to analyse and truly understand the effects that are obtained. They are stated in such a way that, if an interaction is present, they enable an analysis of the nature of the interaction. For example, if the interaction is present, the contrast Textvs.Java and NoPPTvs.PPT give the possibility to answer the question: "*Do the changes in accuracy*

when using PPTs (in comparison to when no technique is used) differ between textual and Java tools?"

The limitation of such contrasts is that they do not tell anything about the differences between the individual levels. To elaborate, for the above example it might be that the changes in accuracy found when PPTs are used (in comparison when no PPT is used) are significantly different between textual tools and the Java tools, but this does not say anything about the significance of the individual change in accuracy when PPTs are used (in comparison when no PPT is used) for the Java tools or the textual tools.

To solve that problem, a simple effects analysis is used (as described in [46, pp. 525-528]) to break down an interaction term. "*The analysis looks at the effect of one independent variable at individual levels of the other independent variables*" [46, p. 525]. In this research, the simple effects analysis looking at the effect of techniques at each level of tool is performed. The reason for *why* we should analyse the effect of techniques at each level is that in the primary interest of this research is *the effect of techniques* and therefore this set-up enables the testing of the stated hypotheses (H1 and H2).

To perform simple effects analysis a new variable must be created which combines all levels of the tool variable and the technique variable (called *ToolTechniqueCombo*). Contrasts that were created for this new variable are the same as described above, with the difference that since the tool and technique are now combined, first the tool contrasts described in Table 8.2 are used and then contrasts for the technique are used on each individual tool (Table 8.3). In Appendix F the coding of these contrasts is presented.

8.2 SOCO dataset analysis

In this section, the results based on the SOCO dataset are described. As already stated, data are analysed qualitatively and quantitatively, and to ensure objectivity, statistical tests are performed. Before the SOCO dataset was ready to be used and analysed statistically some preparations were needed which are described first, after that for each assignment of the SOCO dataset is analysed individually and then an overall discussion is given which ends with a set of guidelines. The last part of this Section is the qualitative verification of the guidelines.

8.2.1 SOCO dataset preparation for analysis

Before the analysis on the SOCO dataset could be performed some preparation was needed that is described in three subsections. First, there is an inability to process four SOCO assignments, and there is the problem that all assignments of the SOCO dataset need to be equalized in the ratio of plagiarised to non-plagiarised matches. Second, to have a good statistical analysis, assignments need to be analysed separately, since there are variances between them which could lead to wrong conclusions if they were treated as equal. Third, solutions to the violation of the assumptions required to perform ANOVA need to be found.

Equalization of the ratio of plagiarised to non-plagiarised matches

It was already stated (Section 6.2) that the results from the T1, C1 and C2 assignments are used *as is*, and that the results from the A1, A2, B1 and B2 assignments are used partially. The reason for using the A and B assignments partially is that they are too large to be used for this experiment, the calculations would take too long.

Since there is a big difference in the ratio of plagiarised and non-plagiarised matches between the first group of assignments (T1, C1, C2) and the second group of assignments (A1, A2, B1, B2) it was decided to use a subset of the second group of assignments where the ratio is more equal to the first group.

In Table 6.1 the SOCO dataset structure is presented and from it one can easily calculate that the ratio of plagiarised matches to non-plagiarised matches for T1 is 0.25% and for C2 it is 0.37%, while the ratio for the A and B assignments is approximately 0.001%. To put the ratios of the A and B assignments closer to the T1 and C2 datasets, a subset of the A and B assignments was created so that the ratio is 0.31%, which is the average of the ratios for the T1 and C2 assignments.

To create a subset of assignments called D1 the following process was used. From the A1 assignment all plagiarised files were taken (86 files) and it was calculated that to have a ratio of approximately 0.31% 102 non-plagiarised files were needed. The 102 non-plagiarised files were then selected at random from the original assignment. The same procedure was used to create the subset of the A2, B1 and B2 assignments. These subsets were named from D1 to D4 to make a clear distinction from the original assignments. The complete structure of the SOCO dataset, and how it was used in the experiment, is presented in Table 8.4 in the same format as in 6.1 with the additional column *Original name* which is the name of the assignment from which the new assignment was created.

Table 8.4: SOCO dataset structure for experiment

	Assignment		**Number of**			
Collection	**original**	**new**	**Files**	**Plagiarized files**	**Matches**	**Plagiarised matches**
Test	A1	D1	188	86	17,578	54
Test	A2	D2	175	75	15,225	47
Test	B1	D3	218	124	23,653	73
Test	B2	D4	149	62	11,026	34
Test	C1	C1	124	0	7,626	0
Test	C2	C2	88	28	3,828	14
Training	T1	T1	259	115	33,411	84

Preparation for statistical analysis

To perform a statistical analysis on the SOCO dataset the assignments are seen as participants on which the statistical analysis is performed using the various tools and techniques. The problem is that there are only six assignments which are different enough to create large variances, and this gives low power to the statistical test. In addition, it is hard to get any statistically significant results. To solve that problem subsets of the assignments were created from the larger assignments (A1, A2, B1, B2), an idea taken from the information retrieval area (e.g., [39])

Since the D1 to D4 assignments are subsets generated from the A and B assignments, it is possible to generate more subsets which come from the same original source. For this research, 31 subset assignments were generated from each of the original A and B assignments. The subset assignments were marked as D1-n where n is a number from 1 to 31. In this context, D1 represents a group of subset assignments marked from D1-1 to D1-31 whereby each of these is generated from the same original source and using the same generation process, so the ratios and numbers from Table 8.4 remain. One subset assignment is seen as one participant on which the statistical analysis is performed using the various tools and techniques. The only difference between assignments in the D1 group are the randomly selected non-plagiarised files. This means that all assignments in the same group have the same characteristics (i.e., problem scenario and complexity mentioned in Section 6.2).

To do a statistical test, only assignments from one group are tested together to control the variability. This means that at the end four separate case studies are done one for each group (D1, D2, D3 and D4). Because of this design, T1 and C2 — although they have the F1 measure calculated — were not tested statistically. However, they are analysed manually and used to verify the guidelines generated based on the comparisons done with the D1, D2, D3 and D4 groups of subset assignments. In the rest of the thesis, D1, D2, D3 and D4 are referred to as *assignments* rather than groups of subset assignments.

Overcoming assumption violations

To see if observed differences are significant it was planned to use ANOVA as stated in Chapter 3. To use ANOVA one assumption is that the data within groups are normally distributed [46], in other words, that the residuals are normally distributed. In addition, since this is a repeated measures design there is the assumption of sphericity [46, p. 551]. Sphericity is not a problem if ANOVA is performed using Multi level linear model (MLM) [46, p. 576], sometimes referred to as mixed-effects model.

In Figure 8.4, the residuals histogram and Q-Q plot are presented for the D1 assignments where the data look more or less normal. A similar observation was made for the D2 assignments, as shown in Figure 8.7. To be sure, a Shapiro-Wilk test of normality was done on each level of the tools and techniques, and it was established that for some combinations of technique and tool the data are not normally distributed (Appendix G). Regarding the D3 and D4 assignments,

it is already clear from Figure 8.10 and Figure 8.13 that the data are not normally distributed. All results of the Shapiro-Wilk tests are presented in Appendix G.

Since there is a clear problem of normality in all of the datasets it was decided to use the bootstrap method [46, 87] to overcome this problem. The bootstrap method enables us "*to generate confidence intervals and test statistical hypotheses without having to assume a specific underlying theoretical distribution*"[87, p. 309]. In addition, the bootstrap method can be used if outliers are a problem [87, p. 304] or if there is a problem with heteroscedasticity [46, p. 298]. Figures 8.4, 8.7, 8.10 and 8.13 present the scatter plot of residuals against predicted values to check for heteroscedasticity.

According to [46, 189], repeated measures ANOVA can be performed using MLM rather than using the classic ANOVA approach based on linear regression. Since in this research there are multiple factors and the repeated measures design is used, the MLM approach is used as done in [46]. The only difference to [46] is that instead of using the *lme* function from package *nlme* the function *lmer* from package *lme4* is used. That both functions can be used is demonstrated in [189]. The reason why *lmer* is chosen over *lme* is because *lmer* is faster in execution and it has some other functions like *bootMer* from the *lme4* package or *PBmodcomp* function from the *pbkrtest* package. The main reasons to use MLM are that it is easier to add multiple factors to the model and that it is easy to model the variance (in this case the variance within one assignments subset) by using random intercepts [46]. In addition, as already mentioned, MLM does not have the assumption of sphericity.

Since bootstraps are used, all p-values are approximated for all relevant statistics (like F [1] or t). Since the MLM approach is used, to calculate the bootstrapped statistics for hypothesis testing the non-parametric model based bootstrap [109] is used. The null-model that is used is the intercept only model with included random intercepts [73]. In cases where confidence intervals are needed (i.e., effects sizes [9]), the parametric bootstrap is used, based on the full-model instead on the null-model to get more precise confidence intervals [73, 109]. The randomization while creating bootstraps is based on randomization of the residuals rather than the original data, as suggested in [87, 129].

The full model used was:

$$\begin{gathered} lmer(F1 \sim Tool + Technique + Tool : Technique + \\ (1|Participant) + (1|Tool : Participant) + (1|Technique : Participant), \\ data = SOCO.Dn, REML = FALSE) \end{gathered} \tag{8.1}$$

[1] There are two F statistics in this research — one is F1 which, is the measure calculated from Precision and Recall, and the other is the F statistic used to do hypothesis testing in ANOVA. In this thesis the first F statistic (calculated from Precision and Recall) is always referred to as F1 and the other is referred to as just F and it is often followed by brackets with two numbers which represent the degrees of freedom.

The null model was:

$$lmer(F1 \sim (1|Participant) + (1|Tool : Participant) + (1|Technique : Participant), \\ data = SOCO.Dn, REML = FALSE) \tag{8.2}$$

The full model for the simple effects analysis was:

$$lmer(F1 \sim ToolTechniqueCombo+ \\ (1|Participant) + (1|Tool : Participant) + (1|Technique : Participant), \\ data = SOCO.Dn, REML = FALSE) \tag{8.3}$$

and the null model was the same as written above. Recall that the term *Participant* represents one subset assignment (e.g., Dn-1, Dn-2, Dn-3, etc.) from the D1, D2, D3 or D4 group of assignments.

To perform the bootstrap hypothesis testing the function *boot* from package *boot* was used, and to get confidence intervals the function *boot.ci* from package *boot* was used or function *bootMer* from the *lme4* package. Models were compared using the *PBmodcomp* function from the *pbkrtest* package. The full list of packages used in R is presented in Appendix L.

8.2.2 Results for D1 assignments

D1 assignments are created from the A1 assignment in the SOCO dataset. The A1 assignment is a medium difficulty problem with small complexity. On average, files in A1 have 99.26 Lines of code (LOC). In Figure 8.3 the boxplot for the F1 scores is presented at each level of tool and technique in the D1 assignments. It can be observed that preprocessing techniques seem to have a stronger effect on textual versions of the tools than on the Java versions. For the Java version there is an indication that PPTs have some effect on Sherlock-java while they do not affect SIM-java and JPlag-java. The effect of PPTs on the textual versions is strongest on Sherlock-text but, rather than increasing the accuracy, the accuracy decreases, especially with the AllNOR technique. On the other hand, for JPlag-text and SIM-text the effect is positive, but it seems to be slightly stronger for SIM-text.

To confirm that there is an overall difference, ANOVA with bootstrap was performed. As explained in Section 8.2.1, the bootstrap method is used because the residuals are not normally distributed (Figure 8.4). The results of ANOVA with the F statistic and the p-values (original (column Pr(>|t|)) and bootstrapped (column p.boot)) are presented in Table 8.5. The result shows that the bootstrapped p-value (p_b) is less than 0.01 for the main effects (tool and technique) and for the interaction effect (tool $\times$ technique), which means that there is an overall difference. A significant interaction effect indicates that for some techniques there was a different effect depending on which tool was used alongside. That the interaction effect is significant is also confirmed by comparing interaction model and the model using only the main effects (i.e., tool

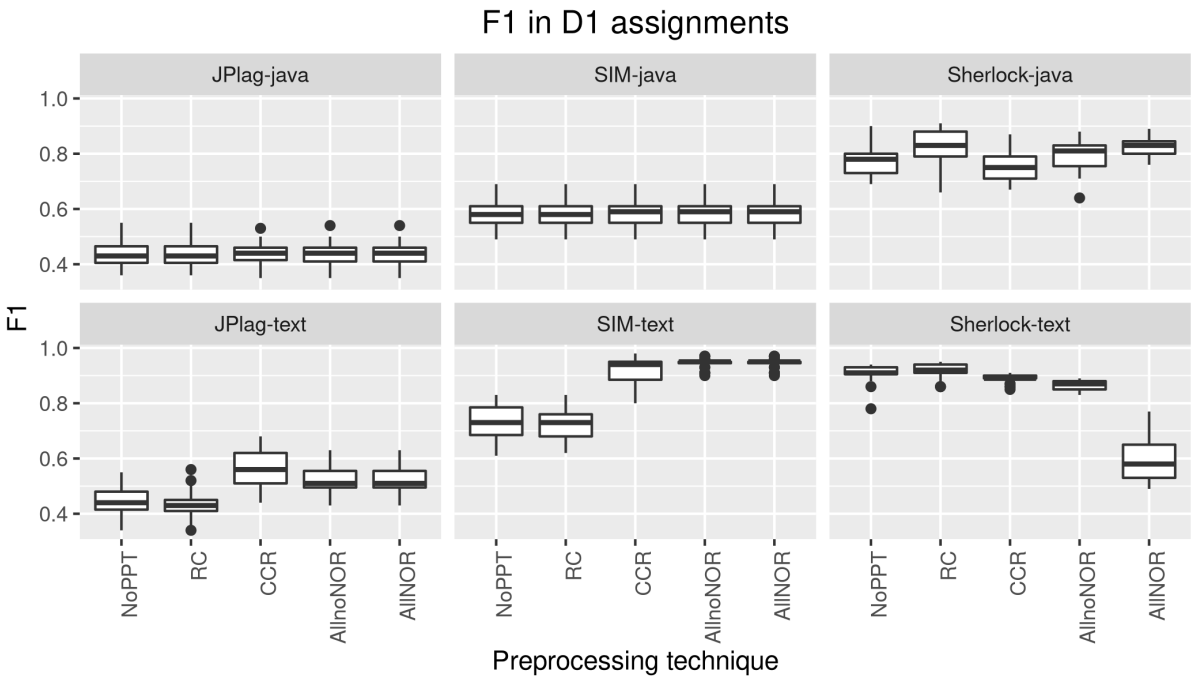


Figure 8.3: F1 score for SOCO D1 assignment with 3*IQR

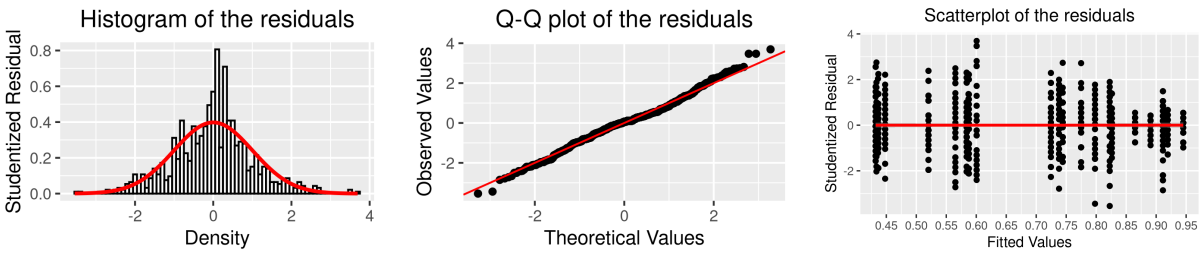


Figure 8.4: D1 assignments - residuals

Table 8.5: ANOVA results for SOCO D1

	Sum Sq	Mean Sq	NumDF	DenDF	F value	Pr(>F)	p.boot
Tool	4.07	0.814	5	155	818.0	0.0000	0.0001
Technique	0.34	0.085	4	124	85.9	0.0000	0.0001
Tool:Technique	**3.95**	**0.198**	**20**	**620**	**198.7**	**0.0000**	**0.0001**

and technique) whereby $\chi^2(20) = 1367.779$, $p_b < 0.01$ (more details are available in Table H.1).

Since the overall difference was confirmed, planned comparisons (contrasts) were performed. Table 8.6 presents the results for the defined contrasts. Since there is a difference in interaction it does not make sense to analyse the main effects, since they are suppressed by the interaction, but it is interesting to note that, if the interaction is ignored, there is still a difference for all stated contrasts in tool and technique factors except for the contrast measuring the difference between single and combo techniques. Although this information is not strictly valid, since there is an interaction between tool and techniques, it still gives an indication that the differences hypothesised in H1 and H2 might be significant.

To visually display the interaction, the mean comparison graph is presented in Figure 8.5. Although the interaction can be seen in Figure 8.3 it is more clear in Figure 8.5, since all values are presented on the same graph. In Figure 8.5 one can clearly see that the lines are not parallel, which is an indication that an interaction effect exists, of course the objective confirmation that the observed interaction is significant is done by using statistical tests (Table 8.5).

Figure 8.5 indicates that techniques have an effect on accuracy for all textual tools and for the tool Sherlock-java, while there is no change in accuracy for JPlag-java and SIM-java. The change, when present, is mostly positive, which means that using a PPT increases the accuracy. Although the effect is mostly positive there is a high negative effect on accuracy when techniques are used with Sherlock-text.

By looking at Figure 8.5 it can be observed that some techniques have a reverse effect depending on which tool they are used and/or to what technique they are compared. For example, CCR (compared to when no technique is used) has a negative effect on Sherlock-java while there is a positive effect on JPlag-text and SIM-text. At the same time, when CCR is combined with the RC and the NOR techniques (in the AllNOR) the effect is positive in comparison to when no technique is used for Sherlock-java.

Similarly, while the AllNOR technique has a positive effect (in comparison when no technique is used) for JPlag-text and SIM-text, it has a negative effect when compared to the CCR technique for JPlag-text. When the AllNOR technique is compared to the CCR technique for SIM-text this has a positive effect, but for JPlag-text this has a negative effect. This means that when using JPlag-text it would be better to just use the CCR technique rather than the AllNOR combination technique, at least on the SOCO D1 assignment.

Analysing the graphs only visually can be informative, but it can also be misleading. Thus,

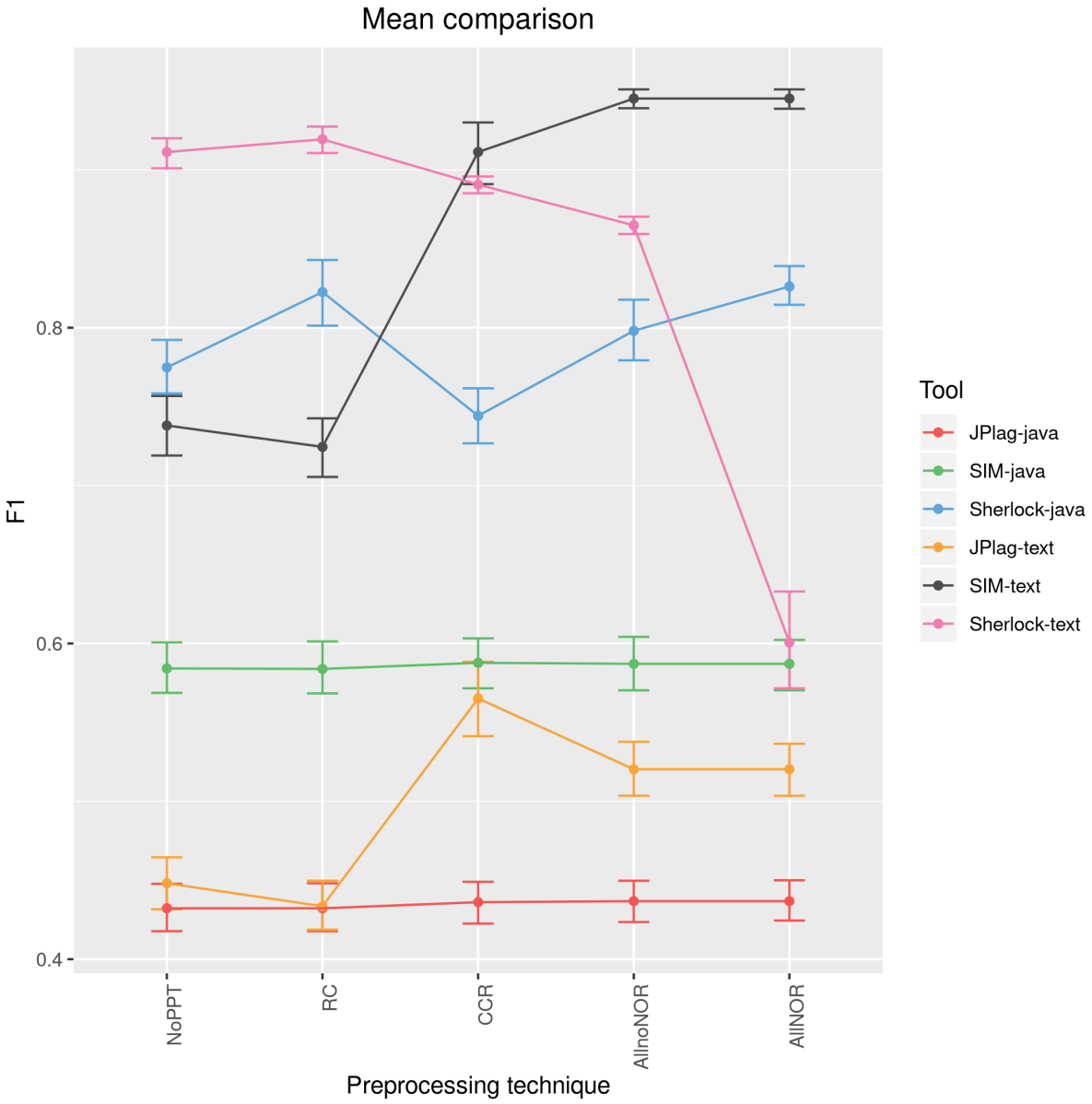


Figure 8.5: F1 mean comparison for SOCO D1

some differences might seem important when inspected visually but are not in the end significant when tested statistically (or vice versa). To verify the differences and to interpret the graph correctly the interaction effect is analysed further using planned contrasts and simple effects analysis (as mentioned in Section 8.1.2).

The results of the planned comparison is presented in Table 8.6 and the corresponding effect sizes are presented in Table I.1. The results of the simple effects analysis are presented in Table 8.7 and the corresponding effect sizes are presented in Table I.2. From 20 interaction contrasts, 14 contrasts have a significant difference with $p_b < 0.01$, although most p_b values were less than 0.0001, the result are considered significant if p_b is less or equal to 0.01. From 24 comparisons in the simple effects analysis, 11 comparisons had a significant result with $p_b < 0.01$. Note, there are 29 comparisons in the simple effects analysis, but only comparisons including techniques at each level of tool are counted.

Although using planned comparisons and the simple effects analysis limits the number of comparisons, it is out of the scope of this research to do the individual interpretation for all of the comparisons. Instead, interpretation is done in groups and only for comparisons that are most interesting for this research, which means that comparisons that help answer the research question and help confirm the stated hypothesis are the priority.

Table 8.6: Contrasts results for SOCO D1

Name	Estimate	SE	df	t.value	Pr(>|t|)	p.boot
(Intercept)	0.667	0	31	228.7363	0.0000	**0.0001**
Tool.TextvsJava	-0.062	0	155	-23.7553	0.0000	**0.0001**
TT.SHvsOthers	-0.054	0	155	-20.6218	0.0000	**0.0001**
TJ.SHvsOthers	-0.094	0	155	-35.9487	0.0000	**0.0001**
TT.SIMvsJPlag	-0.178	0	155	-39.1278	0.0000	**0.0001**
TJ.SIMvsJPlag	-0.076	0	155	-16.6443	0.0000	**0.0001**
NoPPTvsPPT	0.005	0	124	8.8276	0.0000	**0.0001**
SinglevsCombo	0.001	0	124	0.5975	0.5513	0.8008
RCvsCCR	0.018	0	124	10.8250	0.0000	**0.0001**
AnoNvsAN	-0.020	0	124	-11.6699	0.0000	**0.0001**
Tool.TextvsJava:NoPPTvsPPT	-0.003	0	620	-5.4427	0.0000	**0.0001**
TT.SHvsOthers:NoPPTvsPPT	0.013	0	620	25.1444	0.0000	**0.0001**
TJ.SHvsOthers:NoPPTvsPPT	-0.001	0	620	-2.5992	0.0096	**0.0073**
TT.SIMvsJPlag:NoPPTvsPPT	-0.008	0	620	-9.1208	0.0000	**0.0001**
TJ.SIMvsJPlag:NoPPTvsPPT	0.000	0	620	0.1080	0.9140	0.5662
Tool.TextvsJava:SinglevsCombo	0.005	0	620	4.0800	0.0001	0.0221
TT.SHvsOthers:SinglevsCombo	0.041	0	620	35.5109	0.0000	**0.0001**
TJ.SHvsOthers:SinglevsCombo	-0.004	0	620	-3.8591	0.0001	**0.0004**
TT.SIMvsJPlag:SinglevsCombo	-0.027	0	620	-13.2677	0.0000	**0.0001**
TJ.SIMvsJPlag:SinglevsCombo	0.000	0	620	0.1611	0.8721	0.5987
Tool.TextvsJava:RCvsCCR	-0.030	0	620	-18.3619	0.0000	**0.0001**
TT.SHvsOthers:RCvsCCR	0.031	0	620	19.1510	0.0000	**0.0001**
TJ.SHvsOthers:RCvsCCR	0.014	0	620	8.3837	0.0000	**0.0001**
TT.SIMvsJPlag:RCvsCCR	-0.014	0	620	-4.8688	0.0000	**0.0001**
TJ.SIMvsJPlag:RCvsCCR	0.000	0	620	0.0000	1.0000	0.5130
Tool.TextvsJava:AnoNvsAN	0.024	0	620	14.8934	0.0000	**0.0001**
TT.SHvsOthers:AnoNvsAN	0.044	0	620	26.9264	0.0000	**0.0001**
TJ.SHvsOthers:AnoNvsAN	-0.005	0	620	-2.8603	0.0044	**0.0034**
TT.SIMvsJPlag:AnoNvsAN	0.000	0	620	0.0000	1.0000	0.5259
TJ.SIMvsJPlag:AnoNvsAN	0.000	0	620	0.0000	1.0000	0.5265

Note:
TT - ToolText, TJ - ToolJava, SH - Sherlock, AnoN - AllnoNOR, AN - AllNOR

Table 8.7: Simple effect analysis result for SOCO D1

Name	Estimate	SE	df	t.value	Pr(>|t|)	p.boot
(Intercept)	0.667	0	31.0	228.7363	0.0000	**0.0001**
TextvsJava	-0.062	0	155.0	-23.7553	0.0000	**0.0001**
TT.SHvsOthers	-0.054	0	155.0	-20.6218	0.0000	**0.0001**
TT.SIMvsJPlag	-0.178	0	155.0	-39.1278	0.0000	**0.0001**
TJ.SHvsOthers	-0.094	0	155.0	-35.9487	0.0000	**0.0001**
TJ.SIMvsJPlag	-0.076	0	155.0	-16.6443	0.0000	**0.0001**
TT.SH.NoPPTvsPPT	-0.018	0	743.6	-14.5161	0.0000	**0.0001**
TT.SH.SinglevsCombo	-0.086	0	743.6	-30.2497	0.0000	**0.0001**
TT.SH.RCvsCCR	-0.014	0	743.6	-3.5650	0.0004	**0.0003**
TT.SH.AllnoNORvsAllNOR	-0.132	0	743.6	-32.8057	0.0000	**0.0001**
TT.JPlag.NoPPTvsPPT	0.012	0	743.6	9.7027	0.0000	**0.0001**
TT.JPlag.SinglevsCombo	0.010	0	743.6	3.6821	0.0002	0.0453
TT.JPlag.RCvsCCR	0.066	0	743.6	16.3428	0.0000	**0.0001**
TT.JPlag.AllnoNORvsAllNOR	0.000	0	743.6	0.0000	1.0000	0.5204
TT.SIM.NoPPTvsPPT	0.029	0	743.6	22.5342	0.0000	**0.0001**
TT.SIM.SinglevsCombo	0.064	0	743.6	22.3474	0.0000	**0.0001**
TT.SIM.RCvsCCR	0.093	0	743.6	23.1923	0.0000	**0.0001**
TT.SIM.AllnoNORvsAllNOR	0.000	0	743.6	0.0000	1.0000	0.5241
TJ.SH.NoPPTvsPPT	0.005	0	743.6	3.5974	0.0003	0.0580
TJ.SH.SinglevsCombo	0.014	0	743.6	5.0416	0.0000	**0.0018**
TJ.SH.RCvsCCR	-0.039	0	743.6	-9.7336	0.0000	**0.0001**
TJ.SH.AllnoNORvsAllNOR	0.014	0	743.6	3.4849	0.0005	0.0699
TJ.JPlag.NoPPTvsPPT	0.001	0	743.6	0.5067	0.6125	0.7739
TJ.JPlag.SinglevsCombo	0.001	0	743.6	0.4532	0.6506	0.7326
TJ.JPlag.RCvsCCR	0.002	0	743.6	0.4807	0.6309	0.7469
TJ.JPlag.AllnoNORvsAllNOR	0.000	0	743.6	0.0000	1.0000	0.5271
TJ.SIM.NoPPTvsPPT	0.000	0	743.6	0.3547	0.7229	0.7017
TJ.SIM.SinglevsCombo	0.001	0	743.6	0.2266	0.8208	0.6289
TJ.SIM.RCvsCCR	0.002	0	743.6	0.4807	0.6309	0.7522
TJ.SIM.AllnoNORvsAllNOR	0.000	0	743.6	0.0000	1.0000	0.5198

Note:
TT - ToolText, TJ - ToolJava, SH - Sherlock

Differences between no preprocessing and preprocessing

The first interaction term looks at the effect of techniques (i.e., all PPTs combined) relative to when no PPT is used when comparing textual (i.e., all textual tools combined) and Java versions (i.e., all Java tools combined) of the tools. This contrast is significant and this result tells us that the change in accuracy found when the PPT technique is used (compared to when no technique is used) is different for Java and textual tools. In other words, Java and textual tools respond significantly different from using PPTs.

The next four contrasts look at the effect of techniques (i.e., all PPTs combined) relative to when no PPT is used comparing: Sherlock-text vs. other text tools (i.e., SIM-text and JPlag-text combined); Sherlock-java vs. other Java tools (i.e., SIM-java and JPlag-java combined); SIM-text vs. JPlag-text, and SIM-java vs. JPlag-java.

For the four contrasts the interaction difference was significant except for SIM-java vs. JPlag-java. This means that using PPTs (in comparison to when no technique is used) effects the SIM-java and JPlag-java tools in the same way.

Based on the four contrasts it is safe to say that in most cases the tools' accuracy changes differently when PPTs is used in comparison when no technique is used. This indicates, but does not confirm, that using PPTs makes a difference to an individual tool. By examining the interaction graphs (Figure J.1) it is visible that the accuracy increases except for Sherlock-text.

To check H1, if there exists a difference in plagiarism detection accuracy between at least one PPT and when no PPT is used, a simple effects analysis was performed. The results of the simple effects analysis for SOCO D1 are presented in Table 8.7 and the corresponding effect sizes in Table I.2. The analysis confirmed that using PPTs made a significant difference (at $p_b < 0.01$) in comparison when no PPT is used for textual versions of the three tools. On the other hand, for JPlag-java ($p_b = 0.78$), SIM-java ($p_b = 0.70$) and Sherlock-java ($p_b = 0.05$), the difference is not big enough to be considered significant, although it can not be said that there is no difference and there is always the possibility of a type 2 error (the difference is there but the test was not able to detect the difference). For tools where the difference was significant, techniques have a positive effect on SIM-text and JPlag-text but a negative effect on Sherlock-text.

Based on the presented results it can be stated that the H1 hypothesis is confirmed on the SOCO D1 assignments.

Differences between different preprocessing techniques

In the previous analysis it was confirmed that using PPTs significantly increases the accuracy for most tools and that the increase is different between most tools. The problem with the previous analysis is that all techniques were put together, and if one technique had a positive effect and another technique a negative effect the effects could cancel each other. To overcome the problem, the PPTs were analysed further in comparison to each other. Since contrasts were

used and not post-hoc tests [2] not every technique was compared to every other technique, rather first the single techniques (i.e., RC and CCR combined) were compared to combo techniques (i.e., AllnoNOR and AllNOR combined) and then the two single techniques were compared to each other and the two combo techniques to each other. The contrasts of the tools are the same as before.

Regarding the interaction contrasts, as can be seen from Table 8.6 10 contrasts showed significant differences at $p_b < 0.01$, and 5 did not. It is no surprise that every contrast where SIM-java and JPlag-java were compared was non-significant, as the means of those tools across different techniques are parallel (Figure 8.5). Also for both tools the means are more or less the same (Figure 8.5). This leads to the conclusion that using PPTs has no effect on SIM-java or JPlag-java.

The contrasts that compare single and combo techniques show a significant result ($p_b < 0.01$) except for SIM-java vs. JPlag-java and Text vs. Java. The contrast that compared together all textual and all Java tools had $p_b = 0.02$, which is considered non-significant, meaning text and Java tools respond the same when single techniques are compared to combo techniques. But by looking at the interaction graph (Figure J.1 and Figure J.2) one can see that the lines go in the opposite direction and by looking at 8.5 it is more likely that there is a difference but that the test was too weak to consider it significant. In addition, significance can be looked at p-value is less than 0.05. and in this case this interaction is significant. Because of all of this, it is open to debate whether the comparison should be considered significant or not.

The 10 comparisons that compare single techniques (RC vs. CCR) and the comparisons that compare combo techniques (AllnoNOR vs. AllNOR) are mostly significant, $p_b < 0.01$. Non-significant results were only for contrasts comparing SIM-java vs. JPlag-java for both single and combo comparisons, and for SIM-text vs. JPlag-text when comparing combo techniques.

From the 10 contrasts that compare different techniques and that are significant it can be concluded that accuracy changes are different for different tools when comparing different techniques in most cases. Again, as with the previous analysis (no PPT vs. PPT), this indicates (but does not confirm) that there is a difference in using different techniques with a specific tool. To check H2, if there exists a difference in plagiarism detection accuracy between at least two different PPTs, a simple effects analysis was used.

The simple effects analysis of single techniques compared to combo techniques showed a significant difference in accuracy with $p_b < 0.01$ for Sherlock-text, SIM-text and Sherlock-java. The difference was not significant for SIM-java ($p_b = 0.63$), JPlag-java ($p_b = 0.74$) and JPlag-text ($p_b = 0.05$). The result for JPlag-text could be debated if this is considered significant for the same reasons as the result before (text vs. Java), but since all other results had a much lower p-value, the result for JPlag-text can be considered correct. From the interaction graphs (Figure J.1 and Figure J.2) it can be seen that combo techniques show an increase in accuracy

[2]Contrasts and simple effects analysis were chosen over post-hoc because of the greater test strengths, since fewer comparisons are made, as described in 8.1.2. Also, the analysis is more focused and faster in execution in R.

in comparison to single techniques for all tools except Sherlock-text where a strong decrease is observed.

A simple effects analysis of RC vs. CCR techniques shows that the CCR technique increases the accuracy except in the cases of Sherlock-text and Sherlock-java. The simple effects analysis for the combo techniques (AllnoNOR vs. AllNOR) showed that there is no difference between the two techniques for SIM and JPlag tools (both versions had $p_b = 0.5$) and for Sherlock-java ($p_b = 0.07$) but there is a significant difference for Sherlock-text ($p_b < 0.01$).

It is interesting that, although Sherlock-java, SIM-java and JPlag-java have not a significant difference between the combo techniques, there is a significant interaction ($p_b < 0.01$ for 'TJ.SHvs.Others:AnoNvs.AN') between Sherlock-java and the other Java tools (i.e., SIM-java and JPlag-java combined) when the combo techniques are compared. This means that Sherlock-java responded significantly different than the other two tools when comparing combo techniques, although Sherlock-java individually has no significant change in accuracy when comparing combo techniques. In this case, adding the NOR technique in the combination AllNOR has a positive effect on Sherlock-java while SIM-java and JPlag-java were not affected at all. This situation shows how techniques can make a distinction between the tools even though at first glance everything seems to be the same and that jumping to the conclusion that *adding NOR technique to Sherlock-java makes no sense* is wrong.

Based on the presented results it can be stated that the H2 hypothesis is confirmed for the SOCO D1 assignments.

8.2.3 Results for D2 assignments

The D2 assignments are created from the A2 assignments in the SOCO dataset. The A2 assignments are considered a medium difficulty problem with large complexity, as explained in Section 6.2. On average, files in A2 have 107.87 LOC. In Figure 8.6 the box-plot for the F1 scores is presented at each level of tool and technique for the D2 assignments. It can be observed that preprocessing techniques seem to have a stronger effect on textual versions of the tools than on the Java versions, which is the same as for D1. For the Java version, it seems that the techniques have no effect on any Java tool. The effect of PPTs on the textual versions is strongest on Sherlock-text, and again, as for D1, there is a decrease rather than an increase in accuracy. JPlag-text and SIM-text have (as in D1) a positive effect with a slightly stronger effect for SIM-text.

To confirm that there is an overall difference ANOVA with bootstrap was performed for the same reasons as for D1. In Figure 8.7, a histogram, Q-Q plot and scatter plot of the residuals are presented. The results of ANOVA with the F statistic and the p-values (original and bootstrapped) are presented in Table 8.8. The result shows that the p_b value is less than 0.01 for the main effects (tool and technique) and for the interaction effect. The model comparison is presented in Figure H.2 where for this dataset $\chi^2(20) = 1272.371$, $p_b < 0.01$. To follow up, contrast comparisons and simple effects analysis was performed as it was done for the D1 assignments.

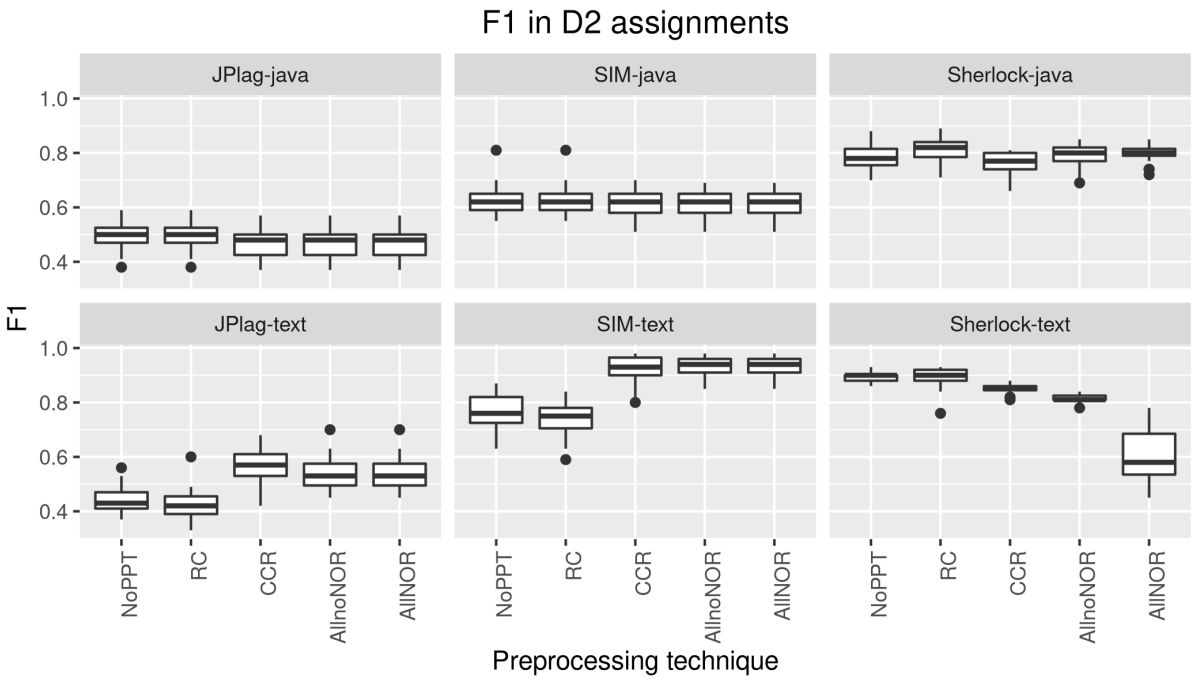


Figure 8.6: F1 score for SOCO D2 assignment with 3*IQR

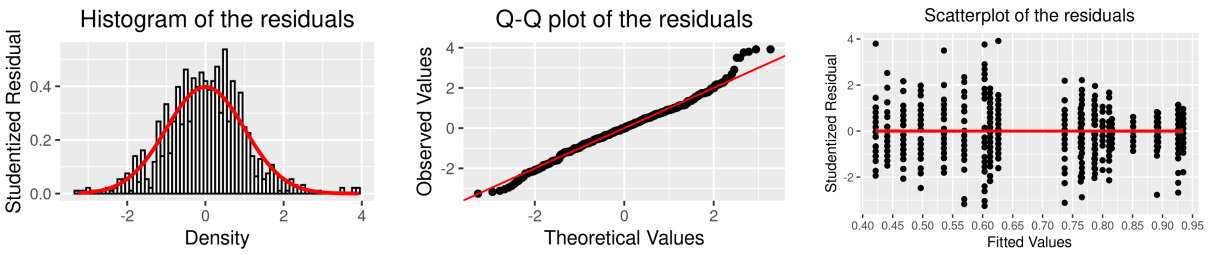


Figure 8.7: D2 assignments - residuals

Table 8.8: ANOVA results for SOCO D2

	Sum Sq	Mean Sq	NumDF	DenDF	F value	Pr(>F)	p.boot
Tool	3.40	0.679	5	155	675.5	0.0000	0.0001
Technique	0.23	0.059	4	744	58.2	0.0000	0.0001
Tool:Technique	**3.39**	**0.169**	**20**	**744**	**168.5**	**0.0000**	**0.0001**

The visual representation of the means in the form of a comparison graph is presented in Figure 8.8.

Figure 8.8 indicates that techniques have an effect on accuracy for all textual tools and the Sherlock-java tool while there is a small change in accuracy for JPlag-java and SIM-java. The change is positive for SIM-text and JPlag-text and negative for the other tools, especially for Sherlock-text. By looking at Figure 8.8 it can be observed that some techniques have a reverse effect, depending on which tool they are using and/or to what technique they are compared. An interesting observation in this graph is that the CCR technique has such a negative effect on Sherlock-java that when combined with RC the accuracy is lower than without techniques even though there is a positive effect of RC. In general, Figure 8.8 looks a lot like the mean comparison graph for the D1 assignment. This is not unexpected since D1 and D2 are both based on the A assignments from the SOCO dataset which represent the same problem only with different complexity.

Analysing the graphs only visually, as stated when the D1 assignment was analysed, can be informative but it can also be misleading. Therefore, the results of the planned comparison are presented in Table 8.9 and the corresponding effect sizes are presented in Table I.3. The results of the simple effects analysis are presented in Table 8.10 and the corresponding effect sizes are presented in Table I.4. From 20 interaction contrasts, 12 have a significant difference with $p_b < 0.01$, and although most p_b values were less than 0.0001 the result is considered significant if the p_b is less or equal to 0.01. From 24 comparisons in the simple effects analysis 14 comparisons had a significant result with $p_b < 0.01$.

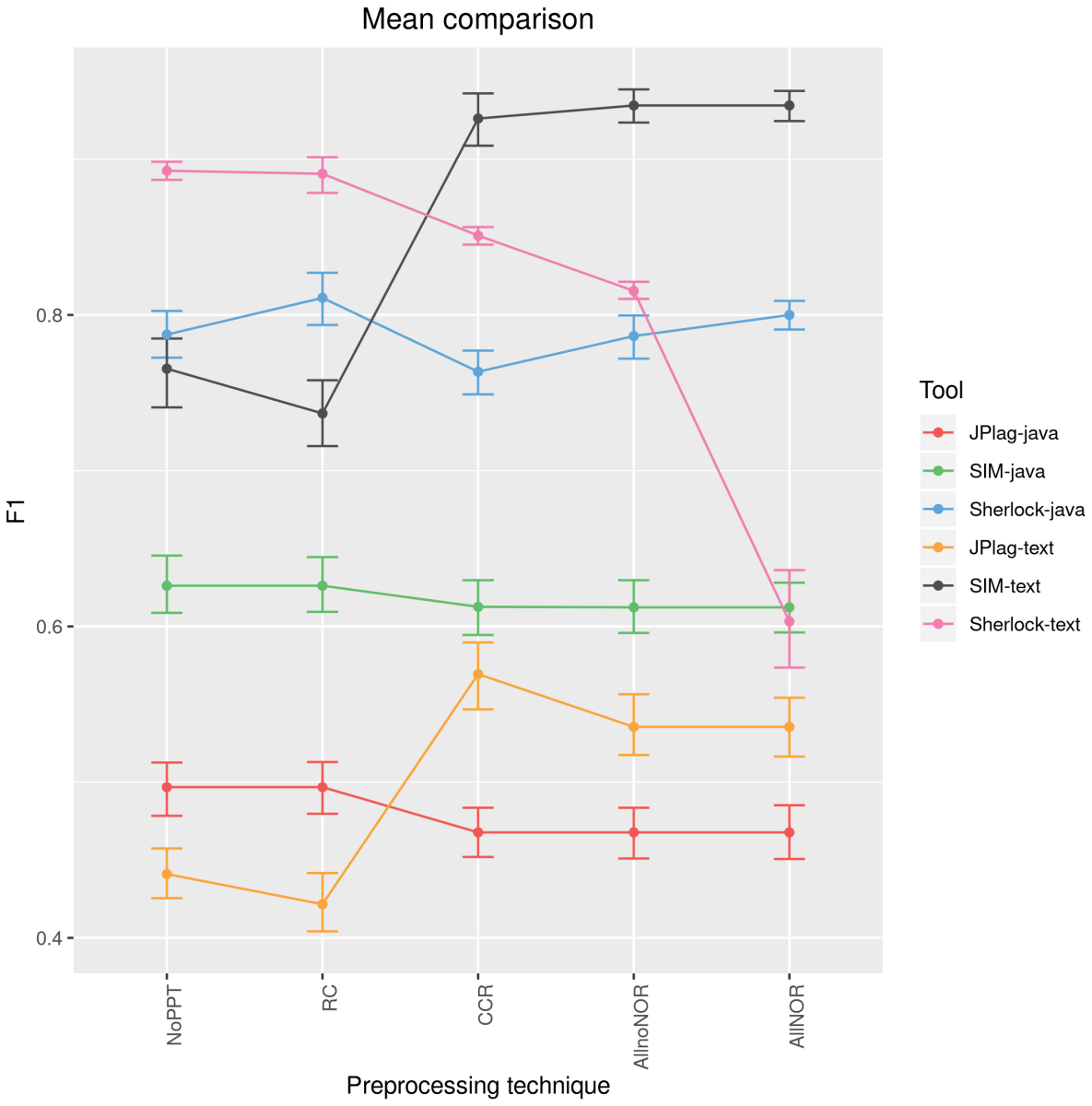


Figure 8.8: F1 mean comparison for SOCO D2

Table 8.9: Contrasts results for SOCO D2

Name	Estimate	SE	df	t.value	Pr(>|t|)	p.boot
(Intercept)	0.676	0	31	192.3335	0.0000	0.0395
Tool.TextvsJava	-0.047	0	155	-18.1540	0.0000	**0.0001**
TT.SHvsOthers	-0.044	0	155	-16.7051	0.0000	**0.0001**
TJ.SHvsOthers	-0.080	0	155	-30.8469	0.0000	**0.0001**
TT.SIMvsJPlag	-0.179	0	155	-39.7728	0.0000	**0.0001**
TJ.SIMvsJPlag	-0.069	0	155	-15.3500	0.0000	**0.0001**
NoPPTvsPPT	0.002	0	744	3.8630	0.0001	0.0284
SinglevsCombo	-0.003	0	744	-2.4399	0.0149	**0.0072**
RCvsCCR	0.017	0	744	10.5152	0.0000	**0.0001**
AnoNvsAN	-0.017	0	744	-10.0737	0.0000	**0.0001**
Tool.TextvsJava:NoPPTvsPPT	-0.004	0	744	-7.6175	0.0000	**0.0001**
TT.SHvsOthers:NoPPTvsPPT	0.013	0	744	25.4588	0.0000	**0.0001**
TJ.SHvsOthers:NoPPTvsPPT	-0.001	0	744	-2.4202	0.0157	0.0101
TT.SIMvsJPlag:NoPPTvsPPT	-0.004	0	744	-4.7742	0.0000	**0.0001**
TJ.SIMvsJPlag:NoPPTvsPPT	-0.001	0	744	-1.2719	0.2038	0.1027
Tool.TextvsJava:SinglevsCombo	0.000	0	744	0.1966	0.8442	0.6193
TT.SHvsOthers:SinglevsCombo	0.039	0	744	33.4073	0.0000	**0.0001**
TJ.SHvsOthers:SinglevsCombo	-0.003	0	744	-2.4052	0.0164	**0.0087**
TT.SIMvsJPlag:SinglevsCombo	-0.016	0	744	-7.8312	0.0000	**0.0001**
TJ.SIMvsJPlag:SinglevsCombo	-0.002	0	744	-0.9213	0.3572	0.1851
Tool.TextvsJava:RCvsCCR	-0.032	0	744	-19.6404	0.0000	**0.0001**
TT.SHvsOthers:RCvsCCR	0.035	0	744	21.1123	0.0000	**0.0001**
TJ.SHvsOthers:RCvsCCR	0.004	0	744	2.6493	0.0082	0.2721
TT.SIMvsJPlag:RCvsCCR	-0.010	0	744	-3.6539	0.0003	**0.0007**
TJ.SIMvsJPlag:RCvsCCR	-0.004	0	744	-1.3596	0.1744	0.0955
Tool.TextvsJava:AnoNvsAN	0.019	0	744	11.4474	0.0000	**0.0001**
TT.SHvsOthers:AnoNvsAN	0.035	0	744	21.5211	0.0000	**0.0001**
TJ.SHvsOthers:AnoNvsAN	-0.002	0	744	-1.3737	0.1700	0.0945
TT.SIMvsJPlag:AnoNvsAN	0.000	0	744	0.0000	1.0000	0.5284
TJ.SIMvsJPlag:AnoNvsAN	0.000	0	744	0.0000	1.0000	0.5209

Note:
TT - ToolText, TJ - ToolJava, SH - Sherlock, AnoN - AllnoNOR, AN - AllNOR

Table 8.10: Simple effects analysis result for SOCO D2

Name	Estimate	SE	df	t.value	Pr(>|t|)	p.boot
(Intercept)	0.676	0	31	192.3335	0.0000	0.0437
TextvsJava	-0.047	0	155	-18.1540	0.0000	**0.0001**
TT.SHvsOthers	-0.044	0	155	-16.7051	0.0000	**0.0001**
TT.SIMvsJPlag	-0.179	0	155	-39.7728	0.0000	**0.0001**
TJ.SHvsOthers	-0.080	0	155	-30.8469	0.0000	**0.0001**
TJ.SIMvsJPlag	-0.069	0	155	-15.3500	0.0000	**0.0001**
TT.SH.NoPPTvsPPT	-0.020	0	744	-16.1001	0.0000	**0.0001**
TT.SH.SinglevsCombo	-0.081	0	744	-28.3533	0.0000	**0.0001**
TT.SH.RCvsCCR	-0.020	0	744	-4.9271	0.0000	**0.0001**
TT.SH.AllnoNORvsAllNOR	-0.106	0	744	-26.3578	0.0000	**0.0001**
TT.JPlag.NoPPTvsPPT	0.015	0	744	11.7046	0.0000	**0.0001**
TT.JPlag.SinglevsCombo	0.020	0	744	7.0246	0.0000	**0.0001**
TT.JPlag.RCvsCCR	0.074	0	744	18.3463	0.0000	**0.0001**
TT.JPlag.AllnoNORvsAllNOR	0.000	0	744	0.0000	1.0000	0.5334
TT.SIM.NoPPTvsPPT	0.024	0	744	18.4563	0.0000	**0.0001**
TT.SIM.SinglevsCombo	0.052	0	744	18.0996	0.0000	**0.0001**
TT.SIM.RCvsCCR	0.095	0	744	23.5138	0.0000	**0.0001**
TT.SIM.AllnoNORvsAllNOR	0.000	0	744	0.0000	1.0000	0.5267
TJ.SH.NoPPTvsPPT	0.001	0	744	0.4434	0.6576	0.7419
TJ.SH.SinglevsCombo	0.003	0	744	1.0480	0.2950	0.9747
TJ.SH.RCvsCCR	-0.024	0	744	-5.8885	0.0000	**0.0001**
TJ.SH.AllnoNORvsAllNOR	0.007	0	744	1.6824	0.0929	0.6666
TJ.JPlag.NoPPTvsPPT	-0.004	0	744	-3.4202	0.0007	**0.0002**
TJ.JPlag.SinglevsCombo	-0.007	0	744	-2.5492	0.0110	**0.0053**
TJ.JPlag.RCvsCCR	-0.015	0	744	-3.6052	0.0003	**0.0004**
TJ.JPlag.AllnoNORvsAllNOR	0.000	0	744	0.0000	1.0000	0.5146
TJ.SIM.NoPPTvsPPT	-0.002	0	744	-1.6214	0.1054	0.0547
TJ.SIM.SinglevsCombo	-0.004	0	744	-1.2463	0.2130	0.1107
TJ.SIM.RCvsCCR	-0.007	0	744	-1.6824	0.0929	0.0509
TJ.SIM.AllnoNORvsAllNOR	0.000	0	744	0.0000	1.0000	0.5269

Note:
TT - ToolText, TJ - ToolJava, SH - Sherlock

Differences between no preprocessing and preprocessing

The first five interaction terms look at the effects of techniques (i.e., all PPTs combined) relative to when no PPT is used when comparing textual (i.e., all textual tools combined) vs. Java versions (i.e., all Java tools combined) of the tools, Sherlock-text vs. other text tools (i.e., SIM-text and JPlag-text combined), Sherlock-java vs. other Java tools (i.e., SIM-java and JPlag-java combined), SIM-text vs. JPlag-text, and SIM-java vs. JPlag-java.

For the five contrasts the interaction differences are significant except for SIM-java vs. JPlag-java. This means that using PPTs (in comparison to when no technique is used) affects the SIM-java and JPlag-java tools in the same way.

Based on the five contrasts it is safe to say that in most cases the tool accuracy changes differently when PPTs is used in comparison to when no technique is used. This indicates but does not confirm that using PPTs makes a difference to an individual tool. By examining the interaction graphs (Figure J.3) it is apparent that the accuracy increases for Sherlock-java, SIM-text, JPlag-text, and decreases for Sherlock-text, SIM-java, and JPlag-java.

To check H1, if there exists a difference in plagiarism detection accuracy between at least one PPT and when no PPT is used, a simple effects analysis was performed. The results of the simple effects analysis for SOCO D2 are presented in Table 8.10 and the corresponding effect sizes in Table I.4. The analysis confirmed that using PPTs made a significant difference (at $p_b < 0.01$) in comparison to when no PPT is used for Sherlock-text, JPlag-text, SIM-text, and JPlag-java. On the other hand, for Sherlock-java ($p_b = 0.74$) and SIM-java ($p_b = 0.05$) the difference is not big enough to be considered significant. For tools where the difference is significant, techniques have a positive effect on all tools except JPlag-java.

Based on the presented results it can be stated that the H1 hypothesis is confirmed for the SOCO D2 assignments.

Differences between different preprocessing techniques

In the previous analysis of the SOCO D2 assignments it was confirmed that using PPTs significantly changes the accuracy for most tools and that the change is different between most tools. The problem with the previous analysis, as for the D1 assignments, is that all techniques were put together and if one technique had a positive effect and another technique a negative effect the effects could cancel each other. To overcome the problem the PPTs were analysed further in comparison to each other in the same way as for the D1 assignments.

Regarding the interaction contrasts, as can be seen from Table 8.9, 8 contrasts showed significant differences at $p_b < 0.01$ and 7 did not. It is no surprise that every contrast where SIM-java and JPlag-java were compared was non-significant as the means of those tools across different techniques are parallel (Figure 8.8). Also, for both tools the means are more or less the same (Figure 8.8). This leads to a conclusion that using PPTs has no effect on SIM-java or JPlag-java.

The contrasts that compare single and combo technique do not show a significant result (at $p_b < 0.01$) for SIM-java vs. JPlag-java ($p_b = 0.19$) and Text vs. Java ($p_b = 0.62$). The 10 comparisons that compare single techniques (RC vs. CCR) and the comparisons that compare combo techniques (AllnoNOR vs. AllNOR) are significant $p_b < 0.01$for 5 contrasts. Non-significant results are for contrasts comparing SIM-java vs. JPlag-java and Sherlock-java vs. other Java tools for both single and combo comparisons, and for SIM-text vs. JPlag-text when comparing combo techniques.

From the 8 contrasts that compare different techniques and that are significant it can be concluded that accuracy changes are different for different tools when comparing different techniques mostly for textual versions of the tools. Again, as with the previous analysis (no PPT vs. PPT) this indicates but does not confirm that there is a difference in using different techniques on an individual tool. To check H2, if there exists a difference in plagiarism detection accuracy between at least two different PPTs, a simple effects analysis was used.

The simple effects analysis of single techniques compared to combo techniques showed a significant difference in accuracy with $p_b < 0.01$ for Sherlock-text, SIM-text, JPlag-text, and JPlag-java. The difference was not significant for SIM-java ($p_b = 0.11$) and Sherlock-java ($p_b = 0.97$). From the interaction graphs (Figure J.3 and Figure J.4) it can be seen that combo techniques show an increase in accuracy in comparison to single techniques for SIM-text, JPlag-text and Sherlock-java and a decrease for the others, which means that using combo techniques was successful 50% of the time.

A simple effects analysis of RC vs. CCR techniques shows that the CCR technique decreases the accuracy for all tools except in case of SIM-text and JPlag-text. The simple effects analysis for the combo techniques (AllnoNOR vs. AllNOR) shows that there is no difference between the two techniques for the SIM and JPlag tools (both versions had $p_b = 0.5$) and for Sherlock-java ($p_b = 0.66$), but that there is a significant difference for Sherlock-text ($p_b < 0.01$).

Based on the presented results it can be stated that the H2 hypothesis is confirmed for the SOCO D2 assignments.

8.2.4 Results for D3 assignments

D3 assignments are created from the B1 assignments in the SOCO dataset. B1 assignments relate to a medium difficulty problem with small complexity as did the A1 assignment. On average, files in B1 have 90.86 LOC, which is 9 lines fewer than the A1 assignments. In Figure 8.9 the boxplot for the F1 scores is presented at each level of tool and technique for the D3 assignments. It can be observed that preprocessing techniques seem to have a stronger effect on textual versions of the tools than on the Java versions, which is the same as for the D1 and D2 assignments. For the Java version there seems to be a small negative effect for SIM-java and JPlag-java and a positive effect for Sherlock-java. For the textual versions, the effect of PPTs was negative for Sherlock-text although it is not so strong as for the D1 and D2 assignments. JPlag-text and SIM-text have again a positive effect (as it was for D1 and D2) with a slightly

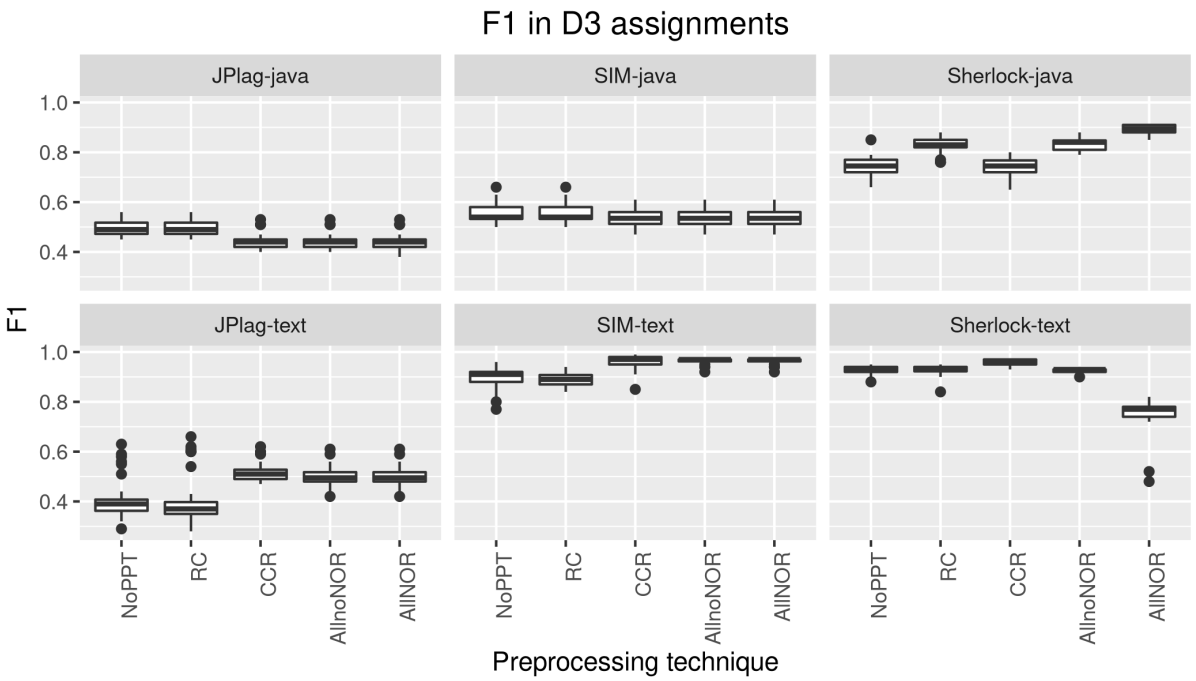


Figure 8.9: F1 score for SOCO D3 assignment with 3*IQR

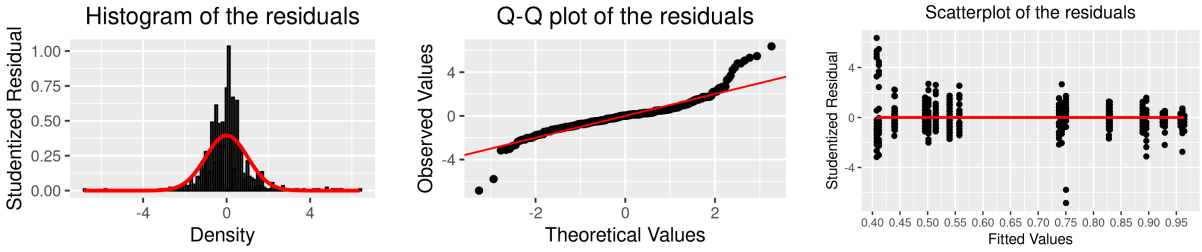


Figure 8.10: D3 assignments - residuals

stronger effect for JPlag-text which is different than for the D1 and D2 assignments.

To confirm that there is an overall difference, as with the first two assignments ANOVA with bootstrap was performed. In Figure 8.10, a histogram, Q-Q plot and scatter plot of the residuals are presented. The results of the ANOVA with the F statistic and the p-values (original and bootstrapped) are presented in Table 8.11. The result shows that p_b value is less than 0.01 for the main effects (tool and technique) and for the interaction effect. The model comparison is presented in Table H.3 where for this dataset $\chi^2(20) = 959.451$, $p_b < 0.01$. Next, contrast comparisons and a simple effects analysis were performed as for the D1 and D2 assignments. A visual representation of the means for the D3 assignments is presented in Figure 8.11.

Figure 8.11 indicates that techniques have an effect on accuracy for all tools except SIM-java, where the change is very small. The change is positive for SIM-text, JPlag-text, Sherlock-java, but negative for the other tools, especially for Sherlock-text. By looking at Figure 8.11, as it was for the first two assignments it can be observed that some techniques have a reverse effect depending on which tool they used and/or with what technique they are compared. In general, Figure 8.11 looks different from the mean comparison graph for the D1 and D2 assignments and

Table 8.11: ANOVA results for SOCO D3

	Sum Sq	Mean Sq	NumDF	DenDF	F value	Pr(>F)	p.boot
Tool	7.53	1.506	5	177	1636.9	0.0000	0.0001
Technique	0.07	0.017	4	122	18.4	0.0000	0.0001
Tool:Technique	**1.91**	**0.095**	**20**	**601**	**103.6**	**0.0000**	**0.0001**

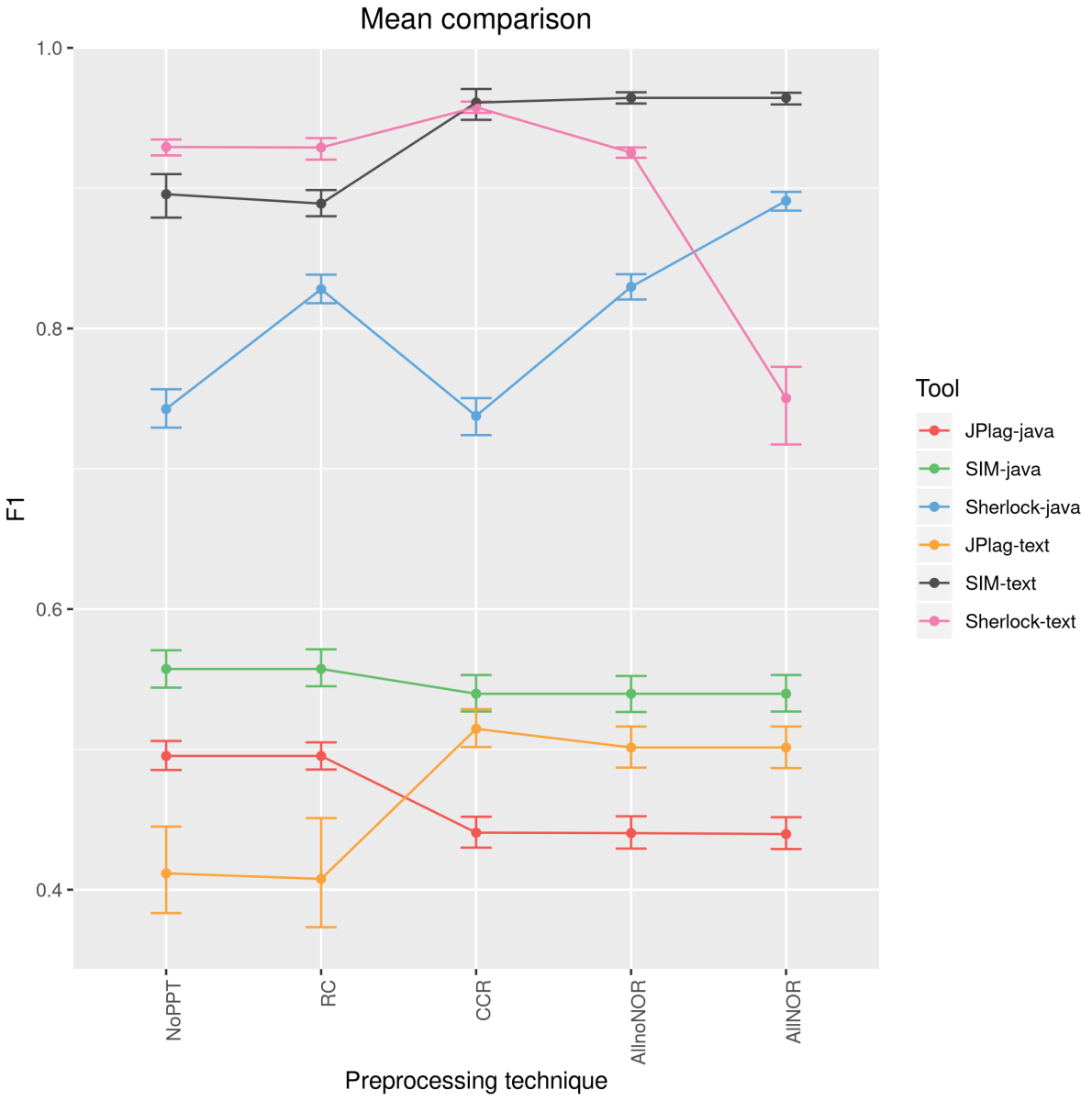


Figure 8.11: F1 mean comparison for SOCO D3

quite similar to the D4 assignment (described later). This is not unexpected since D1 and D2 are both based on the A assignments from the SOCO dataset and D3 and D4 are both based on the B assignments from the SOCO dataset.

Analysing the graphs only visually, as it was stated before, can be informative but it can also be misleading. Therefore, the results of the planned comparison for this assignment are presented in Table 8.12 and the corresponding effect sizes are presented in Table I.5. The results of the simple effects analysis for the D3 assignments are presented in Table 8.13 and the corresponding effect sizes are presented in Table I.6. From 20 interaction contrasts, 15 have a significant difference with $p_b < 0.01$, although for most the p_b values were less than 0.0001, and the result is considered significant if p_b is less or equal to 0.01 (as it was done for the D1 and D2 assignments). From 24 comparisons in the simple effects analysis, 16 comparisons have significant result with $p_b < 0.01$.

Table 8.12: Contrasts results for SOCO D3

Name	Estimate	SE	df	t.value	Pr(>|t|)	p.boot
(Intercept)	0.686	0	179.80	304.4271	0.0000	**0.0001**
Tool.TextvsJava	-0.081	0	177.07	-36.7241	0.0000	**0.0001**
TT.SHvsOthers	-0.066	0	177.07	-29.8239	0.0000	**0.0001**
TJ.SHvsOthers	-0.100	0	177.07	-45.5599	0.0000	**0.0001**
TT.SIMvsJPlag	-0.234	0	177.07	-61.2247	0.0000	**0.0001**
TJ.SIMvsJPlag	-0.042	0	177.07	-11.0611	0.0000	**0.0004**
NoPPTvsPPT	0.003	0	121.62	6.2394	0.0000	**0.0001**
SinglevsCombo	0.001	0	121.62	0.9822	0.3280	0.9919
RCvsCCR	0.004	0	121.62	2.1309	0.0351	0.4618
AnoNvsAN	-0.010	0	121.62	-5.4141	0.0000	**0.0001**
Tool.TextvsJava:NoPPTvsPPT	-0.002	0	600.89	-3.6599	0.0003	**0.0003**
TT.SHvsOthers:NoPPTvsPPT	0.007	0	600.89	12.9306	0.0000	**0.0001**
TJ.SHvsOthers:NoPPTvsPPT	-0.007	0	600.89	-14.0077	0.0000	**0.0001**
TT.SIMvsJPlag:NoPPTvsPPT	0.002	0	600.89	2.3510	0.0190	0.3766
TJ.SIMvsJPlag:NoPPTvsPPT	-0.003	0	600.89	-3.2077	0.0014	**0.0007**
Tool.TextvsJava:SinglevsCombo	0.006	0	600.89	4.9152	0.0000	**0.0019**
TT.SHvsOthers:SinglevsCombo	0.024	0	600.89	21.4181	0.0000	**0.0001**
TJ.SHvsOthers:SinglevsCombo	-0.016	0	600.89	-14.1435	0.0000	**0.0001**
TT.SIMvsJPlag:SinglevsCombo	0.000	0	600.89	0.1064	0.9153	0.5689
TJ.SIMvsJPlag:SinglevsCombo	-0.005	0	600.89	-2.4476	0.0147	**0.0086**
Tool.TextvsJava:RCvsCCR	-0.031	0	600.89	-19.3069	0.0000	**0.0001**
TT.SHvsOthers:RCvsCCR	0.010	0	600.89	6.3429	0.0000	**0.0001**
TJ.SHvsOthers:RCvsCCR	0.009	0	600.89	5.6478	0.0000	**0.0003**
TT.SIMvsJPlag:RCvsCCR	0.009	0	600.89	3.1604	0.0017	0.1366
TJ.SIMvsJPlag:RCvsCCR	-0.009	0	600.89	-3.3410	0.0009	**0.0008**
Tool.TextvsJava:AnoNvsAN	0.020	0	600.89	12.2862	0.0000	**0.0001**
TT.SHvsOthers:AnoNvsAN	0.029	0	600.89	18.2468	0.0000	**0.0001**
TJ.SHvsOthers:AnoNvsAN	-0.010	0	600.89	-6.4298	0.0000	**0.0001**
TT.SIMvsJPlag:AnoNvsAN	0.000	0	600.89	0.0000	1.0000	0.5384
TJ.SIMvsJPlag:AnoNvsAN	0.000	0	600.89	-0.0602	0.9520	0.4950

Note:
TT - ToolText, TJ - ToolJava, SH - Sherlock, AnoN - AllnoNOR, AN - AllNOR

Table 8.13: Simple effect analysis result for SOCO D3

Name	Estimate	SE	df	t.value	Pr(>|t|)	p.boot
(Intercept)	0.686	0	179.80	304.4271	0.0000	**0.0001**
TextvsJava	-0.081	0	177.07	-36.7241	0.0000	**0.0001**
TT.SHvsOthers	-0.066	0	177.07	-29.8239	0.0000	**0.0001**
TT.SIMvsJPlag	-0.234	0	177.07	-61.2247	0.0000	**0.0001**
TJ.SHvsOthers	-0.100	0	177.07	-45.5599	0.0000	**0.0001**
TJ.SIMvsJPlag	-0.042	0	177.07	-11.0611	0.0000	**0.0005**
TT.SH.NoPPTvsPPT	-0.008	0	716.11	-6.1515	0.0000	**0.0001**
TT.SH.SinglevsCombo	-0.053	0	716.11	-18.7249	0.0000	**0.0001**
TT.SH.RCvsCCR	0.014	0	716.11	3.5977	0.0003	0.0573
TT.SH.AllnoNORvsAllNOR	-0.087	0	716.11	-21.9629	0.0000	**0.0001**
TT.JPlag.NoPPTvsPPT	0.014	0	716.11	11.0463	0.0000	**0.0001**
TT.JPlag.SinglevsCombo	0.020	0	716.11	7.1291	0.0000	**0.0001**
TT.JPlag.RCvsCCR	0.053	0	716.11	13.4288	0.0000	**0.0001**
TT.JPlag.AllnoNORvsAllNOR	0.000	0	716.11	0.0000	1.0000	0.5304
TT.SIM.NoPPTvsPPT	0.010	0	716.11	7.7787	0.0000	**0.0001**
TT.SIM.SinglevsCombo	0.020	0	716.11	6.9812	0.0000	**0.0001**
TT.SIM.RCvsCCR	0.036	0	716.11	9.0362	0.0000	**0.0001**
TT.SIM.AllnoNORvsAllNOR	0.000	0	716.11	0.0000	1.0000	0.5268
TJ.SH.NoPPTvsPPT	0.016	0	716.11	12.5280	0.0000	**0.0001**
TJ.SH.SinglevsCombo	0.039	0	716.11	13.7553	0.0000	**0.0001**
TJ.SH.RCvsCCR	-0.045	0	716.11	-11.3371	0.0000	**0.0001**
TJ.SH.AllnoNORvsAllNOR	0.031	0	716.11	7.6975	0.0000	**0.0001**
TJ.JPlag.NoPPTvsPPT	-0.008	0	716.11	-6.5616	0.0000	**0.0001**
TJ.JPlag.SinglevsCombo	-0.014	0	716.11	-4.9696	0.0000	**0.0001**
TJ.JPlag.RCvsCCR	-0.027	0	716.11	-6.8608	0.0000	**0.0001**
TJ.JPlag.AllnoNORvsAllNOR	0.000	0	716.11	-0.0837	0.9333	0.4874
TJ.SIM.NoPPTvsPPT	-0.003	0	716.11	-2.1034	0.0358	0.0152
TJ.SIM.SinglevsCombo	-0.004	0	716.11	-1.5678	0.1174	0.0595
TJ.SIM.RCvsCCR	-0.009	0	716.11	-2.2172	0.0269	0.0131
TJ.SIM.AllnoNORvsAllNOR	0.000	0	716.11	0.0000	1.0000	0.5240

Note:
TT - ToolText, TJ - ToolJava, SH - Sherlock

Differences between no preprocessing and preprocessing

The first five interaction terms look at the effect of techniques (i.e., all PPTs combined) relative to when no PPT is used when comparing the same tools, as was done for assignments D1 and D2. For the five contrasts the interaction differences are significant except for SIM-text vs. JPlag-text. This means that using PPTs (in comparison to when no technique is used) affects SIM-text and JPlag-text tool in the same way.

Based on the five contrasts it is safe to say that in most cases the tool accuracy changes differently when PPTs is used in comparison when no technique is used. This indicates but does not confirm that using PPTs makes a difference for an individual tool used on the D3 assignments. By examining the interaction graphs (Figure J.5) it is apparent that the accuracy increases for SIM-text, JPlag-text and Sherlock-java but decreases for the other three tools.

To check H1, if there exists a difference in plagiarism detection accuracy between at least one PPT and when no PPT is used, a simple effects analysis was performed. The results of the simple effects analysis for SOCO D3 are presented in Table 8.13 and the corresponding effect sizes in Table I.6. The analysis confirms that using PPTs makes significant difference (at $p_b < 0.01$) in comparison to when no PPT is used for all tools except SIM-java ($p_b = 0.02$). For tools where the difference was significant, techniques have a positive effect on SIM-text, JPlag-text and Sherlock-java and a negative effect on the other three tools.

Based on the presented results it can be stated that the H1 hypothesis is confirmed for the SOCO D3 assignments.

Differences between different preprocessing techniques

In the previous analysis of the SOCO D3 assignments it was confirmed that using PPTs significantly changes the accuracy for most tools and that the change is different between most tools. The problem with the previous analysis, is that all techniques were put together and if one technique had a positive effect and another technique a negative effect the effects could cancel each other. To overcome the problem the PPTs were analysed further in comparison to each other in the same way as for the D1 and D2 assignments.

Regarding the interaction contrasts, as can be seen from Table 8.12 11 contrasts show significant differences at $p_b < 0.01$ and 4 did not. The contrasts that compare single and combo technique do not show a significant result (at $p_b < 0.01$) for SIM-text vs. JPlag-text ($p_b = 0.56$). The 10 comparisons that compare single techniques (RC vs. CCR) and the comparisons that compare combo techniques (AllnoNOR vs. AllNOR) are significant $p_b < 0.01$ for 7 contrasts. Non-significant results are for contrasts comparing SIM-text vs. JPlag-text for both single and combo comparisons, and for SIM-java vs. JPlag-java when comparing combo techniques. In comparison to the first two assignments (D1 and D2) there is no more similarity between SIM-text and JPlag-text and not between SIM-java and JPlag-java.

From the 11 contrasts that compare different techniques and that are significant it can be

concluded that accuracy changes are different for different tools when comparing different techniques mostly for textual versions of the tools. Again, as with the previous analysis (no PPT vs. PPT), this indicates but does not confirm that there is a difference in using different techniques on an individual tool. To check H2, if there exists a difference in plagiarism detection accuracy between at least two different PPTs, a simple effects analysis was used.

The simple effects analysis of single techniques compared to combo techniques showed a significant difference in accuracy with $p_b < 0.01$ for all tools except SIM-java ($p_b = 0.06$). From the interaction graphs (Figure J.3 and Figure J.4) it can be seen that combo techniques show an increase in accuracy in comparison to single techniques for SIM-text, JPlag-text and Sherlock-java and a decrease for the others. This means that using combo techniques was successful 50% of the time, and this is the same as for the D2 assignment.

A simple effects analysis of RC vs. CCR techniques shows that the CCR technique decreases the accuracy for all Java versions of the tools and an increase for the textual versions of the tools. The simple effects analysis for the combo techniques (AllnoNOR vs. AllNOR) shows that there is no difference between the two techniques for SIM and JPlag tools (both versions had $p_b = 0.5$) and but there is a significant difference for Sherlock tool (both versions $p_b < 0.01$).

Based on the presented results it can be stated that the H2 hypothesis is confirmed for the SOCO D3 assignments.

8.2.5 Results for D4 assignments

D4 assignments are created from the B2 assignments in the SOCO dataset. The B2 assignments are of medium difficulty but with high complexity, as are the A2 assignments. On average files in B2 have 102.46 LOC, which is 6 lines fewer than the A2 assignments. In Figure 8.12 the boxplot for F1 scores is presented at each level of tool and technique for the D4 assignments. It can be observed that preprocessing techniques seem to have a stronger effect on textual versions of the tools than on the Java versions, which is the same as for the other three assignments. For the Java version, there seems to be a small positive effect on Sherlock-java and small negative effect on JPlag-java, while for SIM-java there is no effect. The effect of PPTs on the textual versions is again strongest for Sherlock-text and again as for the other three assignments, there is a decrease rather than an increase of the accuracy. SIM-text has again (as in D1, D2 and D3) a positive effect but for JPlag-text there seems to be a small negative effect.

To confirm that there is an overall difference, as before ANOVA with bootstrap was performed. In Figure 8.13 a histogram, Q-Q plot and scatter plot of the residuals are presented. The results of the ANOVA with the F statistic and the p-values (original and bootstrapped) are presented in Table 8.14. The result shows that the p_b value is less than 0.01 for the main effects (tool and technique) and for the interaction effect. The model comparison is presented in Table H.4 where for this dataset $\chi^2(20) = 99.504$, $p_b < 0.01$. Next, contrast comparisons and a simple effects analysis were performed as for the D1 to D3 assignments. The visual representation of the means is presented in Figure 8.14.

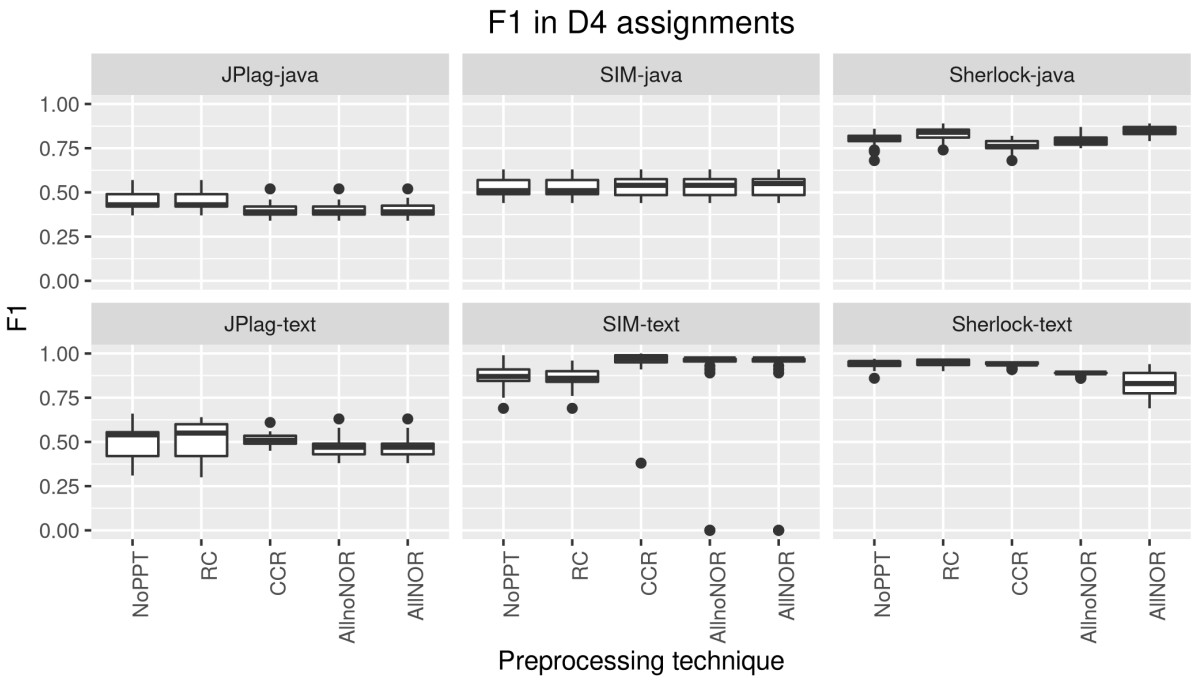


Figure 8.12: F1 score for SOCO D4 assignment with 3*IQR

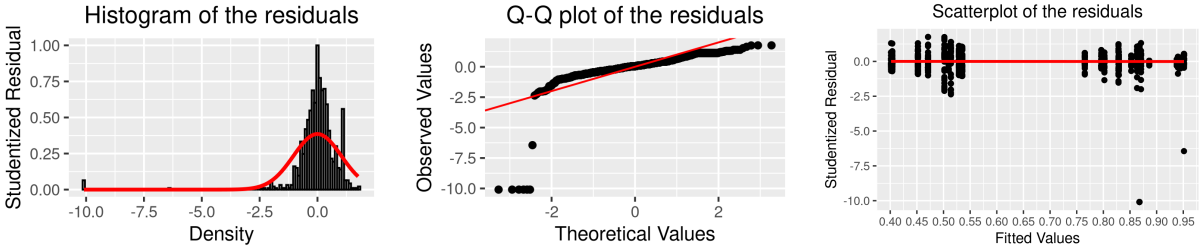


Figure 8.13: D4 assignments - residuals

Table 8.14: ANOVA results for SOCO D4

	Sum Sq	Mean Sq	NumDF	DenDF	F value	Pr(>F)	p.boot
Tool	12.66	2.533	5	155	436.2	0e+00	0.0001
Technique	0.13	0.034	4	124	5.8	3e-04	0.0003
Tool:Technique	**0.62**	**0.031**	**20**	**620**	**5.4**	**0e+00**	**0.0001**

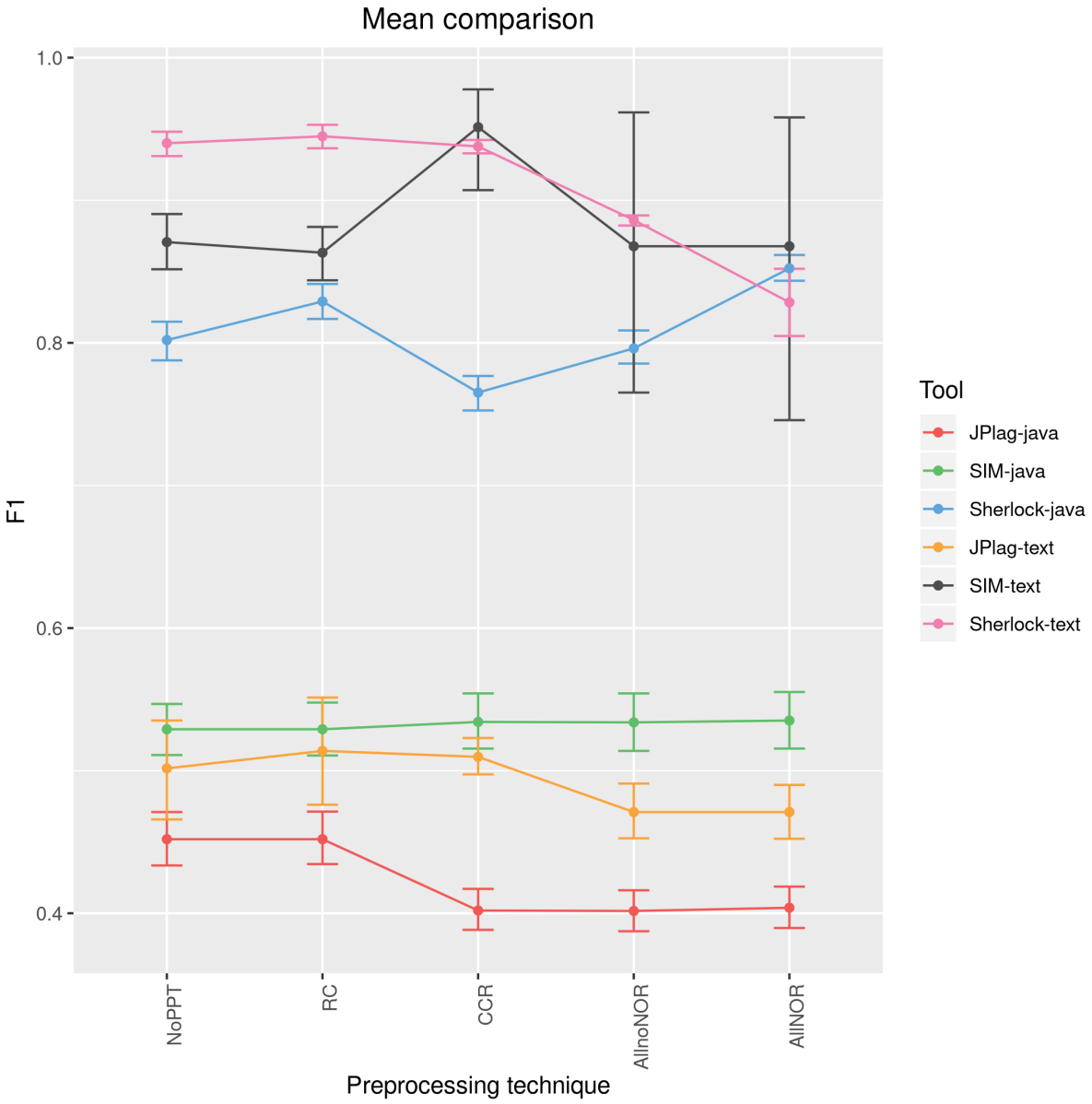


Figure 8.14: F1 mean comparison for SOCO D4

Figure 8.14 indicates that techniques have an effect on accuracy for all tools except SIM-java. The change is positive for SIM-text and Sherlock-java and negative for the other tools, especially for Sherlock-text. By looking at Figure 8.8, as for the three previous assignments, it can be observed that some techniques have a reverse effect depending on which tool is used and/or to what technique they are compared.

In general, Figure 8.8 looks a lot like the mean comparison graph for the D3 assignment and differs from D1 and D2. As it seems from the mean comparison graph, the difference between D3 and D4 is greater than that between D2 and D1. A detailed comparison of the results for all four assignments is given in the Discussion Section.

Analysing the graphs only visually, as stated before, can be informative but it can also be misleading. Therefore, the results of the planned comparison for the D4 assignments are presented in Table 8.15 and the corresponding effect sizes are presented in Table I.7. The results of the simple effects analysis are presented in Table 8.16 and the corresponding effect sizes are presented in Table I.8. From 20 interaction contrasts only 3 have a significant difference with $p_b < 0.01$. From 24 comparisons in the simple effects analysis only 9 had a significant result with $p_b < 0.01$. In comparison to the other three assignments, for this assignment the preprocessing techniques had the least effect.

Table 8.15: Contrasts results for SOCO D4

Name	Estimate	SE	df	t.value	Pr(>|t|)	p.boot
(Intercept)	0.675	0	31	137.8302	0.0000	0.0924
Tool.TextvsJava	-0.087	0	155	-20.6044	0.0000	**0.0003**
TT.SHvsOthers	-0.073	0	155	-17.2757	0.0000	**0.0010**
TJ.SHvsOthers	-0.111	0	155	-26.2041	0.0000	**0.0005**
TT.SIMvsJPlag	-0.195	0	155	-26.7350	0.0000	**0.0007**
TJ.SIMvsJPlag	-0.055	0	155	-7.5269	0.0000	0.0290
NoPPTvsPPT	-0.002	0	124	-1.4921	0.1382	0.0836
SinglevsCombo	-0.013	0	124	-4.5237	0.0000	**0.0001**
RCvsCCR	-0.003	0	124	-0.6443	0.5206	0.2760
AnoNvsAN	0.000	0	124	0.0390	0.9689	0.5356
Tool.TextvsJava:NoPPTvsPPT	0.000	0	620	0.2604	0.7947	0.6365
TT.SHvsOthers:NoPPTvsPPT	0.003	0	620	2.3497	0.0191	0.3657
TJ.SHvsOthers:NoPPTvsPPT	-0.002	0	620	-1.3470	0.1785	0.0908
TT.SIMvsJPlag:NoPPTvsPPT	-0.003	0	620	-1.2522	0.2110	0.1099
TJ.SIMvsJPlag:NoPPTvsPPT	-0.004	0	620	-1.9007	0.0578	0.0324
Tool.TextvsJava:SinglevsCombo	0.014	0	620	5.0761	0.0000	**0.0008**
TT.SHvsOthers:SinglevsCombo	0.007	0	620	2.6174	0.0091	0.2843
TJ.SHvsOthers:SinglevsCombo	-0.006	0	620	-2.2518	0.0247	0.0169
TT.SIMvsJPlag:SinglevsCombo	0.000	0	620	-0.0667	0.9469	0.4954
TJ.SIMvsJPlag:SinglevsCombo	-0.007	0	620	-1.4001	0.1620	0.0864
Tool.TextvsJava:RCvsCCR	-0.015	0	620	-3.9125	0.0001	**0.0001**
TT.SHvsOthers:RCvsCCR	0.008	0	620	2.0685	0.0390	0.4980
TJ.SHvsOthers:RCvsCCR	0.007	0	620	1.7487	0.0808	0.6470
TT.SIMvsJPlag:RCvsCCR	-0.023	0	620	-3.3707	0.0008	**0.0008**
TJ.SIMvsJPlag:RCvsCCR	-0.014	0	620	-2.0153	0.0443	0.0265
Tool.TextvsJava:AnoNvsAN	0.010	0	620	2.4768	0.0135	0.3335
TT.SHvsOthers:AnoNvsAN	0.010	0	620	2.4360	0.0151	0.3347
TJ.SHvsOthers:AnoNvsAN	-0.009	0	620	-2.2931	0.0222	0.0128
TT.SIMvsJPlag:AnoNvsAN	0.000	0	620	0.0000	1.0000	0.5327
TJ.SIMvsJPlag:AnoNvsAN	0.000	0	620	0.0354	0.9718	0.5430

Note:
TT - ToolText, TJ - ToolJava, SH - Sherlock, AnoN - AllnoNOR, AN - AllNOR

Table 8.16: Simple effects analysis result for SOCO D4

Name	Estimate	SE	df	t.value	Pr(>|t|)	p.boot
(Intercept)	0.675	0	31.00	137.8302	0.0000	0.0976
TextvsJava	-0.087	0	155.00	-20.6044	0.0000	**0.0004**
TT.SHvsOthers	-0.073	0	155.00	-17.2757	0.0000	**0.0014**
TT.SIMvsJPlag	-0.195	0	155.00	-26.7350	0.0000	**0.0003**
TJ.SHvsOthers	-0.111	0	155.00	-26.2041	0.0000	**0.0002**
TJ.SIMvsJPlag	-0.055	0	155.00	-7.5269	0.0000	0.0292
TT.SH.NoPPTvsPPT	-0.008	0	743.13	-2.6413	0.0084	**0.0046**
TT.SH.SinglevsCombo	-0.042	0	743.13	-6.0932	0.0000	**0.0001**
TT.SH.RCvsCCR	-0.004	0	743.13	-0.3639	0.7161	0.3755
TT.SH.AllnoNORvsAllNOR	-0.029	0	743.13	-2.9606	0.0032	**0.0030**
TT.JPlag.NoPPTvsPPT	-0.002	0	743.13	-0.6642	0.5067	0.2739
TT.JPlag.SinglevsCombo	-0.020	0	743.13	-2.9589	0.0032	**0.0035**
TT.JPlag.RCvsCCR	-0.002	0	743.13	-0.2150	0.8298	0.4367
TT.JPlag.AllnoNORvsAllNOR	0.000	0	743.13	0.0000	1.0000	0.5279
TT.SIM.NoPPTvsPPT	0.003	0	743.13	1.0931	0.2747	0.9574
TT.SIM.SinglevsCombo	-0.020	0	743.13	-2.8653	0.0043	**0.0057**
TT.SIM.RCvsCCR	0.044	0	743.13	4.5153	0.0000	**0.0079**
TT.SIM.AllnoNORvsAllNOR	0.000	0	743.13	0.0000	1.0000	0.5305
TJ.SH.NoPPTvsPPT	0.002	0	743.13	0.5649	0.5723	0.8215
TJ.SH.SinglevsCombo	0.014	0	743.13	1.9648	0.0498	0.5899
TJ.SH.RCvsCCR	-0.032	0	743.13	-3.2748	0.0011	**0.0022**
TJ.SH.AllnoNORvsAllNOR	0.028	0	743.13	2.8779	0.0041	0.1938
TJ.JPlag.NoPPTvsPPT	-0.007	0	743.13	-2.4059	0.0164	**0.0095**
TJ.JPlag.SinglevsCombo	-0.012	0	743.13	-1.7543	0.0798	0.0571
TJ.JPlag.RCvsCCR	-0.025	0	743.13	-2.5636	0.0106	**0.0072**
TJ.JPlag.AllnoNORvsAllNOR	0.001	0	743.13	0.1158	0.9079	0.5761
TJ.SIM.NoPPTvsPPT	0.001	0	743.13	0.2615	0.7938	0.6870
TJ.SIM.SinglevsCombo	0.001	0	743.13	0.2105	0.8333	0.6850
TJ.SIM.RCvsCCR	0.003	0	743.13	0.2646	0.7914	0.6559
TJ.SIM.AllnoNORvsAllNOR	0.001	0	743.13	0.0662	0.9473	0.5584

Note:
TT - ToolText, TJ - ToolJava, SH - Sherlock

Differences between no preprocessing and preprocessing

The first five interaction terms look at the effect of techniques (i.e., all PPTs combined) relative to when no PPT is used when comparing the same tools, as was done in other assignments. For the five contrasts the interaction difference was not significant for any of the comparisons. This means that using PPTs (compared to when no technique is used) affects all tested tool comparisons in the same way.

Based on the five contrasts the question arises as to whether the tool accuracy changes the same way when PPTs are used in comparison to when no technique is used. This indicates but does not confirm that using PPTs makes no difference. By examining the interaction graphs (Figure J.7) it is apparent that the accuracy increases for Sherlock-java, SIM-text, SIM-java, and decreases for Sherlock-text, JPlag-text, and JPlag-java.

To check H1, if there exists a difference in plagiarism detection accuracy between at least one PPT and when no PPT is used, a simple effects analysis was performed. The results of the simple effects analysis for SOCO D4 are presented in Table 8.16 and the corresponding effect sizes in Table I.8. The analysis confirmed that using PPTs made a significant difference (at $p_b < 0.01$) compared with when no PPT is used for Sherlock-text and JPlag-java. On the other hand, for the other tools the difference was not big enough to be considered significant. For tools where the difference was significant, techniques had a negative effect. Although this fact is concerning, one should take into account that all techniques were looked together (i.e., techniques were not individually compared to when no technique is used) and that the differences that are visible in Figure 8.14 are masked.

Based on the presented results, although with a negative effect, it can still be stated that the H1 hypothesis is confirmed for the SOCO D4 assignments.

Differences between different preprocessing techniques

In the previous analysis of the SOCO D4 assignments it was confirmed that using PPTs significantly decreases the accuracy for two tools and that the decrease is different between the two tools. The problem with the previous analysis is that all techniques were put together and if one technique had a positive effect and another technique a negative effect the effects could cancel each other. To overcome the problem the PPTs were analysed further in comparison to each other in the same way as for the other assignments.

Regarding the interaction contrasts, as can be seen from Table 8.15 3 contrasts showed significant differences at $p_b < 0.01$ and others did not. The contrasts that compare single and combo techniques only showed significant results ($p_b < 0.01$) for text vs. Java tools. The 10 comparisons that compare single techniques (RC vs. CCR) and the comparisons that compare combo techniques (AllnoNOR vs. AllNOR) are significant $p_b < 0.01$ for 2 contrasts. Significant results were also obtained for contrasts comparing single techniques for text vs. Java and SIM-text vs. JPlag-text.

Since there were only 3 contrasts that compare different techniques and that are significant, it can be concluded that accuracy changes for most tools are similar when comparing different techniques. To check H2, if there exists a difference in plagiarism detection accuracy between at least two different PPTs, a simple effects analysis was used.

The simple effects analysis of single techniques compared to combo techniques showed a significant difference in accuracy with $p_b < 0.01$ for all textual versions of the tools. From the interaction graphs (Figure J.7 and Figure J.8) it can be seen that combo techniques show an increase in accuracy in comparison to single techniques for Sherlock-java and JPlag-java and a decrease for the others. This means that using combo techniques was unsuccessful in this assignment since the positive effects of a PPTs were not significant, only negative effects were significant.

A simple effects analysis of RC vs. CCR techniques shows that the CCR technique increases the accuracy for SIM-text and SIM-java but only for SIM-text was this significant. There was a decrease for other tools which was significant for Sherlock-java and JPlag-java. The simple effects analysis for the combo techniques (AllnoNOR vs. AllNOR) showed that there is no difference between the two techniques for the SIM and JPlag tools (both versions had $p_b = 0.5$) and Sherlock-java ($p_b = 0.19$) but there is a significant difference for Sherlock-text ($p_b < 0.01$) which was negative.

Based on the presented results it can be stated that the H2 hypothesis is confirmed on the SOCO D3 assignments, but as previously (no PPT vs. PPT) most significant changes are negative. Even though most significant changes are negative some positive effects can be observed (Figure 8.14) for SIM-text and Sherlock-java.

8.2.6 Discussion

In the previous four sections, the H1 and H2 hypotheses were confirmed for the SOCO dataset using the statistical analysis. To answer the research question Q1 and fulfil the second goal G2, the results of the statistical tests for the four assignments are organized and qualitatively examined. The qualitative examination should help identify any patterns that are repeated on all four assignments.

In Table 8.17 the summary of the results is presented which is, in essence, the answer to the research question (Q1): "*How do different PPTs affect the plagiarism detection accuracy?*". Table 8.17 gives an overview of all statistically significant results for the four assignments (D1, D2, D3 and D4). If the comparison was not statistically significant for some assignment it is not listed. The plus and minus symbol present in which direction was the effect of the preprocessing techniques. Each contrast was stated in the way that the second part in the contrast name was expected to improve the accuracy in contrast to the first part and therefore the effect should have been positive. For example, when comparing no PPT vs. PPT it was expected that using PPT should have a positive effect, similarity if comparing Single vs. Combo it was expected that, when using a combination of techniques, the combination should have a positive effect in

comparison to when only one single technique is used.

From Table 8.17 it can be seen that throughout the four assignments that were statistically tested, similar conclusions can be drawn. Overall we can state the following, the effect of PPT is not significant for SIM-java for all tested assignments on all comparisons. This means using PPT when SIM-java is used makes no difference at all. If we take into account that preprocessing takes time, the best suggestion is to not use PPT with SIM-java. For JPlag-java and Sherlock-text there were some significant results but with a negative effect. This means that if PPTs are used in combination with JPlag-java and Sherlock-java they gave worse results than when no PPTs were used. The suggestion is therefore as for SIM-java not to use PPT in combination with JPlag-java and Sherlock-text.

To answer why this happens one needs to look at the Precision and Recall and from this it can be seen that for JPlag-java the Recall is constant, which means that using PPT includes more matches but most of the plagiarised matches were already included (Recall > 0.9). Similarly, for Sherlock-text the Recall increases slightly but the Precision decreases a lot. So in the case of JPlag-java and Sherlock-text the techniques increase the number of potentially plagiarised matches without including any new (or just a small number of) plagiarised matches. This decreases Precision, and since Recall is constant it decreases also the F1 value. Now it is a logical follow up question as to why a technique lowers the Precision, but to answer that question is out of the scope of this research. To answer such questions one needs to analyse in detail how the JPlag-java or the Sherlock-text algorithms work, to understand why code parts removal or reordering would cause a match to be suspicious that was not suspicious before using a PPT.

A positive effect of the techniques was found for Sherlock-java, SIM-text and JPlag-text. Sherlock-java as presented in Table 8.17 had a positive effect when PPTs were used, it had a positive effect with combo techniques but a negative effect with single techniques. From this it can be stated that it makes sense to use PPTs with Sherlock-java, but one needs to be careful

Table 8.17: Summary of SOCO assignements significant comparisons

	NoPPT vs	**Single vs**	**RC vs**	**AllnoNOR vs**
Tool	**PPT**	**Combo**	**CCR**	**AllNOR**
SIM-java				
JPlag-java	-D2 -D3 -D4	-D2 -D3	-D2 -D3 -D4	
SH-java	+D3	+D1 +D3	-D1 -D2 -D3 -D4	+D3
SIM-text	+D1 +D2 +D3	+D1 +D2 +D3 -D4	+D1 +D2 +D3 +D4	
JPlag-text	+D1 +D2 +D3	+D2 +D3 -D4	+D1 +D2 +D3	
SH-text	-D1 -D2 -D3 -D4	-D1 -D2 -D3 -D4	-D1 -D2	-D1 -D2 -D3 -D4

Note:
Symbol + means effect was positive and - means effect was negative.
SH - Sherlock

which to choose. To be more precise, it is not advisable to use the CCR technique, rather it is better to use the simpler RC technique. Since combinations had a positive effect in comparison to single technique, it would be good to use them but without the CCR technique meaning using only the RC technique in combination with the NOR technique. Note that the combination of RC and NOR was not tested directly in this research, and the conclusion is based on the fact that there is a negative effect of the CCR technique and a positive impact of the NOR technique when RC is combined with CCR.

Regarding SIM-text and Jplag-text, it is clear that PPTs have with those tools the most positive impact. When using both tools it is advisable to use the CCR technique over RC, but even better it is advisable to combine those two techniques. Since there is no significant difference between the combo techniques it is better to use the AllnoNOR combination which does not include the NOR technique. There is a small chance that combo will have a negative effect in comparison to single techniques, so when doing the analysis it would be good to run both the CCR and the AllnoNOR techniques. Since there is the same influence of techniques on SIM-text and Jplag-text, it is reasonable to ask if the influence has the same magnitude. The answer to that question is obtained by the planned contrasts (Tables 8.6, 8.9, 8.12, and 8.15) and it can be stated that there is a difference in magnitude and that the increase in accuracy is overall significantly larger for SIM-text that it is for JPlag-text. In short, SIM-text responds better to PPTs than JPlag-text.

Based on the previous analysis it can be concluded that PPTs have a bigger impact on textual tools than on Java tools, but the effect is not always positive (i.e., Sherlock-text). Although it was not explicitly tested there is a good indication (number of assignments with significant positive effect) that SIM-text and JPlag-text have a bigger positive impact than Sherlock-java. If the results of Sherlock-java are compared to SIM-text and JPlag-text there is a positive effect of the CCR technique (over RC technique) on SIM-text and JPlag-text, while for Sherlock-java CCR the technique has a negative effect. Similarly, the AllNOR technique has no significant effect (in comparison to the AllnoNOR technique) for SIM-text and JPlag-text, but it has for Sherlock-java. The fact that the NOR technique did not impact SIM-text and JPlag-text indicates that those tools have some prevention of simple code reformatting implemented in the algorithm, while Sherlock-java does not have that initially. Another interesting conclusion that can be drawn from that is although Sherlock-java has not integrated the NOR technique it is mostly not a problem since only one of four assignments showed a significant change.

The reason why the positive effect happens on the three tools is the same reason as to why the negative effect happens for JPlag-java and Sherlock-text, and this is that the Precision was increased while the Recall remains more or less constant. For Sherlock-java there is some decrease in Recall but the increase of Precision is larger. Again, the logical follow up question would be why a technique increases the Precision, but as stated before to answer that question is currently out of the scope of this research.

From a high level view it can be stated that the reason why different techniques have different

effects on the accuracy is that techniques try to process different elements of the code. For example RC only removes comments while CCR does not remove comments but removes many other elements. So it is no surprise that the effects were different, although (as visible from the discussion) it depends on the tool and to some extent on the assignments whether the effect is positive or negative, or if there is no effect. The reason why the effect of the same technique is different for different tools is that the underlying algorithms in the tools are different, so if a tool's algorithm ignores comments it is natural that the RC technique has no effect. It seems the complexity of the programs appears not to have any major impact on the final result.

In summary, from the statistical tests throughout the SOCO dataset it is clear that using PPTs makes a difference (H1 confirmed) and that different techniques have different impacts (H2 confirmed). The discussion above gave the answer to *how* do different techniques impact the accuracy of the individual tools (Q1). Based on the results here is a short guideline on which techniques to use on which tools:

- SIM-java — no PPT should be used;
- JPlag-java — no PPT should be used;
- Sherlock-java — RC technique or combination of RC and NOR techqniue should be used;
- SIM-text — CCR technique or AllnoNOR technique should be used;
- JPlag-text — CCR technique or AllnoNOR technique should be used;
- Sherlock-text — no PPT should be used.

Note that the guidelines are only based on the significant results obtained in the statistical analyses. This means that only comparisons that are used in the statistical analyses are used to create the guidelines. Different guidelines might be created if they would be based only on the mean comparison graphs or if every individual technique would be compared to when no technique is used.

8.2.7 Guidelines verification

To verify if the created guidelines would be good to make decisions about which technique to use or not use, they were tested on T1, C1, and C2 assignments. The test was simple, namely whether the overall result is better by following the guidelines or if the result is better by doing something else (usually the opposite).

In Figure 8.1b the results for the SOCO T1 assignment were presented and in Figure 8.2b the results for the SOCO C2 assignment were presented with the same configuration as it was for the D1 to D4 assignments and with the same threshold level. For C1, since there are no plagiarised cases, the analysis is done simply on the number of false positives. The results are presented in Figure 8.15, note that a smaller number is better in this case.

For JPlag-java and SIM-java the guidelines say not to use PPTs, and this would be a good decision for 3 of 4 cases in the T1 and C2 assignments. Only for JPlag-java in T1 would it be better to use some technique which could get an F1 increase for 0.05. On C1 for JPlag-java there is an increase of 8 false positive matches when using PPTs, so not using PPT is a good decision, on the other hand, for SIM-java there is a decrease of 4 false positive matches so the decision is not good. In general, following the guidelines for SIM-java and JPlag-java is good for 4 of 6 times, and the 2 times when it would be better to use PPTs does not make a big difference in comparison to when the techniques are not used.

For Sherlock-java the guidelines say to use RC or combination of RC and NOR. For T1 this is good advice while for C2 it is not. For C1 the guidelines are good since the use of RC decreases for 6 false positives and the mentioned combination for approximately 7 false positives, but it must be stated that using CCR would make an even better decease of an additional 10 cases. In general, the guidelines for Sherlock-java can be considered good for 2 of 3 cases. They lead to an increase in accuracy for T1 and C1 on the other hand for C2 they lead to a decrease in accuracy.

For Sherlock-text the guidelines say to not use PPT. Such a decision would not be good for T1 or for C2 since better results could be achieved by using PPTs except for the AllNOR technique. For C1 also it would be better to use PPTs, but the best is AllnoNOR which would make a decrease of 14 false positives, on the other hand, AllNOR would increase the false positives for 16 matches. Overall, the guidelines can be considered bad for Sherlock-text since for 3 cases the results were worse than when PPTs were used.

For SIM-text and JPlag-text the guidelines say to use CCR or AllnoNOR. This advice would be good for C2 while for T1 the techniques would make no difference for JPlag-text (decrease is only 0.005) and a small decrease of F1 value for 0.02 for SIM-text. Regarding C1, using AllnoNOR decreases the false positives for 43 matches for SIM-text and 133 matches for JPlag-text which makes a huge difference. In general, one can conclude the guidelines are good for 5 of 6 cases, and the only time when it would be better to use PPTs it does not make a big difference in comparison to when the techniques are not used.

In summary, the guidelines are considered good for both versions of SIM and JPlag, where the guidelines are better for the textual versions of the two tools (SIM-text and Jplag-text) and are considered neutral for Sherlock-java, meaning following the guidelines or not would probably lead to similar results. As for Sherlock-text, the guidelines are considered bad, meaning better results can be achieved doing the opposite.

This short verification is by no means precise and it is prone to be biased. This is especially true for the cases where the differences are small like 0.05 or 0.02 in the F1 value. In addition, there is no evidence that the differences seen are statistically significant, but regardless of these limitations to the verification it is plausible to say that the obtained guidelines are a step in the right direction, at least for assignments similar to SOCO and of course with the same configuration of the tools and the same calculation of the threshold level.

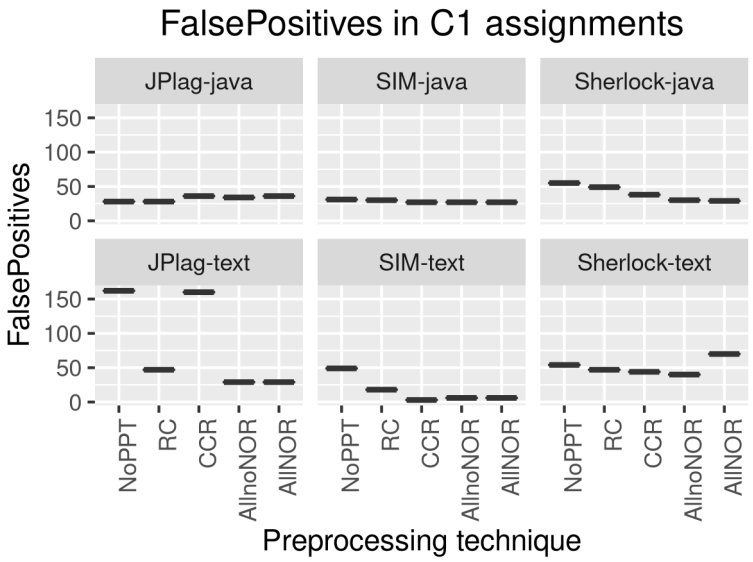


Figure 8.15: False positives for SOCO C1 assignment with 3*IQR

8.3 RSS dataset analysis

In this Section the results based on the RSS dataset are described. As already stated, data are analysed qualitatively and quantitatively, and to ensure objectivity statistical tests are performed. The RSS dataset has the same issues as the SOCO dataset regarding the violations of assumptions so they are solved in the same way as before (Section 8.2.1).

In comparison to the SOCO dataset the RSS dataset has one more issue and that is the number of participants. In the RSS dataset the "*Participants*" are academic years, this means there are only six participants available, additionally two are not valid since in two academic years there were no plagiarised matches which means there are only four "*Participants*" for the statistical analysis. According to [87] there is no simple answer to the number of participants and some say that a sample size of 20-30 is needed for good results. The problem with the RSS dataset is that it is not easy to increase the number of participants since an expert is needed to identify real plagiarised matches. This means that statistical tests will have small strength so it is expected that the results will be non significant because of the small number of participants and not because there is no effect. Because of this issue, there is a separate Section 8.3.3 which shows some benefits of using preprocessing techniques regardless of the statistical results.

The MLM equations that are used for the RSS dataset look also the same as they were for the SOCO datasets (Section 8.2.1) the only difference is that for the RSS dataset the term "*Participant*" represents one academic year. Assignments in the RSS dataset are therefore a group of assignments from different academic years. In this context, A1 then represents a group of the first assignment from different academic years. As in the SOCO dataset one group has the same characteristics like the complexity of the problem, type of the problem, what Java technologies were or were not used (JSF, ORM, web filters, EJB), etc.

To do a statistical test only assignments from one group should be tested together to control

the variability. This means that with the RSS dataset four case studies can be done, one for each group (A1, A2, A3 and A4). In the rest of the Section, A1, A2, A3 and A4 will be referred to as *assignments* rather than a group of assignments from different academic years. Note that only A1 assignments are analysed in this research, and the reason is that the time required to confirm real plagiarised matches was long (several weeks of work) and the expert managed to confirm only one assignment group in six academic years. The A4 assignments are only be used in Section 8.3.3 to display some effects of the preprocessing techniques regardless of the fact is the match plagiarised or not. The assignments A2 and A3 are not used at all and will be analysed in future work.

In the rest of this Section assignment A1 of the RSS dataset is analysed and then an overall discussion is given. In the last part of this Section the limitations of the statistical analysis are given, together with a description of some other benefits of using preprocessing techniques whereby the data from the A4 assignments will be used.

8.3.1 Results for A1 assignments

A1 assignments are the first assignments given in each academic year. The first assignment uses only standard Java, meaning there are no enterprise or web technologies (e.g., JSF, JSP), frameworks (e.g., ORM) and similar. In this assignment, students must usually implement their own server using *SocketServer* class and a client application which can communicate with this server using a unique protocol (defined by the teacher). In the implementation it is expected that the students use regular expressions and threads, and that they create unit tests for some classes. To help the students they can use all the code that has been implemented during the lab sessions or presented in the lectures. Also, for some parts of the assignments they have to use external libraries, for example, to work with JSON the students can use the Gson library [3]. On average one submission has 1800 LOC whereby the smallest was 422 LOC and largest 3872 LOC. Usually a submission with less than 500 LOC means the student submitted just the code given by the teacher. In comparison to the SOCO dataset, but also to other studies, these are much longer solutions, which is normal considering this is a course in the first year of graduate studies.

In Figure 8.16 the boxplots for F1 scores are presented at each level of tool and technique for the A1 assignments. It can be observed that preprocessing techniques seem to have a stronger effect on textual versions of the tools (especially JPlag-text and SIM-text) than on the Java versions, which is consistent with the results from the SOCO dataset. For the Java version, there seems to be a positive effect which varies from PPT to PPT. The effect of PPTs on the textual versions is strongest for JPlag-text which is consistent with D3 in the SOCO dataset. From Figure 8.16 one can see that TE technique was added which requires a template to be excluded form the solution and was therefore not used in the SOCO dataset. Also, in the AllNOR and AllnoNOR techniques the TE technique is included.

[3] `https://github.com/google/gson`

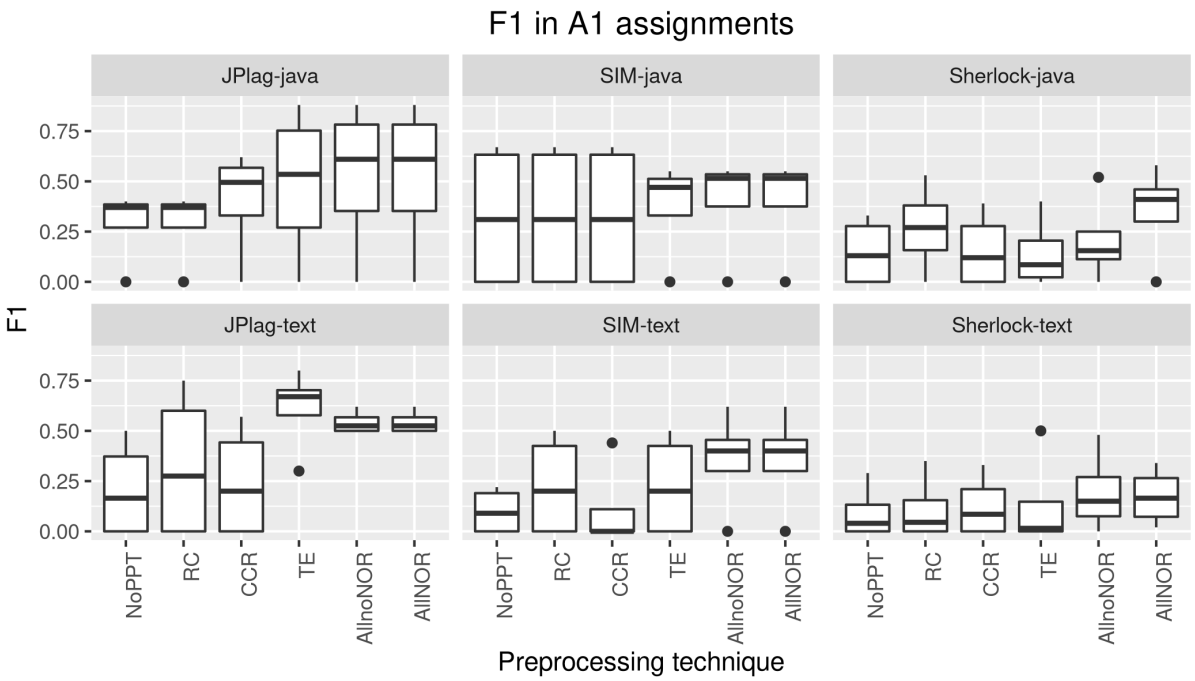


Figure 8.16: F1 score for RSS A1 assignment with 3*IQR

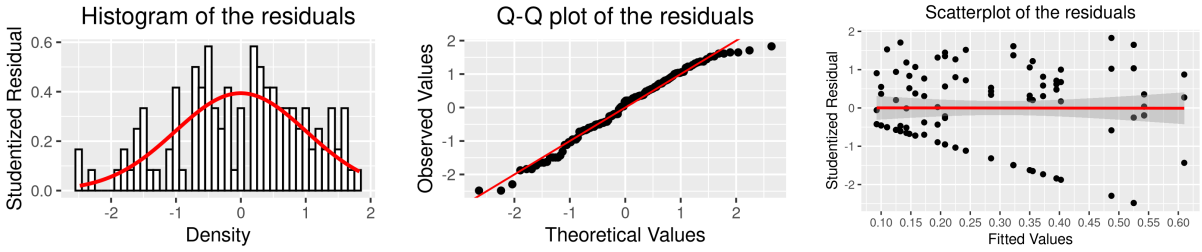


Figure 8.17: A1 assignments - residuals

Since there is a small number of "*Participants*" it is not expected to see any statistically significant differences, and because of that, instead of using p-value of 0.01 as a reference, the p value of 0.05 will be used. To improve a bit the strength but to not modify the experiment design it was decided to remove the RC technique from any statistical analysis. The reason why the RC technique was removed is that it showed poor results in the SOCO dataset and as it can be seen in Figure 8.16 it does not look promising in this dataset. In addition, the idea was to remove a single technique to again have two single techniques and two combo techniques as in the SOCO dataset. Note that the AllNOR and AllnoNOR techniques still include the RC technique.

To confirm that there is an overall difference, as before ANOVA with bootstrap was performed. In Figure 8.17 a histogram, Q-Q plot and scatter plot of the residuals are presented. The results of the ANOVA with the F statistic and the p-values (original and bootstrapped) are presented in Table 8.18. The result shows that the p_b value is larger than 0.05 for the main effects (tool and technique) but for the interaction effect it is less than 0.05. The model comparison is presented in Table H.5 where for this dataset $\chi^2(20) = 37.9$, $p_b < 0.05$. Next, contrast comparisons and a simple effects analysis were performed as for the SOCO dataset. The visual

Table 8.18: ANOVA results for RSS A1

	Sum Sq	Mean Sq	NumDF	DenDF	F value	Pr(>F)	p.boot
Tool	0.19	0.037	5	20	4.3	0.0080	0.3467
Technique	0.14	0.034	4	16	4.0	0.0199	0.3383
Tool:Technique	**0.42**	**0.021**	**20**	**80**	**2.4**	**0.0029**	**0.0406**

representation of the means is presented in Figure 8.18.

Figure 8.18 indicates that techniques have an effect on accuracy for all tools. The change is mostly positive for all tools. By looking at Figure 8.18, as for the SOCO dataset, it can be observed that some techniques have a reverse effect depending on which tool is used and/or to what technique they are compared.

Analysing the graphs only visually, as stated before, can be informative but it can also be misleading. Therefore, the results of the planned comparison for the A1 assignments are presented in Table 8.19 and the corresponding effect sizes are presented in Table I.9. The results of the simple effects analysis are presented in Table 8.20 and the corresponding effect sizes are presented in Table I.10. From 20 interaction contrasts only 1 has a significant difference with $p_b < 0.05$ and one is close with $p_b = 0.0647$. From 24 comparisons in the simple effects analysis only 3 had a significant result with $p_b < 0.05$. In comparison to the SOCO dataset these are the worst results, but then one has to keep in mind the issue with the number of participants.

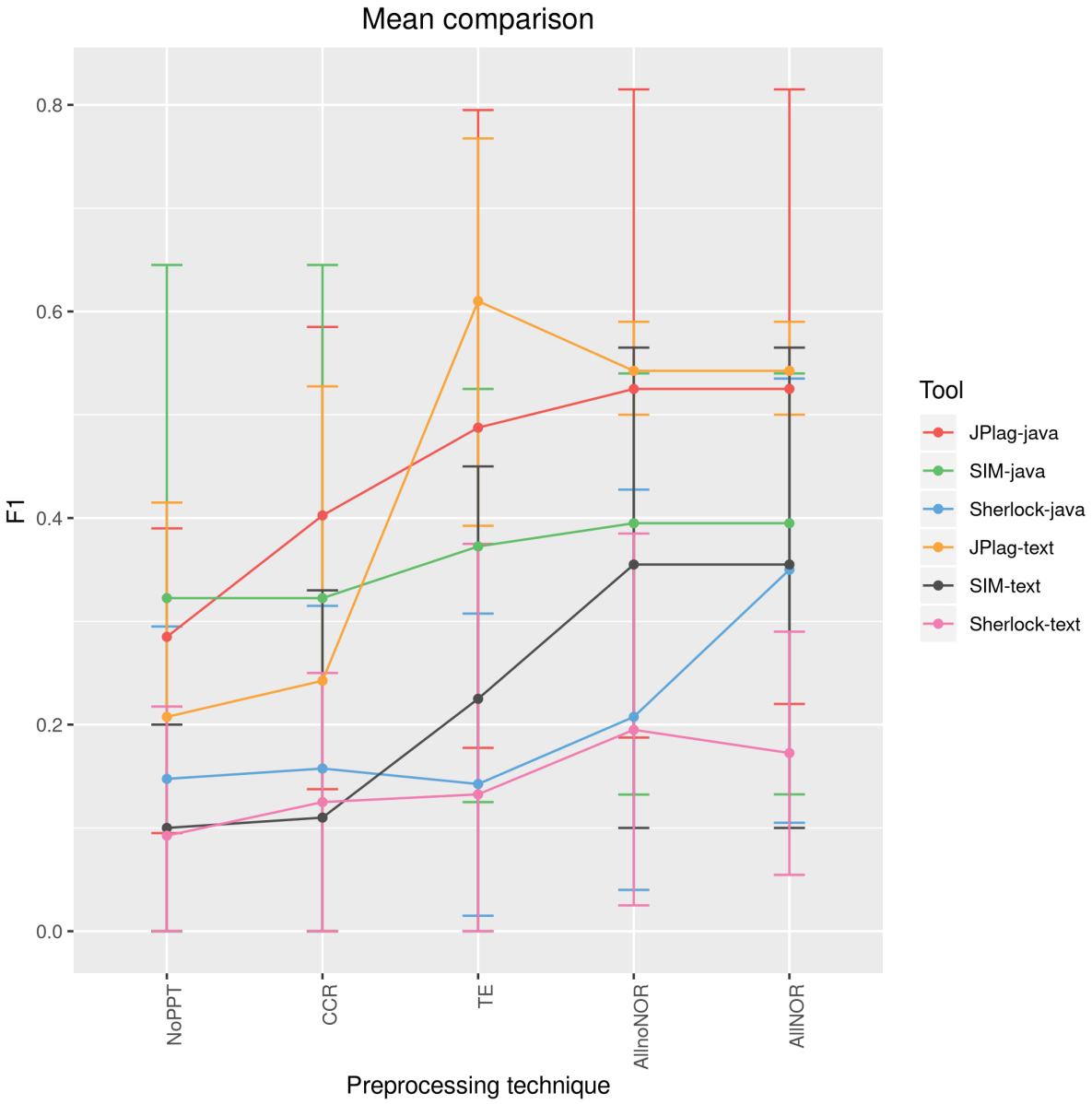


Figure 8.18: F1 mean comparison for RSS A1

Table 8.19: Contrasts results for RSS A1

Name	Estimate	SE	df	t.value	Pr(>|t|)	p.boot
(Intercept)	0.301	0	4	3.9898	0.0163	0.5994
Tool.TextvsJava	0.034	0	20	1.3740	0.1846	0.4879
TT.SHvsOthers	0.062	0	20	2.4745	0.0224	0.3616
TJ.SHvsOthers	0.067	0	20	2.6980	0.0138	0.3515
TT.SIMvsJPlag	0.100	0	20	2.3105	0.0316	0.3743
TJ.SIMvsJPlag	0.042	0	20	0.9646	0.3462	0.9392
NoPPTvsPPT	0.027	0	16	2.7730	0.0136	0.3202
SinglevsCombo	0.051	0	16	2.3324	0.0331	0.3386
CCRvsTE	0.051	0	16	1.6358	0.1214	0.4034
AnoNvsAN	0.010	0	16	0.3218	0.7518	0.5443
Tool.TextvsJava:NoPPTvsPPT	-0.006	0	80	-1.4607	0.1480	0.1098
TT.SHvsOthers:NoPPTvsPPT	0.010	0	80	2.4361	0.0171	0.3742
TJ.SHvsOthers:NoPPTvsPPT	0.004	0	80	0.9019	0.3698	0.9530
TT.SIMvsJPlag:NoPPTvsPPT	0.012	0	80	1.5706	0.1202	0.7263
TJ.SIMvsJPlag:NoPPTvsPPT	0.015	0	80	2.0546	0.0432	0.5330
Tool.TextvsJava:SinglevsCombo	-0.009	0	80	-0.8988	0.3715	0.2321
TT.SHvsOthers:SinglevsCombo	0.016	0	80	1.6989	0.0932	0.6754
TJ.SHvsOthers:SinglevsCombo	-0.011	0	80	-1.1399	0.2577	0.1746
TT.SIMvsJPlag:SinglevsCombo	-0.018	0	80	-1.0821	0.2825	0.1758
TJ.SIMvsJPlag:SinglevsCombo	0.008	0	80	0.4936	0.6230	0.7668
Tool.TextvsJava:CCRvsTE	-0.031	0	80	-2.2941	0.0244	**0.0237**
TT.SHvsOthers:CCRvsTE	0.039	0	80	2.8986	0.0048	0.2261
TJ.SHvsOthers:CCRvsTE	0.014	0	80	1.0230	0.3094	0.9894
TT.SIMvsJPlag:CCRvsTE	0.063	0	80	2.7116	0.0082	0.2800
TJ.SIMvsJPlag:CCRvsTE	0.009	0	80	0.3759	0.7080	0.7076
Tool.TextvsJava:AnoNvsAN	0.014	0	80	1.0230	0.3094	0.9889
TT.SHvsOthers:AnoNvsAN	0.004	0	80	0.2790	0.7810	0.6684
TJ.SHvsOthers:AnoNvsAN	-0.024	0	80	-1.7670	0.0810	0.0647
TT.SIMvsJPlag:AnoNvsAN	0.000	0	80	0.0000	1.0000	0.5511
TJ.SIMvsJPlag:AnoNvsAN	0.000	0	80	0.0000	1.0000	0.5459

Note:
TT - ToolText, TJ - ToolJava, SH - Sherlock, AnoN - AllnoNOR, AN - AllNOR

Table 8.20: Simple effects analysis result for RSS A1

Name	Estimate	SE	df	t.value	Pr(>|t|)	p.boot
(Intercept)	0.301	0	4.00	3.9898	0.0163	0.5967
TextvsJava	0.034	0	20.00	1.3740	0.1846	0.4803
TT.SHvsOthers	0.062	0	20.00	2.4745	0.0224	0.3616
TT.SIMvsJPlag	0.100	0	20.00	2.3105	0.0316	0.3729
TJ.SHvsOthers	0.067	0	20.00	2.6980	0.0138	0.3470
TJ.SIMvsJPlag	0.042	0	20.00	0.9646	0.3462	0.9342
TT.SH.NoPPTvsPPT	0.013	0	51.01	0.9326	0.3554	0.8091
TT.SH.SinglevsCombo	0.028	0	51.01	0.8996	0.3726	0.7776
TT.SH.CCRvsTE	0.004	0	51.01	0.0867	0.9312	0.2480
TT.SH.AllnoNORvsAllNOR	-0.011	0	51.01	-0.2602	0.7957	0.3016
TT.JPlag.NoPPTvsPPT	0.055	0	51.01	4.0506	0.0002	**0.0077**
TT.JPlag.SinglevsCombo	0.058	0	51.01	1.9015	0.0629	0.4363
TT.JPlag.CCRvsTE	0.184	0	51.01	4.2504	0.0001	**0.0004**
TT.JPlag.AllnoNORvsAllNOR	0.000	0	51.01	0.0000	1.0000	0.4210
TT.SIM.NoPPTvsPPT	0.032	0	51.01	2.3590	0.0222	0.2926
TT.SIM.SinglevsCombo	0.094	0	51.01	3.0669	0.0035	**0.0496**
TT.SIM.CCRvsTE	0.058	0	51.01	1.3301	0.1894	0.7865
TT.SIM.AllnoNORvsAllNOR	0.000	0	51.01	0.0000	1.0000	0.4337
TJ.SH.NoPPTvsPPT	0.013	0	51.01	0.9784	0.3325	0.8378
TJ.SH.SinglevsCombo	0.064	0	51.01	2.1059	0.0402	0.3202
TJ.SH.CCRvsTE	-0.007	0	51.01	-0.1735	0.8630	0.1442
TJ.SH.AllnoNORvsAllNOR	0.071	0	51.01	1.6481	0.1055	0.5783
TJ.JPlag.NoPPTvsPPT	0.040	0	51.01	2.9260	0.0051	0.1052
TJ.JPlag.SinglevsCombo	0.040	0	51.01	1.3085	0.1966	0.8751
TJ.JPlag.CCRvsTE	0.043	0	51.01	0.9831	0.3302	0.8980
TJ.JPlag.AllnoNORvsAllNOR	0.000	0	51.01	0.0000	1.0000	0.4369
TJ.SIM.NoPPTvsPPT	0.010	0	51.01	0.7132	0.4790	0.6482
TJ.SIM.SinglevsCombo	0.024	0	51.01	0.7769	0.4408	0.6833
TJ.SIM.CCRvsTE	0.025	0	51.01	0.5783	0.5656	0.5542
TJ.SIM.AllnoNORvsAllNOR	0.000	0	51.01	0.0000	1.0000	0.4239

Note:
TT - ToolText, TJ - ToolJava, SH - Sherlock

Differences between no preprocessing and preprocessing

The first five interaction terms look at the effect of techniques (i.e., all PPTs combined) relative to when no PPT is used when comparing the same tools, as was done in the SOCO dataset. For the five contrasts the interaction difference was not significant for any of the comparisons. This means that using PPTs (compared to when no technique is used) affects all tested tool comparisons in the same way.

Based on the five contrasts the question arises as to whether the tool accuracy changes the same way when PPTs are used in comparison to when no technique is used. This indicates but does not confirm that using PPTs makes no difference. By examining the interaction graphs (Figure J.9) it is apparent that the accuracy increases for all tools.

To check H1, if there exists a difference in plagiarism detection accuracy between at least one PPT and when no PPT is used, a simple effects analysis was performed. The results of the simple effects analysis for RSS A1 are presented in Table 8.20 and the corresponding effect sizes in Table I.10. The analysis did confirm that using PPTs made a significant difference (at $p_b < 0.01$) compared with when no PPT is used for JPlag-text. On the other hand, for the other tools, the difference was not big enough to be considered significant. For the tool where the difference was significant, techniques had a positive effect. Although this fact is concerning, one should take into account the issue with the "*Participants*" and that all techniques were looked together (i.e., techniques were not individually compared to when no technique is used) and that the differences that are visible in Figure 8.18 are masked. For example as shown in the next section, for SIM-text the difference between single vs. combo techniques is significant ($p_b < 0.05$), so if the combo techniques alone would be compared to no PPT it is probable that the result would be significant.

Based on the presented results, it can still be stated that the H1 hypothesis is confirmed for the RSS A1 assignments.

Differences between different preprocessing techniques

In the previous analysis of the RSS A1 assignments it was confirmed that using PPTs significantly increases the accuracy for one tool. The problem with the previous analysis is that all techniques were put together. To overcome the problem the PPTs were analysed further in comparison to each other in the same way as for the SOCO dataset.

Regarding the interaction contrasts, as can be seen from Table 8.19 1 contrasts show significant differences at $p_b < 0.05$ and one was close with $p_b = 0.0647$. The contrasts that compare single and combo techniques do not show any significant results ($p_b > 0.05$). The 10 comparisons that compare single techniques (TE vs. CCR) and the comparisons that compare combo techniques (AllnoNOR vs. AllNOR) are significant $p_b < 0.05$ for 1 contrast. Significant results were obtained for contrast comparing single techniques for text vs. Java tools. The contrast comparing combo techniques for Sherlock-java vs. Other-java tools (i.e. JPlag-java and SIM-java

combined) is almost significant.

Since there was only 1 contrast that compares different techniques that is significant, it can be concluded that accuracy changes for most tools are similar when comparing different techniques. To check H2, if there exists a difference in plagiarism detection accuracy between at least two different PPTs, a simple effects analysis was used.

The simple effects analysis of single techniques compared to combo techniques showed a significant difference in accuracy with $p_b < 0.05$ for SIM-text. From the interaction graphs (Figure J.9 and Figure J.10) it can be seen that combo techniques show an increase in accuracy in comparison to a single techniques for all tools but only for SIM-text this was significant.

A simple effects analysis of TE vs. CCR techniques shows that the TE technique increases the accuracy for all tools except for Sherlock-java (Figure J.10) but only for JPlag-text was this significant. The simple effects analysis for the combo techniques (AllnoNOR vs. AllNOR) show that there is no difference between the two techniques for the SIM and JPlag tools (both versions had around $p_b = 0.5$). For Sherlock-java there is an increase and for Sherlock-text there is a decrease in accuracy but none were significant.

Based on the presented results it can be stated that the H2 hypothesis is confirmed on the RSS A1 assignments where for SIM-text and JPlag-text significant differences were observed.

8.3.2 Discussion

In the previous sections the H1 and H2 hypotheses were confirmed for the RSS dataset using the statistical analysis. To answer the research question Q1 and fulfil the second goal G2, the results of the statistical tests for the A1 assignments are qualitatively examined. The qualitative examination should help give more insight into the effects of the preprocessing techniques.

For the RSS dataset only one assignment group was used and it was significant only for JPlag-text two times and SIM-text one time. Nonetheless, the significant results are consistent with the results from the SOCO dataset, and that the best results were achieved for textual versions of JPlag and SIM. Another good thing is also (although not statistically significant) almost all preprocessing techniques gave an increase in the F1 value as it can be seen from the interaction graphs in Appendix J.

As already stated, the problem of the previous statistical analysis is the small dataset, and also that two years had to be excluded since no plagiarism occurred in those two academic years, but by using qualitative analysis one can get useful information also from those years. Before the two years are analysed as for the SOCO dataset, first a short discussion about the Precision and Recall is given for the four years that contained plagiarised matches.

In Appendix K graphs for Precision and Recall are given for the four analysed years in the RSS dataset. From these, one can see that in all four years there has been mostly an increase in Recall for all tools comparing no preprocessing and in at least one preprocessing technique. If one looks closer, the most promising techniques are TE and AllNOR. Note that AllnoNOR is almost the same as AllNOR for most tools, but because it has a positive effect on Sherlock the

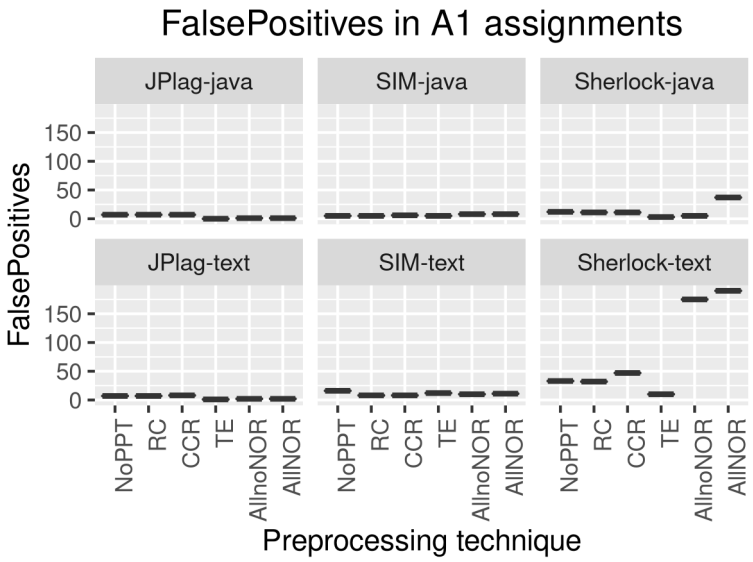


Figure 8.19: False positives for RSS A1 assignment in 2015-2016 with 3*IQR

focus is on the AllNOR.

In regards to Precision there is also a positive effect but not so often as for Recall. Based on these graphs it is concluded that the preprocessing techniques improve Recall more than Precision. In other words, they decrease the number of false negatives rather than decreasing false positives. In some extreme cases some techniques increase the false positives by a lot, for example in Sherlock-text the number of false positives goes up to 300 (e.g., in Figure 8.19 it was 190). Such a high increase is a problem since the marked matches can not all be verified in a reasonable amount of time. This problem might be solved by choosing a different method for the threshold level calculation but to verify this is beyond the scope of this research. For all other tools the number of marked matches does not go that high and it is mostly below 10. In general, it can be stated that the threshold was chosen well.

In Figure 8.19 and Figure 8.20 the false positive rate is presented for the two years that had no plagiarised matches so it was expected to get zero marked matches. In most cases PPTs reduces the number of false positives, which indicates that preprocessing is a good way to improve plagiarism detection.

In summary, based on the RSS dataset the guidelines that were written based on the SOCO dataset are valid with the modification, that instead of using the CCR technique, TE is suggested as a single technique. An even better solution is the combo technique AllnoNOR for JPlag and SIM, while for Sherlock it can be beneficial to use AllNOR in some cases, with the note that the number of marked matches does not increase too high. As in the SOCO dataset the textual versions of the tools seem to respond better to preprocessing and in some cases outperform or come close to the Java version when preprocessing techniques are used. This raises the questions of if the Java versions are necessary or can the same be achieved with preprocessing techniques and the textual versions of a tool.

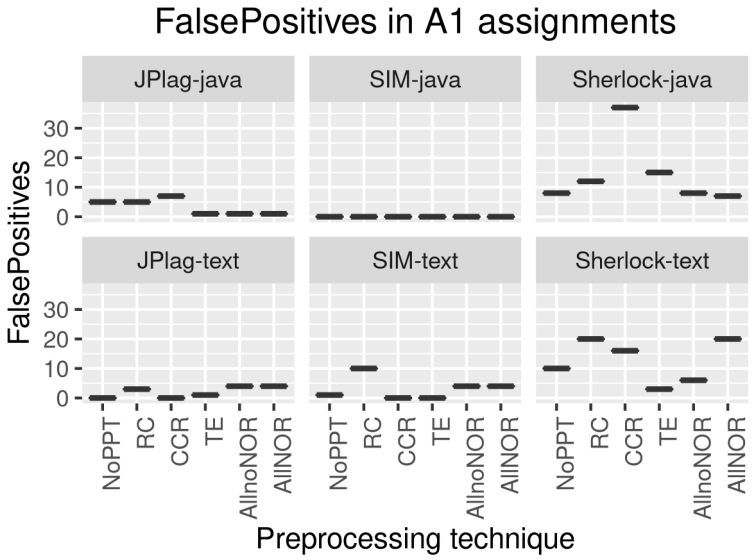


Figure 8.20: False positives for RSS A1 assignment in 2016-2017 with 3*IQR

8.3.3 Limitation of statistical analysis

To give better insight into the benefits of using preprocessing techniques and to overcome the limitations of the statistical analysis in this Section, some interesting observations on individual cases are presented. It can happen that there are no statistically significant differences between using a technique or not using a technique, but still there can be a positive effect of the technique which can be of importance to the teacher.

The first example is when the TE technique is used on the A4 assignment in 2014-2015. By using any tool if no technique is used the report gives above 90% similarity for some matches which dropped to less than 30% when TE is used and to 0% when AllNOR and AllnoNOR techniques are used. The reason was simple — students submitted only the professors' code which was given to them and generated only few getters and setters for the existing classes. Now this is easy to spot for the professor, but there were multiple such cases and even though it does not make a significant difference statistically the teacher still needs to look at such cases, so if the tool does not mark such cases it saves time for the teacher.

One very extreme case of such behaviour was observed in 2017-2018 when Sherlock-java was used. There were 15 matches with 100% and 4 with 99% which dropped below 20% when TE was used and below 5% when AllnoNOR was used. Although this is a lot and the teacher had to go through all such cases, that was not the main problem. The main problem was that the real cases of plagiarism were hidden by the false matches. There was a case with 72% similarity which was ranked 392 originally (which was far below the threshold) but it was ranked first when AllnoNOR technique was used and it was a case of real plagiarism.

This pattern repeats itself throughout all academic years for the A4 assignment. The reason why this pattern is occurring mostly in A4 and not in A1 is that A4 is an assignment with a lot of generated code and it is the last assignment in the course. When the A4 assignment is given

to the students many of them already have enough points to pass the course and do not even try to solve the problem; instead they just do some minor modification and graphical improvements which do not give them any additional points.

This shows that by evaluating techniques only statistically the improvement might be non-significant and one could conclude that they are of no use and that they only waste of resources. But by looking at the individual matches directly one can see there is a huge benefit of using a technique even if the improvement is not statistically significant. The only problem with such evaluation is that it can be biased, so it is good to first do the objective statistical evaluation, but if the results turn out non-significant it can be worthwhile to perform a qualitative evaluation.

8.4 Java or Textual version

The main focus of this thesis was to analyse the effect of preprocessing techniques. In this Section the idea is to take a look at the performance of tools without any techniques. It was already discussed in Chapter 4 that the three selected tools are the top tools used for plagiarism detection and that in different research projects they have been ranked differently. Because of that, it is hard to tell which tool is better, but there is one agreement between the different researchers, and that is *Java tools are better for source-code plagiarism detection then textual tools* since they are specialized for working with source-code. In this research, as it can be seen from the previously presented results the textual versions of the tools had often better results.

In all assignments from the SOCO dataset where there are any plagiarised matches (i.e., D1, D2, D3, D4, T1, C2), SIM-text is better than SIM-java, and Sherlock-text is better than Sherlock-java. In the case of JPlag, JPlag-text is better than JPlag-java in three SOCO assignments (D1, D4, C2). Additionally, when PPTs are used since they have a better impact on textual versions JPlag-text got better than JPlag-java in the D2 and D4 assignments. Even more interesting is that Sherlock-text and SIM-text have the best results in many assignments with the SOCO dataset. Similarly, for the RSS dataset the textual versions often come very close to the Java versions and sometimes they are better like in 2013-2014 the F1 value for JPlag-java was constantly zero while for JPlag-text it was 0.67 in combination with the TE technique. Another example is in 2014-2015 where SIM-text got the F1 value of 0.6 in combination with AllNOR technique and for SIM-java the maximum was the F1 value of 0.5 using the same PPT.

This finding is surprising since it is the opposite of what was expected and what was concluded from previous studies. After a more detailed examination of previous work what was noticed is that the previous research did not actually test textual versions of tools and compare them to the Java versions. Rather this "*fact*" is more of an expectation than an actual fact. In most research, although there is the division of tools into tools made to find plagiarism in textual documents and the tools specialized for source-code documents, researchers usually focus on only one type and then use the corresponding version.

On the other hand, there is a small amount of research that has tested both versions. In [157]

the authors tested JPlag-text, JPlag-java, SIM-text and SIM-java and based on their results one can see that JPlag-text ranked far better than JPlag-java. This confirms that textual versions might have more to offer than it is known. This does not mean that textual versions are always better — in fact in [157] SIM-java has been found to be better than SIM-text, which is the opposite what was found in this research.

A question that flows from this finding is what is the reason for the different results in this research and the results in [157], since the same tools were used and the same dataset (SOCO dataset). The answer lies in the tool's parameters. There is a difference in what parameter configuration was used to run the tools. Additionally, there is a difference as to how the threshold level was selected to calculate F1. It is out of the scope of this research to investigate this further but it is definitely an interesting finding that requires future work and shows the importance of parameters and the selection of the threshold level. This finding also raises other questions like "*How can it be that textual tools perform better than the specialized Java tools?*" and *Are specialized tools necessary at all?*.

8.5 Contributions

This Section lists all contributions (scientific and practical) of this thesis. The scientific contributions of this thesis are as follows.

- Comprehensive overview of the field "*source-code plagiarism detection in academia*". In combination with the results presented in [140] this research presents the largest full review of the field covering articles from 1980 to 2018.
- New definition of plagiarism in programming assignments.
- New parameter calibration method for similarity detection tools. Together with [138] this research presents the complete explanation and demonstration of how to use the calibration method and gives reasons why calibration is important. In addition, this method can be used in other fields like *clone detection*, *authorship attribution*, etc.
- Creation and evaluation of two new PPTs: CCR and TE. This research in combination with [95] presents the full description of the new TE technique. In addition, in this research two new combinations of techniques (i.e., AllNOR and AllnoNOR) were created and evaluated.
- Identification and explanation of the differences in plagiarism detection accuracy, for the tested tools and assignments, between cases in which the PPTs are used and those in which the PPTs are not used.
- Identification and explanation of the differences in plagiarism detection accuracy, for the tested tools and assignments, between different PPTs.

- Explanation of how PPTs affect the plagiarism detection accuracy of tested tools in different student programming assignments. This research shows that a significant improvement of plagiarism detection accuracy can be achieved by using PPTs on some tools and explains when this is the case on the open SOCO dataset and on the RSS dataset.
- Use-case analysis on real student programming assignments which shows the benefits of using PPTs (especially template exclusion). Analysing one assignment group from six academic years is a larger scale of real data than used in most of the previous studies.
- Analysis performed on larger files of real student programming assignments, meaning the solutions had from 500 and up to 3,900 LOC.
- New case study comparison of tools without preprocessing techniques on a new type of RSS dataset. This thesis showed some interesting results whereby the textual version of a tool is better than the specialized Java version.
- Application of designed experiment in plagiarism detection based on multiple assignments, years, techniques and tools.
- New method for determination of threshold level of positive and negative pairs (median +- 2/3 IQR) based on the similarity between pairs.
- New framework for performing experimental comparisons of similarity detection tools using statistical methods for objective results. Although the framework has not been formally evaluated it provides a set of steps that can be followed for performing other comparisons in the future, for example, it gives the description of how to solve the problem of normality (i.e., bootstraps) and similar issues.

Except for the scientific contributions there are some practical contributions of this thesis, which are the following.

- The Multiple Plagiarism Checker (MPC) system which can be used for benchmark and comparison of similarity detection tools (i.e., SIM, JPlag, and Sherlock), which is extensible to other tools. In addition, the system enables using various PPTs in combination with the tools, which is extensible to other techniques.
- The MPC system which can be used as a similarity detection tool which enables the user to chose one or more tools and use them all at once to check similarity between files. In addition, a web graphical interface and a command line interface were developed to enable easy usage of the MPC system on a daily basis.
- The calibration module integrated into the MPC system for calibrating configuration parameters of SIM, JPlag and Sherlock, which can be used to put tools in equal position before comparisons are performed.

- Guidelines for which PPT to use in combination with which similarity detection tool (i.e., SIM, JPlag and Sherlock).
- R scripts for performing statistical analysis when comparing tools which can be used in future research.
- The complete source code of the thesis using R scripts, LaTeX and Sweave files which can be used as examples in future research to generate various graphs, tables and even whole thesis in PDF format from R studio.

CHAPTER 9

FUTURE WORK

In the previous chapter many questions were left unanswered and these questions form the basis for future research. In this chapter an overview of possible future work is given which goes beyond the open questions.

There are many possible directions that one could take if interested in the field of source-code plagiarism. The first possible direction is to continue analysing the effect of preprocessing techniques. This means researching with new techniques, combining different techniques, adding new tools (for example MOSS), testing with other datasets, assignments, etc. Also, one could analyse in more detail the existing results and try to answer why the Precision and/or Recall was increased or decreased with some techniques or why some techniques did not have any effect on some assignments but affected other assignments.

The second direction can be to focus on the differences between the textual and Java versions of the tools and test in which cases the textual version in combination with preprocessing techniques outperform the Java versions, or one could analyse the impact of the configuration parameters on the quality of detection. Similarly one could analyse what is the impact of the selected threshold level on the comparison results or the effect of the threshold level on the tools' accuracy.

A third possible direction for future work is the area of plagiarism prevention rather than focusing only on detection. One could analyse the effects of motivation on plagiarism by using the findings from psychology, like "*A motivational approach to self: Integration in personality*" [37] or "*Using Future Authoring to Improve Student Outcomes*" [47]. The Future Authoring [1] program used at Mohawk College Hamilton already showed a 40% reduction in dropout according to Jordan B. Petterson [2], co-author of [47], so an interesting question would be whether that program also reduces plagiarism.

Other directions that would be possible for future work are for example: a) testing the correctness of the calibration method on other tools or with other datasets; b) expanding the MPC system to use graph representation and investigate the impact of graph visualization of the results; c) investigating the effect of similarity calculation in Sherlock if the calculation would be changed to not include the same lines from two different match parts; d) testing the effect of different inter-quartiles (IQR) multipliers on F1 measure and conclusions about using preprocessing techniques; e) repeating the experiments with Precision and Recall rather than using F-score; f) expanding the experiments to other programming languages.

[1] Online version of the future authoring program for personal evaluation is available at `https://www.selfauthoring.com/`

[2] Statement on Twitter on 4.11.2016 available at `https://twitter.com/jordanbpeterson/status/794686064871010304`

CHAPTER 10

CONCLUSION

Plagiarism detection is a difficult task and requires a lot of time and energy from the teacher to do it well. In order to help teachers find plagiarism, various similarity detection tools have been built, commonly called plagiarism detection tools. Although it is stated that the tools can find plagiarised cases automatically, it is not what they are doing. The tools only find similarity between two files (called matches) and mark the ones that are suspicious based on an implemented algorithm. Teachers, when they are using such a tool, no matter how good the tool is, should act as if the tool marked the wrong matches (false positives) and manually recheck all marked matches to ensure that no student is falsely accused of plagiarism.

Because of that, a new definition of plagiarism was given which includes the teacher's role in the process. Since there are different kinds of plagiarism the focus of this research was on plagiarism in student programming assignments in academia and the definition was given in this context. *Plagiarism, in programming assignments, is the act of taking a significant amount of source-code parts (up to the entire source-code) from other students or from the Internet and using it without noting its origin. A 'significant amount' means that the similarity between two solutions of a programming assignment is high enough that an expert (teacher, ethical board, etc.) considers specific student work as sufficiently 'real' plagiarism to accuse the student of plagiarism.*

Although the tools can not be used to state what is plagiarism and what not, they are of high value since in an environment with a large number of students and with more than one teacher since they can help identify the suspicious cases and save time for the teacher. Additionally, such tools can, with the right algorithm, uncover various obfuscation attempts done by students which would hide plagiarism if the detection is done manually by the teacher. A tool is considered useful when it has two properties, first, it needs to find and mark all suspicious matches, and second, it should mark only those matches that are considered real plagiarised after investigated by the teacher.

The quality of the tool is measured by the number of matches it misses marking as plagiarised that really were plagiarised (false negatives) and the number of matches that were marked as plagiarised but at the end were not cases of real plagiarism (false positives). If one knows the exact number of plagiarised matches, the measures Precision and Recall can be calculated. Since both measures are important to ease the comparison process of different tools, the F1 value can be calculated. Since no tool is perfect researchers always try to improve the tools.

In this research, the idea was to improve the quality of existing tools by preprocessing the submitted files using different preprocessing techniques, rather than building new tools. Although preprocessing has been used in the past by different tools to remove some noise

from the submissions, there has been limited research that would actually measure the effect of such techniques on different tools. In this research *preprocessing technique is understood as a technique which results in modified source-code or similar that can be used with different detection tools instead of being limited to only one tool.*

To measure the effect of preprocessing techniques an experiment was performed on student programming assignments using five techniques (i.e. Remove Comments (RC), Template Exclusion (TE), Common Code Remove (CCR), All techniques without Normalisation (AllnoNOR), and All techniques with Normalisation (AllNOR)) on six tools (i.e. SIM-text, SIM-java, JPlag-text, JPlag-java, Sherlock-text, Sherlock-java) with two different datasets (i.e. SOurce COde Reuse (SOCO) dataset and Real Student Solution (RSS) dataset). Since the experiment was done on programming assignments, specialized tools for detecting source-code plagiarism were used.

The tools were carefully chosen from the best tools that are mentioned in the scientific literature. Since each selected tool supports two modes, the experiment was done with both modes. The first mode is the specialized mode (called Java version) — made specifically to detect plagiarism in source-code, and the second mode is the textual mode (called text version) — made to detect plagiarism in normal text. The whole process of detection was automatized using the newly developed Multiple Plagiarism Checker (MPC) system, the output of which was then used to do statistical analysis which was automated using the system R.

There were three major issues before the experiment could be conducted. First, when performing comparisons with these tools the issue was how to put all tools in equal positions. Each tool has configuration parameters, and since it is not known what the best configuration is a problem arises which values to choose. In order to have an objective selection and not to put one tool in a better position to another, a calibration method was used to find out the optimal parameters. The calibration method uses one tool to calibrate the other on a calibration dataset. The main idea is that after the calibration (in an ideal situation) all tools will report the same similarities for the calibration dataset. Since this is not possible, the calibration method tries to find such parameter configuration where the tools report as much as possible similar results. To do the calibration the method uses a measure called calibration difference sum.

The second issue was the selection of preprocessing techniques. In the scientific literature, many techniques have been described and since it was not possible to test them all the most used techniques were selected. These techniques were further put through a technique selection test where the techniques which show the best effect on a small dataset were to be used in the experiment.

The third issue was the problem of calculating the threshold level. There are various ways how the threshold level can be selected it can be a fixed value or it can be flexible. Since all tools have different algorithms and all assignments can have different percentages, it is better to have a flexible threshold where the threshold is calculated based on the distribution of all matches. In this research the threshold is calculated as follows. First, the median similarity of all matches is

calculated and then the threshold level is set to 3 inter-quartiles from the median.

In this experiment two hypotheses were tested which are stated in the introduction chapter. The hypotheses were confirmed using statistical analysis using two different datasets. First is the SOCO dataset which is an open dataset containing seven assignments for which the plagiarised matches are known. The detection was performed on all seven assignments but the statistical analysis could be performed only on four of them. Second, the RSS dataset is a private dataset that contains data from four assignments in six years. Because it was not known which matches are plagiarised, an expert was needed to evaluate all marked matches manually according to the defined procedure (section 6.3.1). Because of that, the statistical analysis was performed only on the first assignment since manual verification of the matches is a long and tedious process.

In all five assignments on which the statistical analysis was performed the H1 and H2 hypotheses were confirmed, at least for one tool. In addition to the hypotheses there was one research questions (RQ1). To answer this question a qualitative analysis of the results was performed which gave more insight into the effects of the techniques.

In summary, from the quantitative and qualitative results, a correctly chosen PreProcessing Technique (PPT) in combination with the right tool (i.e., SIM-text, JPlag-text and Sherlock-java) can improve the accuracy of plagiarism detection. The experimental results, except from the fact that preprocessing techniques have a positive impact on the detection quality, also show that different techniques have a different impact, so it is important which preprocessing technique is chosen in combination with which tool. To be more precise, based on the results it can be stated that for SIM-text and JPlag-text the suggested technique is the AllnoNOR technique, while for Sherlock-java the suggested technique is the AllNOR technique. In some cases the techniques can have a none or a negative effect on some tools, and therefore it is suggested that for SIM-java, JPlag-java and Sherlock-text no PPT should be used.

Although the focus of this research was on the preprocessing technique, the most interesting finding of this research is perhaps about the textual version of the tools itself. At the beginning of the experiment it was expected that the Java versions would give the best results since they are made specifically to detect source-code plagiarism. It was therefore a surprise when at the end the textual versions performed better. The results of the experiment showed that preprocessing has a positive effect on the detection quality whereby the best results were achieved with JPlag-text and SIM-text. Although this is already a surprise since both are textual versions of the tools, an even bigger surprise was that the textual versions in many cases outperformed the Java version just by using preprocessing techniques.

Even though the emphasis of this research is on plagiarism detection, one must note that plagiarism detection is only the second part in the process of fighting plagiarism. Before detection one should consider preventing plagiarism in the first place and use plagiarism detection only as a measure in case the prevention techniques fail. A good teacher will always try to prevent plagiarism first and second do the detection just to make sure students are ethical.

It is important to remember that in the case of plagiarism the teacher is responsible for

their own ethical behaviour as well as the student is for his/hers. While the student does the plagiarism it is up to the teacher to find plagiarism and act ethically as well, which means informing the student that such behaviour is unacceptable, and explaining why this is wrong, but also to understand what led the student to plagiarise and try to help the student if possible so it does not happen in the future. With this, the teacher can help the student become a better person and a better member of society. It is unethical for the teacher to ignore plagiarism or falsely accuse a student of plagiarism. One should always remember the famous phrase "*the line dividing good and evil cuts through the heart of every human being.*" [1]

[1] Solzhenitsyn, Aleksandr. The Gulag Archipelago, 1918-56. Vintage Uk, 2002.

CHAPTER A

EXAMPLE OF CALIBRATION REPORT

```
0  CALIBRATION SUCCESSFUL
1
2  Calibrated Tool: JPlagJava
3  Base Tool: SIMGruneJava
4  Base Tool Params: r = 22,
5  CASES
6
7  Optimal params for all cases (minimal CDS): 84.104675
8  with Params: t = 9,
9
10 Case: 100Precent
11   Similarity Base Tool: 100.0
12   Optimal params similarity Calibrated: 100.0
13   Diff to base tool: 0.0
14   Best Similarity Calibrated: 100.0;   Best Params: t = 1,
15 Case: 100PrecentMixedComplex
16   Similarity Base Tool: 92.5
17   Optimal params similarity Calibrated: 72.54902
18   Diff to base tool: 19.950981
19   Best Similarity Calibrated: 87.745094;   Best Params: t = 1,
20 Case: 100PrecentMixedSimple
21   Similarity Base Tool: 100.0
22   Optimal params similarity Calibrated: 94.97207
23   Diff to base tool: 5.027931
24   Best Similarity Calibrated: 100.0;   Best Params: t = 1,
25 Case: 50Precent
26   Similarity Base Tool: 58.5
27   Optimal params similarity Calibrated: 51.764706
28   Diff to base tool: 6.7352943
29   Best Similarity Calibrated: 58.82353;   Best Params: t = 4,
30 Case: 50PrecentMixedComplex
31   Similarity Base Tool: 58.0
32   Optimal params similarity Calibrated: 50.0
33   Diff to base tool: 8.0
```

```
    Best Similarity Calibrated: 57.058823;   Best Params: t = 4,
Case: 50PrecentMixedSimple
    Similarity Base Tool: 67.0
    Optimal params similarity Calibrated: 49.079754
    Diff to base tool: 17.920246
    Best Similarity Calibrated: 67.484665;   Best Params: t = 3,
Case: soco0
    Similarity Base Tool: 14.5
    Optimal params similarity Calibrated: 0.0
    Diff to base tool: 14.5
    Best Similarity Calibrated: 16.666666;   Best Params: t = 5,
Case: soco1
    Similarity Base Tool: 7.5
    Optimal params similarity Calibrated: 6.912442
    Diff to base tool: 0.5875578
    Best Similarity Calibrated: 6.912442;   Best Params: t = 8,
Case: soco3
    Similarity Base Tool: 54.0
    Optimal params similarity Calibrated: 52.040817
    Diff to base tool: 1.9591827
    Best Similarity Calibrated: 52.040817;   Best Params: t = 8,
Case: soco4
    Similarity Base Tool: 100.0
    Optimal params similarity Calibrated: 100.0
    Diff to base tool: 0.0
    Best Similarity Calibrated: 100.0;   Best Params: t = 1,
Case: soco7
    Similarity Base Tool: 85.5
    Optimal params similarity Calibrated: 79.94543
    Diff to base tool: 5.554573
    Best Similarity Calibrated: 85.40246;   Best Params: t = 4,
Case: soco8
    Similarity Base Tool: 1.0
    Optimal params similarity Calibrated: 1.8887722
    Diff to base tool: 0.88877225
    Best Similarity Calibrated: 1.8887722;   Best Params: t = 9,
Case: 0Precent
    Similarity Base Tool: 0.0
    Optimal params similarity Calibrated: 0.0
```

```
  Diff to base tool: 0.0
  Best Similarity Calibrated: 0.0;   Best Params: t = 1,
Case: 0Precent2
  Similarity Base Tool: 0.0
  Optimal params similarity Calibrated: 0.0
  Diff to base tool: 0.0
  Best Similarity Calibrated: 0.0;   Best Params: t = 1,
Case: soco2
  Similarity Base Tool: 0.0
  Optimal params similarity Calibrated: 0.0
  Diff to base tool: 0.0
  Best Similarity Calibrated: 0.0;   Best Params: t = 1,
Case: soco5
  Similarity Base Tool: 0.0
  Optimal params similarity Calibrated: 0.0
  Diff to base tool: 0.0
  Best Similarity Calibrated: 0.0;   Best Params: t = 1,
Case: soco6
  Similarity Base Tool: 0.0
  Optimal params similarity Calibrated: 0.0
  Diff to base tool: 0.0
  Best Similarity Calibrated: 0.0;   Best Params: t = 1,
Case: soco9
  Similarity Base Tool: 0.0
  Optimal params similarity Calibrated: 2.9801323
  Diff to base tool: 2.9801323
  Best Similarity Calibrated: 0.0;   Best Params: t = 1,
 ALL COMBO PARAM DIFFS
  Param combo: t = 15, has total diff (CDS): 140.41025
  Param combo: t = 27, has total diff (CDS): 234.12029
  Param combo: t = 20, has total diff (CDS): 177.23363
  Param combo: t = 11, has total diff (CDS): 96.528244
  Param combo: t = 2, has total diff (CDS): 599.68134
  Param combo: t = 26, has total diff (CDS): 234.12029
  Param combo: t = 28, has total diff (CDS): 234.12029
  Param combo: t = 24, has total diff (CDS): 234.12029
  Param combo: t = 29, has total diff (CDS): 234.12029
  Param combo: t = 4, has total diff (CDS): 221.81158
  Param combo: t = 9, has total diff (CDS): 84.104675
```

```
112     Param combo: t = 8, has total diff (CDS): 93.469315
113     Param combo: t = 21, has total diff (CDS): 179.96214
114     Param combo: t = 7, has total diff (CDS): 110.32106
115     Param combo: t = 6, has total diff (CDS): 133.79118
116     Param combo: t = 14, has total diff (CDS): 133.5475
117     Param combo: t = 25, has total diff (CDS): 234.12029
118     Param combo: t = 17, has total diff (CDS): 149.36906
119     Param combo: t = 1, has total diff (CDS): 736.3351
120     Param combo: t = 22, has total diff (CDS): 204.25563
121     Param combo: t = 10, has total diff (CDS): 90.94165
122     Param combo: t = 3, has total diff (CDS): 329.10706
123     Param combo: t = 13, has total diff (CDS): 133.5475
124     Param combo: t = 23, has total diff (CDS): 230.9825
125     Param combo: t = 16, has total diff (CDS): 149.36906
126     Param combo: t = 19, has total diff (CDS): 177.23363
127     Param combo: t = 5, has total diff (CDS): 183.4447
128     Param combo: t = 18, has total diff (CDS): 158.86627
129     Param combo: t = 12, has total diff (CDS): 113.78314
```

CHAPTER B

SIM'S LICENCE

Copyright (c) 1986, 2007, Dick Grune, Vrije Universiteit, The Netherlands
All rights reserved.

Redistribution and use in source and binary forms,
with or without modification, are permitted provid
that the following conditions are met:

* Redistributions of source code must retain the above copyright notice, **this** list of conditions and the following disclaimer.

* Redistributions in binary form must reproduce the above copyright notice, **this** list of conditions and the following disclaimer in the documentation and/or other materials provided with the distribution.

* Neither the name of the Vrije Universiteit nor the names of its contributors may be used to endorse or promote products derived from **this** software without specific prior written permission.

THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS ``AS IS'' AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE ARE DISCLAIMED.
IN NO EVENT SHALL THE COPYRIGHT OWNER OR CONTRIBUTORS BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.

CHAPTER C

MPC SYSTEM ARCHITECTURE DETAILS

The main job of the MPC system wass to support the process of plagiarism detection described in Section 3.3. The system supports at least in part all phases from the prepare phase to the analysis phase. The system fully automates the prepare phase, the preprocessing phase, the detection phase and the visualization phase. Therefore, the main use-case of the MPC system enables the user to go from entering the input data for the detection to visualization of the results. To present the most important implementation parts of the system this use-case will be used.

The MPC system is divided into main modules, the core module and Web GUI module. The core module has no knowledge of the interface module which made it possible that the system also supports the CMD interface using the same internal logic, in addition to the Web GUI interface. Let us suppose the system is configured, and the assignment data have been downloaded and prepared in one directory. In the most simple form of the main use-case, the user goes to the summary website (Figure C.1), selects the wanted assignment, selects the wanted techniques and tools and clicks run. Once the detection is complete the results are presented in a table (Figure C.2) ready to be inspected in detail using side by side comparison (Figure C.3).

The logic starts with the user visiting the website whereby a session bean is created which creates all necessary objects. In the Figures that follow class diagrams [1] are presented, in the explanation of these diagrams the terms object and class will be used as synonyms to keep things simple, even though there is a difference between object and a class.

In Figure C.4 a class diagram starting with the session bean called *SummaryReportBean*. For Figure C.4 it can be seen that the bean creates a controller object, presenter object, and

[1] All diagrams where generated automatically by the ItelliJ IDEA IDE.

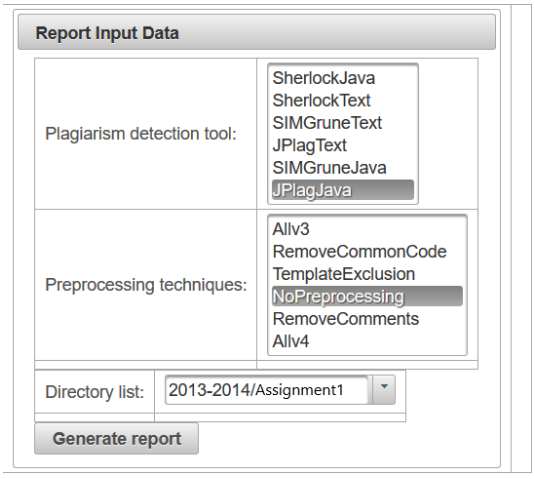


Figure C.1: Summary report main input interface

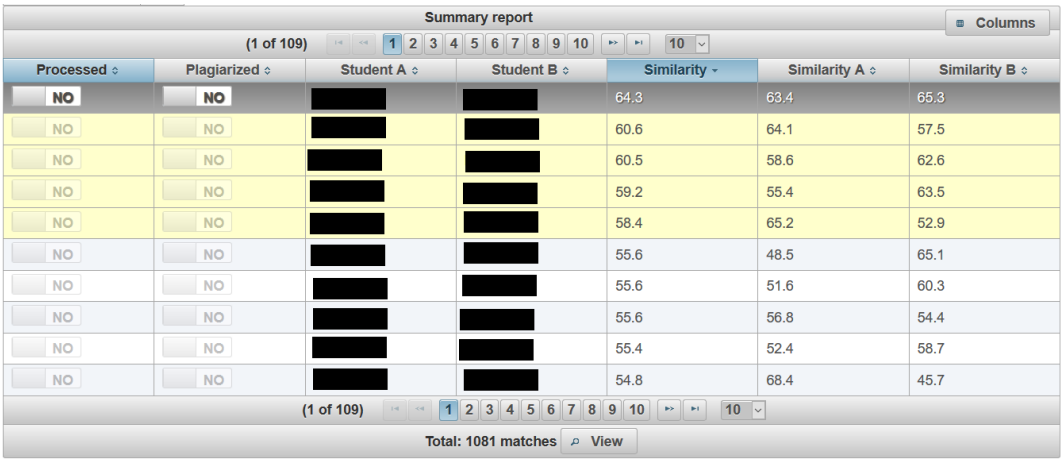


Figure C.2: Summary report table

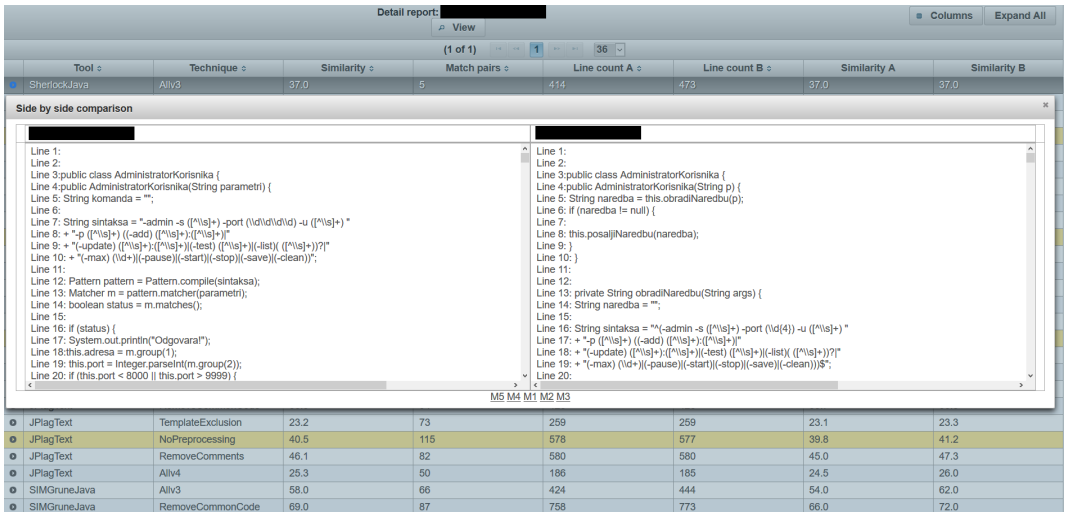


Figure C.3: Match side by side comparison

view model. These three objects will be responsible for executing the actions and presenting the data to the user. One can also see that there are two more objects, a use-case object and an object called *FactoryProvider*. The use-case object is the entrance to the MPC core module and it is located in the core module. The *FactoryProvider* is also located in the core module and is responsible for creating other factories (Figure C.8, Figure C.9, Figure C.7) needed in the system.

Since the presenter, the controller, the view model and the bean are in the MPC Web GUI module they communicate with the core module over the input and output boundary interface. Using the output boundary interface the use-case can communicate back to the GUI without knowing who is on the other side as long it implements the interface. Similarly the controller can send messages over the input boundary interface to the use-case without exactly knowing which use-case is on the other side, while this interface was not really necessary it was put to establish a clear boundary between the core module and the Web GUI module.

Once the user enters the data the controller will pass it to the use-case over the input boundary

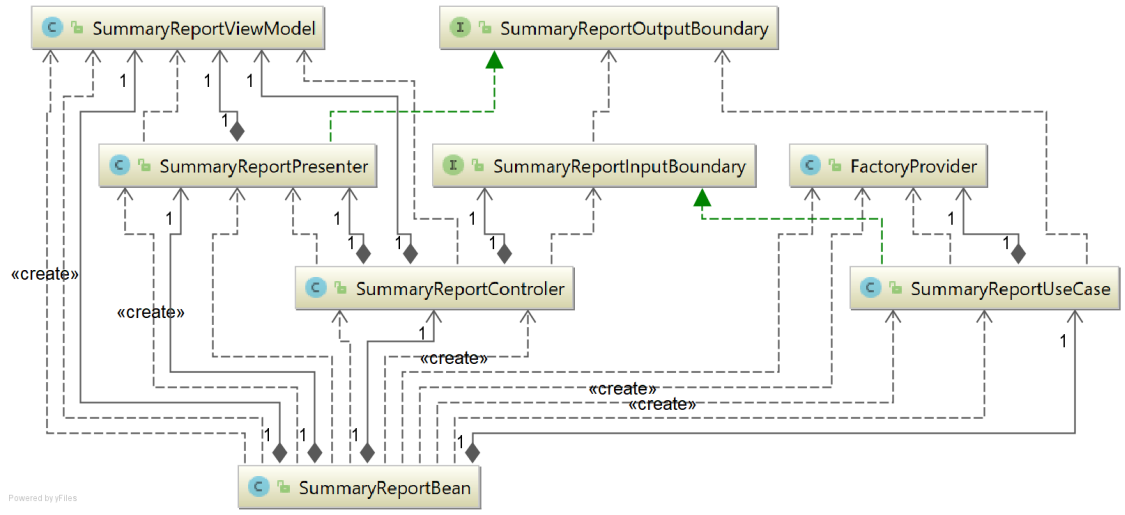


Figure C.4: Summary report class diagram - Web GUI module

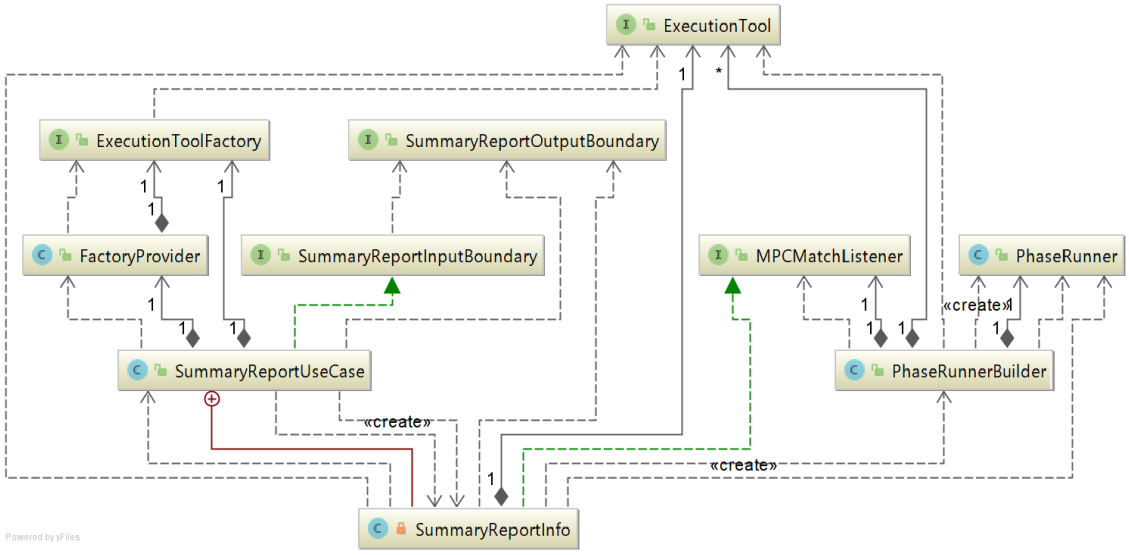


Figure C.5: Summary report class diagram - core module

interface. In Figure C.5 the class diagram is presented starting with the use-case object called *SummaryReportUseCase*. The use-case class has an inner class called *SummaryReportInfo*. *SummaryReportInfo* is just a helper class so to simplify things, let's say that the *SummaryReportInfo* and *SummaryReportUseCase* are one and the same class representing the use-case class. The use-case object creates the necessary factories (presented as *ExecutionToolFactory* interface) using the *FactoryProvider*, which then creates the necessary *ExecutionTools*. Also the use-case object creates the *PhaseRunner* object using the *PhaseRunnerBuilder* and implements the *MPCMatchListener* interface. The *PhaseRunner* is responsible for executing the prepare, preprocessing and detection phases (Figure C.6). Once all detections are done the *PhaseRunner* starts a phase to read all match files and report back to the use-case over the *MPCMatchListener* interface, which then creates the result report and passes it to the presenter over the output boundary interface.

In Figure C.6 the class diagram starting with *PhaseRunnerBuilder* is presented. The *PhaseRunnerBuilder* is responsible for building the *PhaseRunner* object. To create the object it uses the *PhaseFactory* which is created by the *FactoryProvider* (Figure C.7). *PhaseFactory* creates objects that represent the different phases. The *ExecutionPhase* object is used for preprocessing and for the detection phase, whereby the individual tools are represented over the *ExecutionTool* interface. To read the matches at the end the *MPCMatchReaderPhase* object is used which communicates over the *MPCMatchListener* interface to the use-case.

The first phase (prepare phase) is divided into three separate phases, presented in Figure C.7, which are represented by *ArchiveExtractor* object, *Renamer* object and *SubmissionFilesUnifier* object. All three objects are the so called prepare tools presented through the *PrepareTools* interface, which is a specialisation of the overall *ExecutionTool* interface.

The tools that are used in the preprocessing phase and detection phase are created using the corresponding factories (represented by *ExecutionToolFactory* interface): *SimilarityDetectionToolFactory* – presented in Figure C.8, and *PreprocessingTechniqueFactory* – presented in Figure C.9. These tools are passed to the *PhaseRunnerBuilder* by the use-case since every time different tools and techniques can be selected by the user, in comparison the prepare tools are not passed in, rather they are created by the *PhaseFactory* during execution since they are always the same.

Each detection tool, as can be seen in Figure C.8, has their adapters which run the actual tools. Since tools have Java and text versions there are adapters for each version. Each adapter represents in the system the concept of similarity detection tool using the *SimilarityDetectionTool* interface.

Similarly, each technique as can be seen in Figure C.9 has one object representing it. Since some techniques that are used are implemented in the tool Sherlock they are using the same *SherlockAdapter* object as the detection phase. Each object presenting a technique can be recognized by the implementation of the *PreprocessingTechnique* interface. A special technique is the combo technique presented by the class *ComboPreporcessingTechnique* which enables creation of new techniques by combining the individual techniques in any order. Combo techniques are created using the combo use-case, and once created they can be used by any other use-case through the *PreprocessingTechniqueFactory*.

Both the *SimilarityDetectionTool* interface and the *PreprocessingTechnique* interface are specialisations of the *ExecutionTool*, same as the *PrepareTools* interface. To add a new technique or tool only a specialized class needs to be implemented that has the corresponding interface, and the name of the new tool or technique needs to be added to the list of available tools or techniques in the corresponding factory. In the current implementation of the MPC system, the tool Spector [117] is supported, but Spector itself has some issues with processing larger files.

Throughout the class diagrams in Figures C.6, C.7, C.8, and C.9, all main classes of the core module are presented. In figure Figure C.5 an example of the main use-case using these core modules is presented, and in Figure C.4 is presented how this use-case is accessed from the

outside. Note that the same name appearing in the different diagrams is actually representing the same class, to simplify the explanations and because of space issues the whole diagram was split in parts.

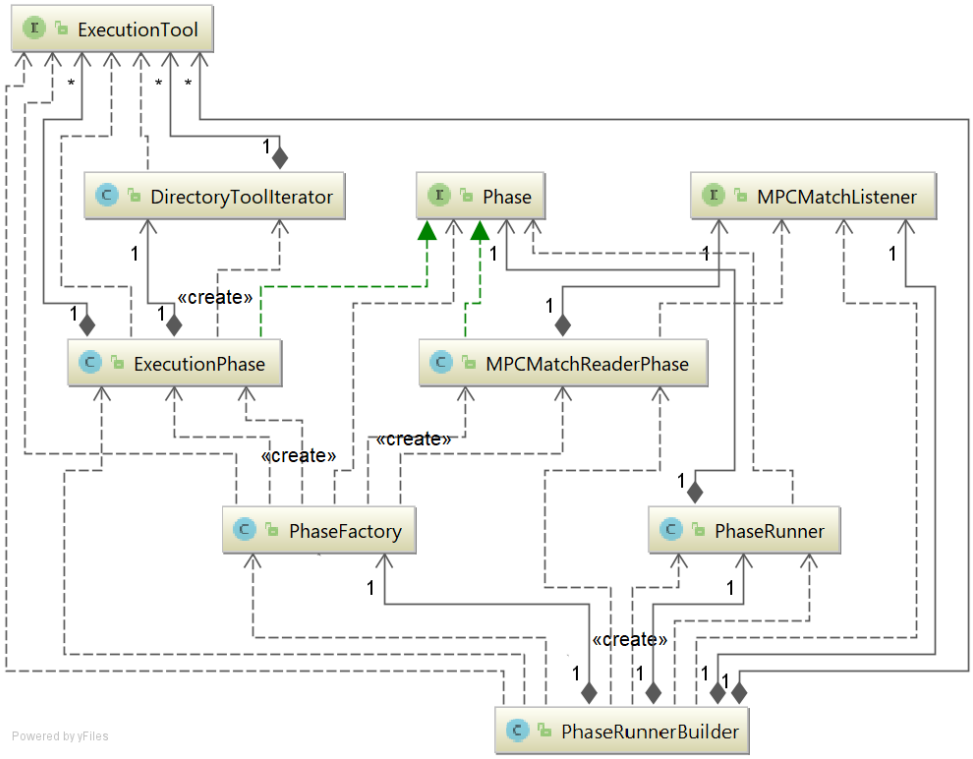


Figure C.6: Phase creation class diagram

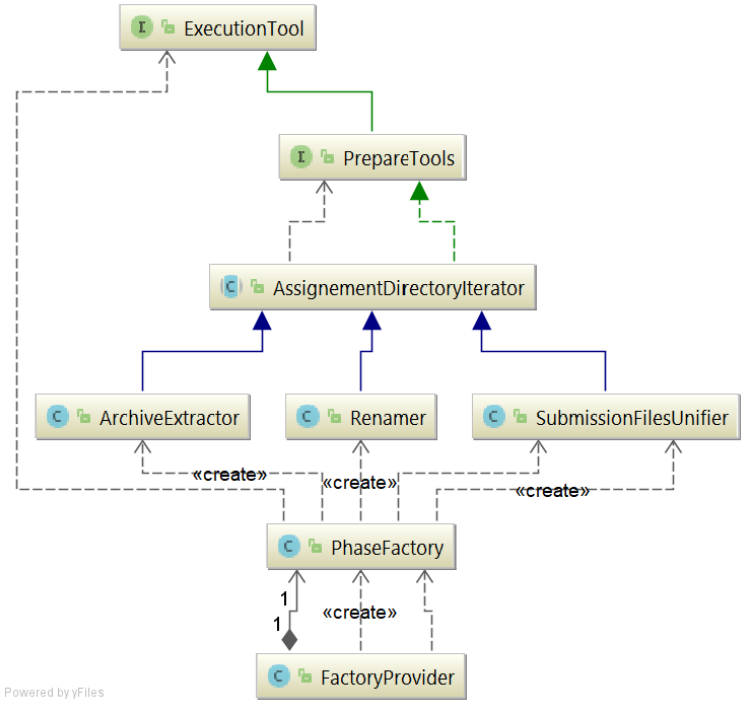


Figure C.7: Prepare tools creation class diagram

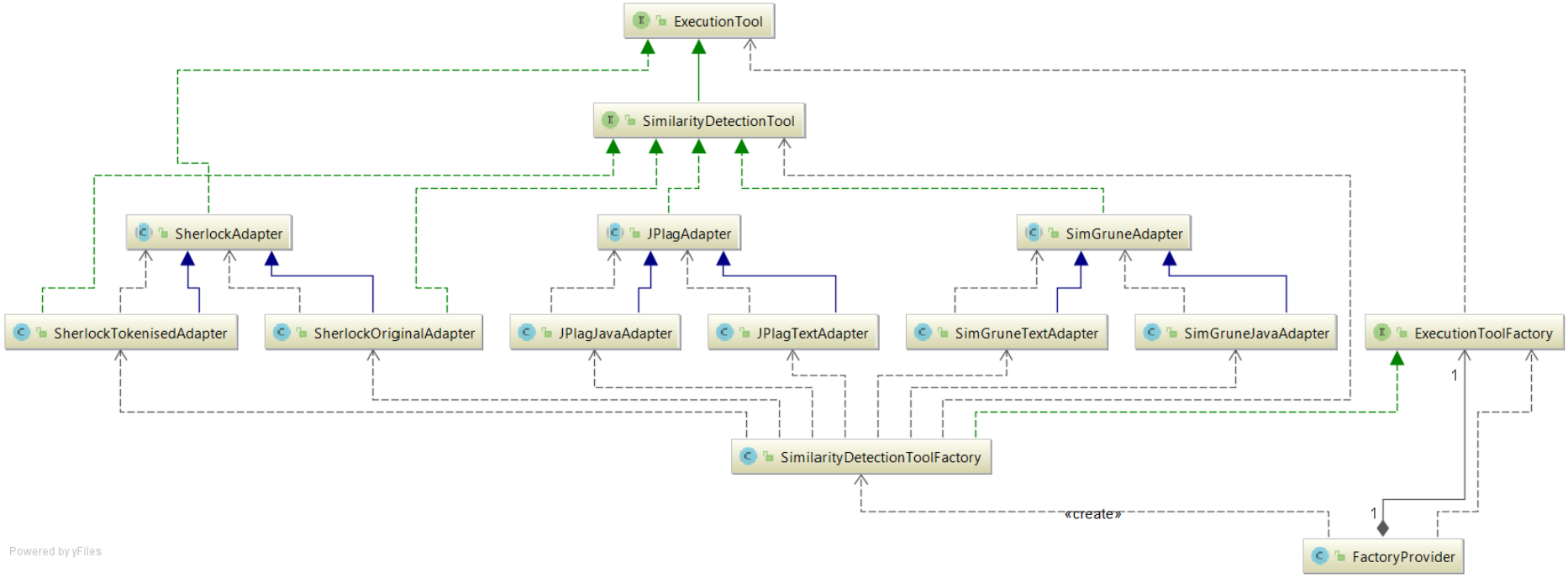


Figure C.8: Detection tools creation class diagram

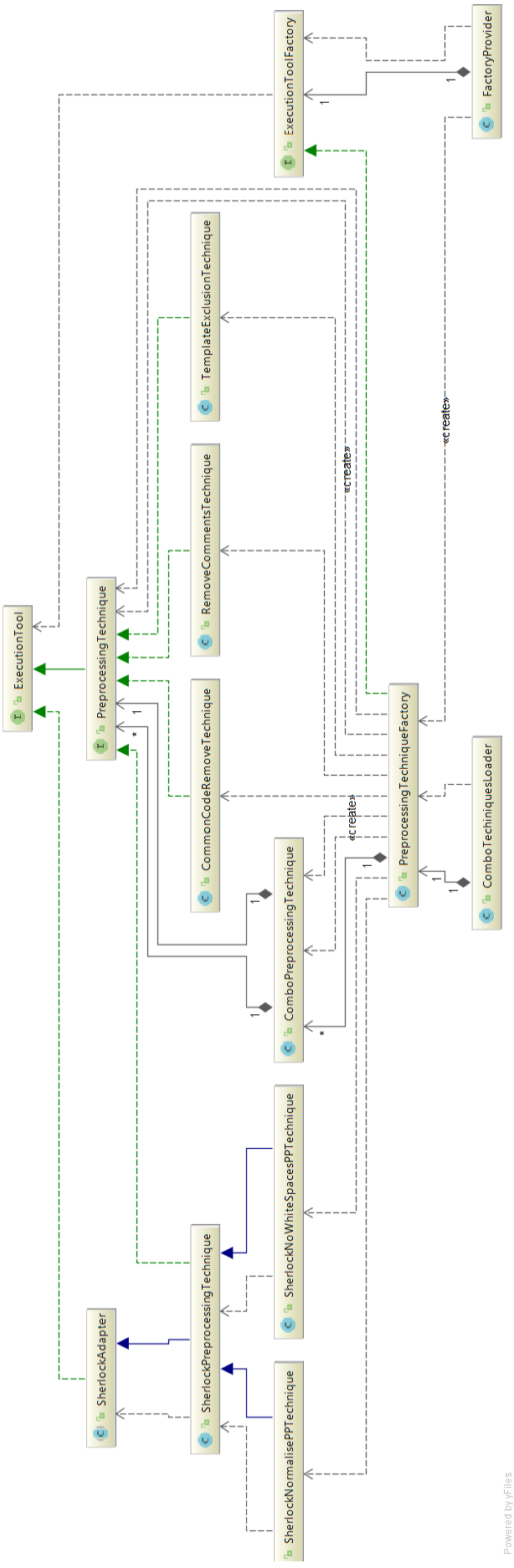


Figure C.9: Preprocessing techniques creation class diagram

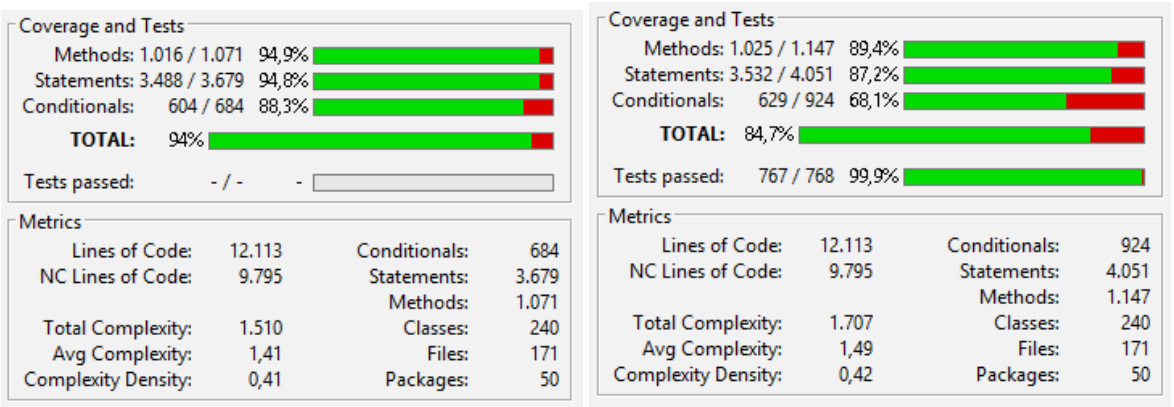
CHAPTER D

MPC SYSTEM COVERAGE REPORT

The code coverage is based on Clover's coverage report [1] is a total of 84% (69.1% of branches, 86.8% of statements and 86% of methods) if models are included and total of 94% (88.3% of branches, 94.8% of statements and 94.9% of methods) if models are excluded. Models are simple classes containing only public fields and two functions (equals, toString) which are used as data transfer objects (DTO), to pass data from one class to another.

Both coverages include various methods that are used by the GUI and where intentionally skipped so the real coverage is a bit higher. The 84% summary is presented in Figure D.1a and the 94% summary in Figure D.1b. The full report with models (84%) is presented at the end of the appendix. There is a bug in the Clover so there is a small difference between the full report and the summary report for the 84% coverage.

The coverage report does not include the template exclusion technique implementation since it is treated as external implementation as the preprocessing techniques in Sherlock.



(a) Without models **(b)** With models

Figure D.1: Coverage report summary

[1] `https://openclover.org/` - used version 4.2.0

Clover Coverage Report
MPC Coverage Report
Coverage timestamp: pet lis 5 2018 09:40:30 CEST

project stats:				
LOC:	15.025	**Methods:**	1.333
NCLOC:	12.414	**Classes:**	268
Files:	196	**Pkgs:**	55

	Branch	Stmt	Method	Total
Clover database pet lis 5 2018 09:37:06 CEST	69,1%	86,8%	86%	**84%**

Packages	Branch	Stmt	Method	Total
org.foi.mpc.executiontools.detectionTools	-	-	-	**-**
org.foi.mpc.usecases	-	-	-	**-**
default-pkg	-	0%	0%	**0%**
org.foi.mpc.executiontools.calibrator.models	0%	0%	0%	**0%**
org.foi.mpc.usecases.reports.pptestreport.models	0%	0%	0%	**0%**
org.foi.mpc.usecases.reports.statisticsReport.models	-	0%	0%	**0%**
org.foi.mpc.usecases.reports.summaryReport.models	0%	0%	0%	**0%**
org.foi.mpc.usecases.toolCalibration.models	16,7%	18,8%	20%	**18%**
org.foi.mpc.usecases.multipleDetecion.models	25%	28,6%	25%	**26,9%**
org.foi.mpc.summaryReport.view	7,1%	42,3%	62,5%	**42,3%**
org.foi.mpc	87,5%	61,5%	20,9%	**51,4%**
org.foi.mpc.usecases.combotechnique.models	50%	56,2%	50%	**53,3%**
org.foi.mpc.usecases.reports.avalibleTools.models	50%	57,1%	50%	**53,8%**
org.foi.mpc.usecases.reports.pptestreport.view.models	0%	67,4%	83,3%	**66,7%**
org.foi.mpc.phases.executionphases	42,3%	74%	100%	**73,9%**
org.foi.mpc.usecases.combotechnique.view.models	41,7%	80%	88,9%	**77,7%**
org.foi.mpc.executiontools.calibrator	46,3%	84,3%	91,7%	**78,1%**
org.foi.mpc.executiontools.techniques	56,2%	82,3%	92,5%	**79,9%**
org.foi.mpc.matches	65,2%	83,3%	94,3%	**80,5%**
org.foi.mpc.pptestreport.view	-	85,7%	72,7%	**82,1%**
org.foi.mpc.main	80%	86,5%	70,6%	**82,8%**
org.foi.mpc.usecases.reports.view.models	50%	83,6%	85,5%	**83,3%**
org.foi.mpc.executiontools.spies	-	89,5%	75%	**83,9%**
org.foi.common.filesystem.file	77,8%	90,4%	81,5%	**86,7%**
org.foi.mpc.usecases.reports.summaryReport.view.models	16,7%	86,5%	94,4%	**87,1%**
org.foi.mpc.executiontools.techniques.sherlock	50%	92,3%	92,3%	**90,2%**
org.foi.mpc.phases.readerphase	100%	92,5%	78,6%	**90,4%**
org.foi.mpc.usecases.reports.statisticsReport	85,3%	93,4%	87,5%	**91,3%**
org.foi.mpc.usecases.reports.statisticsReport.view.model	-	92,9%	92,9%	**92,9%**
org.foi.common.filesystem.directory	91,7%	93,3%	93,1%	**93%**
org.foi.mpc.executiontools.detectionTools.spector	90%	96,4%	88,9%	**94,2%**
org.foi.mpc.statisticsReport.view	50%	100%	84,6%	**94,4%**
org.foi.mpc.executiontools.detectionTools.simgrune	88,6%	95%	97%	**94,5%**
org.foi.mpc.executiontools.detectionTools.sherlock	93,8%	95,8%	95,8%	**95,6%**
org.foi.mpc.executiontools.prepareTools	97,1%	95,1%	97,2%	**96%**
org.foi.mpc.phases.executionphases.spies	-	94,4%	100%	**96,6%**
org.foi.common	100%	97,7%	92,3%	**97,2%**
org.foi.mpc.usecases.reports.summaryReport	97,8%	97,3%	97,8%	**97,4%**
org.foi.mpc.executiontools.detectionTools.JPlag	75%	99%	100%	**97,8%**
org.foi.mpc.executiontools.factories	100%	98,1%	100%	**98,7%**
org.foi.mpc.usecases.reports.summaryReport.view	92,3%	100%	100%	**98,7%**
org.foi.mpc.usecases.reports.statisticsReport.view	83,3%	100%	100%	**98,8%**
org.foi.mpc.usecases.multipleDetecion	94,4%	100%	100%	**98,8%**
org.foi.mpc.abstractfactories	-	100%	100%	**100%**
org.foi.mpc.matches.models	100%	100%	100%	**100%**
org.foi.mpc.phases	-	100%	100%	**100%**
org.foi.mpc.phases.runner	100%	100%	100%	**100%**

org.foi.mpc.usecases.combotechnique	100%	100%	100%	**100%**
org.foi.mpc.usecases.combotechnique.view	-	100%	100%	**100%**
org.foi.mpc.usecases.reports	-	100%	100%	**100%**
org.foi.mpc.usecases.reports.avalibleTools	100%	100%	100%	**100%**
org.foi.mpc.usecases.reports.pptestreport	100%	100%	100%	**100%**
org.foi.mpc.usecases.reports.pptestreport.view	100%	100%	100%	**100%**
org.foi.mpc.usecases.toolCalibration	100%	100%	100%	**100%**
org.foi.mpc.usecases.toolCalibration.view	100%	100%	100%	**100%**

CHAPTER E

CONTRAST CODINGS

To have the planned comparisons as described in Section 8.1.2 they need to be coded. The contrasts are coded in a way that the sum of products of contrasts ends up zero. "*If the products add to zero then we can be sure that the contrasts are independent or orthogonal. It is important for interperation that contrasts are orthogonal.*"[46, p. 421]

In Table E.1 contrast codings for tool variable are presented which are used for analysing SOCO and student dataset. In Table E.2 and Table E.3 contrast codigs for techniques are presented. Table E.2 are te codings for SOCO dataset and Table E.3 are codings for the student dataset.

Table E.1: Tool contrast codings

	Text			**Java**		
Tool	**JPlag**	**SIM**	**Sherlock**	**JPlag**	**SIM**	**Sherlock**
TextvsJava	1	1	1	-1	-1	-1
Text.SherlockvsOthers	0	0	0	1	1	-2
Java.SherlockvsOthers	1	1	-2	0	0	0
Text.SIMvsJPlag	0	0	0	1	-1	0
Java.SIMvsJPlag	1	-1	0	0	0	0

Table E.2: SOCO technique contrast codings

Technique	**NoPPT**	**RC**	**CCR**	**AllnoNOR**	**AllNOR**
NoPPTvsPPT	-4	1	1	1	1
SinglevsCombo	0	-1	-1	1	1
RCvsCCR	0	-1	1	0	0
AllnoNORvsAllNOR	0	0	0	-1	1

Table E.3: Student technique contrast codings

Technique	**NoPPT**	**CCR**	**TE**	**AllnoNOR**	**AllNOR**
NoPPTvsPPT	-4	1	1	1	1
SinglevsCombo	0	-1	-1	1	1
CCRvsTE	0	-1	1	0	0
AllnoNORvsAllNOR	0	0	0	-1	1

CHAPTER F

CONTRAST CODINGS FOR THE SIMPLE EFFECS ANALYSIS

To be able to do the simple effects analysis as described in Section 8.1.2 contrasts need to be created and coded. The contrasts are coded in the same way (i.e., as orhogonal contrasts) as in Appendix E.

In Table F.1, Table F.2, and Table F.2 contrast codings for analysing SOCO dataset are presented. The contrast names for SOCO dataset are: C1 - TextvsJava, C2 - TT.SHvsOthers, C3 - TT.SIMvsJPlag, C4 - TJ.SHvsOthers, C5 - TJ.SIMvsJPlag, C6 - TT.SH.NoNPPvsNPP, C7 - TT.SH.SinglevsCombo, C8 - TT.SH.RCvsCCR, C9 - TT.SH.AllnoNORvsAllNOR, C10 - TT.JPlag.NoNPPvsNPP, C11 - TT.JPlag.SinglevsCombo, C12 - TT.JPlag.RCvsCCR, C13 - TT.JPlag.AllnoNORvsAllNOR, C14 - TT.SIM.NoNPPvsNPP, C15 - TT.SIM.SinglevsCombo, C16 - TT.SIM.RCvsCCR, C17 - TT.SIM.AllnoNORvsAllNOR, C18 - TJ.SH.NoNPPvsNPP, C19 - TJ.SH.SinglevsCombo, C20 - TJ.SH.RCvsCCR, C21 - TJ.SH.AllnoNORvsAllNOR, C22 - TJ.JPlag.NoNPPvsNPP, C23 - TJ.JPlag.SinglevsCombo, C24 - TJ.JPlag.RCvsCCR, C25 - TJ.JPlag.AllnoNORvsAllNOR, C26 - TJ.SIM.NoNPPvsNPP, C27 - TJ.SIM.SinglevsCombo, C28 - TJ.SIM.RCvsCCR, C29 - TJ.SIM.AllnoNORvsAllNOR,
C29 - TJ.SIM.AllnoNORvsAllNOR.

In Table F.4, Table F.5, and Table F.6 contrast codings for analysing student dataset are presented. The contrast names for student dataset are: C1 - TextvsJava, C2 - TT.SHvsOthers, C3 - TT.SIMvsJPlag, C4 - TJ.SHvsOthers, C5 - TJ.SIMvsJPlag, C6 - TT.SH.NoNPPvsNPP, C7 - TT.SH.SinglevsCombo, C8 - TT.SH.CCRvsTE, C9 - TT.SH.AllnoNORvsAllNOR, C10 - TT.JPlag.NoNPPvsNPP, C11 - TT.JPlag.SinglevsCombo, C12 - TT.JPlag.CCRvsTE, C13 - TT.JPlag.AllnoNORvsAllNOR, C14 - TT.SIM.NoNPPvsNPP, C15 - TT.SIM.SinglevsCombo, C16 - TT.SIM.CCRvsTE, C17 - TT.SIM.AllnoNORvsAllNOR, C18 - TJ.SH.NoNPPvsNPP, C19 - TJ.SH.SinglevsCombo, C20 - TJ.SH.CCRvsTE, C21 - TJ.SH.AllnoNORvsAllNOR, C22 - TJ.JPlag.NoNPPvsNPP, C23 - TJ.JPlag.SinglevsCombo, C24 - TJ.JPlag.CCRvsTE, C25 - TJ.JPlag.AllnoNORvsAllNOR, C26 - TJ.SIM.NoNPPvsNPP, C27 - TJ.SIM.SinglevsCombo, C28 - TJ.SIM.CCRvsTE, C29 - TJ.SIM.AllnoNORvsAllNOR,
C29 - TJ.SIM.AllnoNORvsAllNOR.

Table F.1: Simple effects analysis contrast codings - SOCO dataset - part1

	C1	C2	C3	C4	C5	C6	C7	C8	C9	C10	C11
JPlag-java and AllnoNOR	1	0	0	1	1	0	0	0	0	0	0
JPlag-java and AllNOR	1	0	0	1	1	0	0	0	0	0	0
JPlag-java and CCR	1	0	0	1	1	0	0	0	0	0	0
JPlag-java and NoPPT	1	0	0	1	1	0	0	0	0	0	0
JPlag-java and RC	1	0	0	1	1	0	0	0	0	0	0
JPlag-text and AllnoNOR	-1	1	1	0	0	0	0	0	0	1	1
JPlag-text and AllNOR	-1	1	1	0	0	0	0	0	0	1	1
JPlag-text and CCR	-1	1	1	0	0	0	0	0	0	1	-1
JPlag-text and NoPPT	-1	1	1	0	0	0	0	0	0	-4	0
JPlag-text and RC	-1	1	1	0	0	0	0	0	0	1	-1
Sherlock-java and AllnoNOR	1	0	0	-2	0	0	0	0	0	0	0
Sherlock-java and AllNOR	1	0	0	-2	0	0	0	0	0	0	0
Sherlock-java and CCR	1	0	0	-2	0	0	0	0	0	0	0
Sherlock-java and NoPPT	1	0	0	-2	0	0	0	0	0	0	0
Sherlock-java and RC	1	0	0	-2	0	0	0	0	0	0	0
Sherlock-text and AllnoNOR	-1	-2	0	0	0	1	1	0	-1	0	0
Sherlock-text and AllNOR	-1	-2	0	0	0	1	1	0	1	0	0
Sherlock-text and CCR	-1	-2	0	0	0	1	-1	1	0	0	0
Sherlock-text and NoPPT	-1	-2	0	0	0	-4	0	0	0	0	0
Sherlock-text and RC	-1	-2	0	0	0	1	-1	-1	0	0	0
SIM-java and AllnoNOR	1	0	0	1	-1	0	0	0	0	0	0
SIM-java and AllNOR	1	0	0	1	-1	0	0	0	0	0	0
SIM-java and CCR	1	0	0	1	-1	0	0	0	0	0	0
SIM-java and NoPPT	1	0	0	1	-1	0	0	0	0	0	0
SIM-java and RC	1	0	0	1	-1	0	0	0	0	0	0
SIM-text and AllnoNOR	-1	1	-1	0	0	0	0	0	0	0	0
SIM-text and AllNOR	-1	1	-1	0	0	0	0	0	0	0	0
SIM-text and CCR	-1	1	-1	0	0	0	0	0	0	0	0
SIM-text and NoPPT	-1	1	-1	0	0	0	0	0	0	0	0
SIM-text and RC	-1	1	-1	0	0	0	0	0	0	0	0

Table F.2: Simple effects analysis contrast codings - SOCO dataset - part2

	C12	C13	C14	C15	C16	C17	C18	C19	C20
JPlag-java and AllnoNOR	0	0	0	0	0	0	0	0	0
JPlag-java and AllNOR	0	0	0	0	0	0	0	0	0
JPlag-java and CCR	0	0	0	0	0	0	0	0	0
JPlag-java and NoPPT	0	0	0	0	0	0	0	0	0
JPlag-java and RC	0	0	0	0	0	0	0	0	0
JPlag-text and AllnoNOR	0	-1	0	0	0	0	0	0	0
JPlag-text and AllNOR	0	1	0	0	0	0	0	0	0
JPlag-text and CCR	1	0	0	0	0	0	0	0	0
JPlag-text and NoPPT	0	0	0	0	0	0	0	0	0
JPlag-text and RC	-1	0	0	0	0	0	0	0	0
Sherlock-java and AllnoNOR	0	0	0	0	0	0	1	1	0
Sherlock-java and AllNOR	0	0	0	0	0	0	1	1	0
Sherlock-java and CCR	0	0	0	0	0	0	1	-1	1
Sherlock-java and NoPPT	0	0	0	0	0	0	-4	0	0
Sherlock-java and RC	0	0	0	0	0	0	1	-1	-1
Sherlock-text and AllnoNOR	0	0	0	0	0	0	0	0	0
Sherlock-text and AllNOR	0	0	0	0	0	0	0	0	0
Sherlock-text and CCR	0	0	0	0	0	0	0	0	0
Sherlock-text and NoPPT	0	0	0	0	0	0	0	0	0
Sherlock-text and RC	0	0	0	0	0	0	0	0	0
SIM-java and AllnoNOR	0	0	0	0	0	0	0	0	0
SIM-java and AllNOR	0	0	0	0	0	0	0	0	0
SIM-java and CCR	0	0	0	0	0	0	0	0	0
SIM-java and NoPPT	0	0	0	0	0	0	0	0	0
SIM-java and RC	0	0	0	0	0	0	0	0	0
SIM-text and AllnoNOR	0	0	1	1	0	-1	0	0	0
SIM-text and AllNOR	0	0	1	1	0	1	0	0	0
SIM-text and CCR	0	0	1	-1	1	0	0	0	0
SIM-text and NoPPT	0	0	-4	0	0	0	0	0	0
SIM-text and RC	0	0	1	-1	-1	0	0	0	0

Table F.3: Simple effects analysis contrast codings - SOCO dataset - part3

	C21	C22	C23	C24	C25	C26	C27	C28	C29
JPlag-java and AllnoNOR	0	1	1	0	-1	0	0	0	0
JPlag-java and AllNOR	0	1	1	0	1	0	0	0	0
JPlag-java and CCR	0	1	-1	1	0	0	0	0	0
JPlag-java and NoPPT	0	-4	0	0	0	0	0	0	0
JPlag-java and RC	0	1	-1	-1	0	0	0	0	0
JPlag-text and AllnoNOR	0	0	0	0	0	0	0	0	0
JPlag-text and AllNOR	0	0	0	0	0	0	0	0	0
JPlag-text and CCR	0	0	0	0	0	0	0	0	0
JPlag-text and NoPPT	0	0	0	0	0	0	0	0	0
JPlag-text and RC	0	0	0	0	0	0	0	0	0
Sherlock-java and AllnoNOR	-1	0	0	0	0	0	0	0	0
Sherlock-java and AllNOR	1	0	0	0	0	0	0	0	0
Sherlock-java and CCR	0	0	0	0	0	0	0	0	0
Sherlock-java and NoPPT	0	0	0	0	0	0	0	0	0
Sherlock-java and RC	0	0	0	0	0	0	0	0	0
Sherlock-text and AllnoNOR	0	0	0	0	0	0	0	0	0
Sherlock-text and AllNOR	0	0	0	0	0	0	0	0	0
Sherlock-text and CCR	0	0	0	0	0	0	0	0	0
Sherlock-text and NoPPT	0	0	0	0	0	0	0	0	0
Sherlock-text and RC	0	0	0	0	0	0	0	0	0
SIM-java and AllnoNOR	0	0	0	0	0	1	1	0	-1
SIM-java and AllNOR	0	0	0	0	0	1	1	0	1
SIM-java and CCR	0	0	0	0	0	1	-1	1	0
SIM-java and NoPPT	0	0	0	0	0	-4	0	0	0
SIM-java and RC	0	0	0	0	0	1	-1	-1	0
SIM-text and AllnoNOR	0	0	0	0	0	0	0	0	0
SIM-text and AllNOR	0	0	0	0	0	0	0	0	0
SIM-text and CCR	0	0	0	0	0	0	0	0	0
SIM-text and NoPPT	0	0	0	0	0	0	0	0	0
SIM-text and RC	0	0	0	0	0	0	0	0	0

Table F.4: Simple effects analysis contrast codings - student dataset - part1

	C1	C2	C3	C4	C5	C6	C7	C8	C9	C10	C11
JPlag-java and AllnoNOR	1	0	0	1	1	0	0	0	0	0	0
JPlag-java and AllNOR	1	0	0	1	1	0	0	0	0	0	0
JPlag-java and CCR	1	0	0	1	1	0	0	0	0	0	0
JPlag-java and NoPPT	1	0	0	1	1	0	0	0	0	0	0
JPlag-java and TE	1	0	0	1	1	0	0	0	0	0	0
JPlag-text and AllnoNOR	-1	1	1	0	0	0	0	0	0	1	1
JPlag-text and AllNOR	-1	1	1	0	0	0	0	0	0	1	1
JPlag-text and CCR	-1	1	1	0	0	0	0	0	0	1	-1
JPlag-text and NoPPT	-1	1	1	0	0	0	0	0	0	-4	0
JPlag-text and TE	-1	1	1	0	0	0	0	0	0	1	-1
Sherlock-java and AllnoNOR	1	0	0	-2	0	0	0	0	0	0	0
Sherlock-java and AllNOR	1	0	0	-2	0	0	0	0	0	0	0
Sherlock-java and CCR	1	0	0	-2	0	0	0	0	0	0	0
Sherlock-java and NoPPT	1	0	0	-2	0	0	0	0	0	0	0
Sherlock-java and TE	1	0	0	-2	0	0	0	0	0	0	0
Sherlock-text and AllnoNOR	-1	-2	0	0	0	1	1	0	-1	0	0
Sherlock-text and AllNOR	-1	-2	0	0	0	1	1	0	1	0	0
Sherlock-text and CCR	-1	-2	0	0	0	1	-1	-1	0	0	0
Sherlock-text and NoPPT	-1	-2	0	0	0	-4	0	0	0	0	0
Sherlock-text and TE	-1	-2	0	0	0	1	-1	1	0	0	0
SIM-java and AllnoNOR	1	0	0	1	-1	0	0	0	0	0	0
SIM-java and AllNOR	1	0	0	1	-1	0	0	0	0	0	0
SIM-java and CCR	1	0	0	1	-1	0	0	0	0	0	0
SIM-java and NoPPT	1	0	0	1	-1	0	0	0	0	0	0
SIM-java and TE	1	0	0	1	-1	0	0	0	0	0	0
SIM-text and AllnoNOR	-1	1	-1	0	0	0	0	0	0	0	0
SIM-text and AllNOR	-1	1	-1	0	0	0	0	0	0	0	0
SIM-text and CCR	-1	1	-1	0	0	0	0	0	0	0	0
SIM-text and NoPPT	-1	1	-1	0	0	0	0	0	0	0	0
SIM-text and TE	-1	1	-1	0	0	0	0	0	0	0	0

Table F.5: Simple effects analysis contrast codings - student dataset - part2

	C12	C13	C14	C15	C16	C17	C18	C19	C20
JPlag-java and AllnoNOR	0	0	0	0	0	0	0	0	0
JPlag-java and AllNOR	0	0	0	0	0	0	0	0	0
JPlag-java and CCR	0	0	0	0	0	0	0	0	0
JPlag-java and NoPPT	0	0	0	0	0	0	0	0	0
JPlag-java and TE	0	0	0	0	0	0	0	0	0
JPlag-text and AllnoNOR	0	-1	0	0	0	0	0	0	0
JPlag-text and AllNOR	0	1	0	0	0	0	0	0	0
JPlag-text and CCR	-1	0	0	0	0	0	0	0	0
JPlag-text and NoPPT	0	0	0	0	0	0	0	0	0
JPlag-text and TE	1	0	0	0	0	0	0	0	0
Sherlock-java and AllnoNOR	0	0	0	0	0	0	1	1	0
Sherlock-java and AllNOR	0	0	0	0	0	0	1	1	0
Sherlock-java and CCR	0	0	0	0	0	0	1	-1	-1
Sherlock-java and NoPPT	0	0	0	0	0	0	-4	0	0
Sherlock-java and TE	0	0	0	0	0	0	1	-1	1
Sherlock-text and AllnoNOR	0	0	0	0	0	0	0	0	0
Sherlock-text and AllNOR	0	0	0	0	0	0	0	0	0
Sherlock-text and CCR	0	0	0	0	0	0	0	0	0
Sherlock-text and NoPPT	0	0	0	0	0	0	0	0	0
Sherlock-text and TE	0	0	0	0	0	0	0	0	0
SIM-java and AllnoNOR	0	0	0	0	0	0	0	0	0
SIM-java and AllNOR	0	0	0	0	0	0	0	0	0
SIM-java and CCR	0	0	0	0	0	0	0	0	0
SIM-java and NoPPT	0	0	0	0	0	0	0	0	0
SIM-java and TE	0	0	0	0	0	0	0	0	0
SIM-text and AllnoNOR	0	0	1	1	0	-1	0	0	0
SIM-text and AllNOR	0	0	1	1	0	1	0	0	0
SIM-text and CCR	0	0	1	-1	-1	0	0	0	0
SIM-text and NoPPT	0	0	-4	0	0	0	0	0	0
SIM-text and TE	0	0	1	-1	1	0	0	0	0

Table F.6: Simple effects analysis contrast codings - student dataset - part3

	C21	C22	C23	C24	C25	C26	C27	C28	C29
JPlag-java and AllnoNOR	0	1	1	0	-1	0	0	0	0
JPlag-java and AllNOR	0	1	1	0	1	0	0	0	0
JPlag-java and CCR	0	1	-1	-1	0	0	0	0	0
JPlag-java and NoPPT	0	-4	0	0	0	0	0	0	0
JPlag-java and TE	0	1	-1	1	0	0	0	0	0
JPlag-text and AllnoNOR	0	0	0	0	0	0	0	0	0
JPlag-text and AllNOR	0	0	0	0	0	0	0	0	0
JPlag-text and CCR	0	0	0	0	0	0	0	0	0
JPlag-text and NoPPT	0	0	0	0	0	0	0	0	0
JPlag-text and TE	0	0	0	0	0	0	0	0	0
Sherlock-java and AllnoNOR	-1	0	0	0	0	0	0	0	0
Sherlock-java and AllNOR	1	0	0	0	0	0	0	0	0
Sherlock-java and CCR	0	0	0	0	0	0	0	0	0
Sherlock-java and NoPPT	0	0	0	0	0	0	0	0	0
Sherlock-java and TE	0	0	0	0	0	0	0	0	0
Sherlock-text and AllnoNOR	0	0	0	0	0	0	0	0	0
Sherlock-text and AllNOR	0	0	0	0	0	0	0	0	0
Sherlock-text and CCR	0	0	0	0	0	0	0	0	0
Sherlock-text and NoPPT	0	0	0	0	0	0	0	0	0
Sherlock-text and TE	0	0	0	0	0	0	0	0	0
SIM-java and AllnoNOR	0	0	0	0	0	1	1	0	-1
SIM-java and AllNOR	0	0	0	0	0	1	1	0	1
SIM-java and CCR	0	0	0	0	0	1	-1	-1	0
SIM-java and NoPPT	0	0	0	0	0	-4	0	0	0
SIM-java and TE	0	0	0	0	0	1	-1	1	0
SIM-text and AllnoNOR	0	0	0	0	0	0	0	0	0
SIM-text and AllNOR	0	0	0	0	0	0	0	0	0
SIM-text and CCR	0	0	0	0	0	0	0	0	0
SIM-text and NoPPT	0	0	0	0	0	0	0	0	0
SIM-text and TE	0	0	0	0	0	0	0	0	0

CHAPTER G

SHAPIRO-WILK NORMALITY TEST

Results of Shapiro-Wilk normality test for SOCO dataset D1 assigmenets group:

```
## $`Normal distribution`
##  [1] "JPlag-java - NoPPT with W=0.9646739, p=0.3856066;"
##  [2] "JPlag-java - RC with W=0.9646739, p=0.3856066;"
##  [3] "JPlag-java - CCR with W=0.981933, p=0.8640068;"
##  [4] "JPlag-java - AllnoNOR with W=0.9736222, p=0.6233963;"
##  [5] "JPlag-java - AllNOR with W=0.9736222, p=0.6233963;"
##  [6] "SIM-java - NoPPT with W=0.9684525, p=0.4775116;"
##  [7] "SIM-java - RC with W=0.9703781, p=0.5295313;"
##  [8] "SIM-java - CCR with W=0.9645588, p=0.3830315;"
##  [9] "SIM-java - AllnoNOR with W=0.963926, p=0.3691145;"
## [10] "SIM-java - AllNOR with W=0.963926, p=0.3691145;"
## [11] "Sherlock-java - NoPPT with W=0.9622875, p=0.3349699;"
## [12] "Sherlock-java - RC with W=0.9397757, p=0.08134755;"
## [13] "Sherlock-java - CCR with W=0.9488192, p=0.1448137;"
## [14] "Sherlock-java - AllnoNOR with W=0.9316043, p=0.04848914;"
## [15] "Sherlock-java - AllNOR with W=0.9490796, p=0.1472336;"
## [16] "JPlag-text - NoPPT with W=0.968684, p=0.4835915;"
## [17] "JPlag-text - RC with W=0.9604874, p=0.3005428;"
## [18] "JPlag-text - CCR with W=0.9628915, p=0.3472413;"
## [19] "JPlag-text - AllnoNOR with W=0.9712717, p=0.554704;"
## [20] "JPlag-text - AllNOR with W=0.9712717, p=0.554704;"
## [21] "SIM-text - NoPPT with W=0.9618008, p=0.3253471;"
## [22] "SIM-text - RC with W=0.9781271, p=0.7587939;"
## [23] "Sherlock-text - AllNOR with W=0.9072469, p=0.01098593;"
## $`Non-normal distribution`
## [1] "SIM-text - CCR with W=0.8525712, p=0.0005750178;"
## [2] "SIM-text - AllnoNOR with W=0.790909, p=3.519414e-05;"
## [3] "SIM-text - AllNOR with W=0.790909, p=3.519414e-05;"
## [4] "Sherlock-text - NoPPT with W=0.6764819, p=5.32229e-07;"
## [5] "Sherlock-text - RC with W=0.8960805, p=0.005762239;"
## [6] "Sherlock-text - CCR with W=0.8812919, p=0.002538942;"
## [7] "Sherlock-text - AllnoNOR with W=0.8855633, p=0.003204142;"
```

Results of Shapiro-Wilk normality test for SOCO dataset D2 assigmenets group:

```
## $`Normal distribution`
##  [1] "JPlag-java - NoPPT with W=0.9858052, p=0.9450095;"
##  [2] "JPlag-java - RC with W=0.9858052, p=0.9450095;"
##  [3] "JPlag-java - CCR with W=0.975526, p=0.6806886;"
##  [4] "JPlag-java - AllnoNOR with W=0.9715212, p=0.5618368;"
##  [5] "JPlag-java - AllNOR with W=0.9723407, p=0.585556;"
##  [6] "SIM-java - CCR with W=0.9670385, p=0.4414707;"
##  [7] "SIM-java - AllnoNOR with W=0.9593151, p=0.2798053;"
##  [8] "SIM-java - AllNOR with W=0.9593151, p=0.2798053;"
##  [9] "Sherlock-java - NoPPT with W=0.9795755, p=0.8007722;"
## [10] "Sherlock-java - RC with W=0.9618583, p=0.3264707;"
## [11] "Sherlock-java - AllnoNOR with W=0.9390745, p=0.07779658;"
## [12] "JPlag-text - NoPPT with W=0.9466654, p=0.1262406;"
## [13] "JPlag-text - RC with W=0.9317417, p=0.04890997;"
## [14] "JPlag-text - CCR with W=0.9674398, p=0.4515039;"
## [15] "JPlag-text - AllnoNOR with W=0.9490834, p=0.1472693;"
## [16] "JPlag-text - AllNOR with W=0.9490834, p=0.1472693;"
## [17] "SIM-text - NoPPT with W=0.9716205, p=0.5646882;"
## [18] "SIM-text - RC with W=0.9451164, p=0.11436;"
## [19] "SIM-text - AllnoNOR with W=0.9309576, p=0.04655822;"
## [20] "SIM-text - AllNOR with W=0.9309576, p=0.04655822;"
## [21] "Sherlock-text - NoPPT with W=0.9246174, p=0.03135723;"
## [22] "Sherlock-text - CCR with W=0.9184819, p=0.02151919;"
## [23] "Sherlock-text - AllNOR with W=0.9433229, p=0.1019913;"
## $`Non-normal distribution`
## [1] "SIM-java - NoPPT with W=0.8862555, p=0.003328287;"
## [2] "SIM-java - RC with W=0.8862555, p=0.003328287;"
## [3] "Sherlock-java - CCR with W=0.8846273, p=0.003044016;"
## [4] "Sherlock-java - AllNOR with W=0.8907427, p=0.004267217;"
## [5] "SIM-text - CCR with W=0.9052535, p=0.00977406;"
## [6] "Sherlock-text - RC with W=0.8180266, p=0.0001133849;"
## [7] "Sherlock-text - AllnoNOR with W=0.8700778, p=0.001399096;"
```

Results of Shapiro-Wilk normality test for SOCO dataset D3 assigmenets group:

```
## $`Normal distribution`
##  [1] "JPlag-java - NoPPT with W=0.9576201, p=0.26895;"
##  [2] "JPlag-java - RC with W=0.9576201, p=0.26895;"
##  [3] "SIM-java - NoPPT with W=0.9330666, p=0.05926352;"
##  [4] "SIM-java - RC with W=0.9330666, p=0.05926352;"
##  [5] "SIM-java - CCR with W=0.9770035, p=0.7415289;"
##  [6] "SIM-java - AllnoNOR with W=0.9770035, p=0.7415289;"
##  [7] "SIM-java - AllNOR with W=0.9770035, p=0.7415289;"
##  [8] "Sherlock-java - NoPPT with W=0.9696577, p=0.5298511;"
##  [9] "Sherlock-java - RC with W=0.9315619, p=0.05403212;"
## [10] "Sherlock-java - CCR with W=0.9582691, p=0.2795822;"
## [11] "Sherlock-java - AllnoNOR with W=0.9436364, p=0.1139858;"
## [12] "JPlag-text - AllnoNOR with W=0.9559115, p=0.2426764;"
## [13] "JPlag-text - AllNOR with W=0.9559115, p=0.2426764;"
## [14] "SIM-text - RC with W=0.962473, p=0.357731;"
## $`Non-normal distribution`
##  [1] "JPlag-java - CCR with W=0.87775, p=0.002504022;"
##  [2] "JPlag-java - AllnoNOR with W=0.8684517, p=0.001547752;"
##  [3] "JPlag-java - AllNOR with W=0.9003744, p=0.008577694;"
##  [4] "Sherlock-java - AllNOR with W=0.8555798, p=0.0008132337;"
##  [5] "JPlag-text - NoPPT with W=0.8211926, p=0.0001636066;"
##  [6] "JPlag-text - RC with W=0.7608029, p=1.367942e-05;"
##  [7] "JPlag-text - CCR with W=0.8946996, p=0.006246444;"
##  [8] "SIM-text - NoPPT with W=0.8954656, p=0.00651751;"
##  [9] "SIM-text - CCR with W=0.8133351, p=0.0001158564;"
## [10] "SIM-text - AllnoNOR with W=0.5541259, p=2.101866e-08;"
## [11] "SIM-text - AllNOR with W=0.5541259, p=2.101866e-08;"
## [12] "Sherlock-text - NoPPT with W=0.8955069, p=0.006532471;"
## [13] "Sherlock-text - RC with W=0.772367, p=2.138859e-05;"
## [14] "Sherlock-text - CCR with W=0.8612356, p=0.00107563;"
## [15] "Sherlock-text - AllnoNOR with W=0.8601214, p=0.001017588;"
## [16] "Sherlock-text - AllNOR with W=0.6531109, p=3.470444e-07;"
```

Results of Shapiro-Wilk normality test for SOCO dataset D4 assigmenets group:

```
## $`Normal distribution`
##  [1] "JPlag-java - NoPPT with W=0.9479069, p=0.1366391;"
##  [2] "JPlag-java - RC with W=0.9479069, p=0.1366391;"
##  [3] "JPlag-java - CCR with W=0.9306143, p=0.04556583;"
##  [4] "JPlag-java - AllnoNOR with W=0.9281943, p=0.0391624;"
##  [5] "JPlag-java - AllNOR with W=0.9505117, p=0.161267;"
##  [6] "SIM-java - NoPPT with W=0.9559183, p=0.2267597;"
##  [7] "SIM-java - RC with W=0.9559183, p=0.2267597;"
##  [8] "SIM-java - CCR with W=0.9502513, p=0.1586223;"
##  [9] "SIM-java - AllnoNOR with W=0.948355, p=0.1405961;"
## [10] "SIM-java - AllNOR with W=0.9469539, p=0.1285843;"
## [11] "Sherlock-java - RC with W=0.9592104, p=0.2780161;"
## [12] "Sherlock-java - CCR with W=0.951189, p=0.1683487;"
## [13] "Sherlock-java - AllnoNOR with W=0.9285138, p=0.03995139;"
## [14] "JPlag-text - NoPPT with W=0.9103471, p=0.01319518;"
## [15] "JPlag-text - CCR with W=0.9519791, p=0.1769896;"
## [16] "JPlag-text - AllnoNOR with W=0.9329913, p=0.05291376;"
## [17] "JPlag-text - AllNOR with W=0.9329913, p=0.05291376;"
## [18] "SIM-text - NoPPT with W=0.9219974, p=0.0266797;"
## [19] "SIM-text - RC with W=0.9317174, p=0.04883506;"
## [20] "Sherlock-text - AllNOR with W=0.9596934, p=0.2863554;"
## $`Non-normal distribution`
##  [1] "Sherlock-java - NoPPT with W=0.8936639, p=0.005026356;"
##  [2] "Sherlock-java - AllNOR with W=0.8894397, p=0.003968659;"
##  [3] "JPlag-text - RC with W=0.8585491, p=0.0007748055;"
##  [4] "SIM-text - CCR with W=0.374969, p=2.135516e-10;"
##  [5] "SIM-text - AllnoNOR with W=0.3794252, p=2.347685e-10;"
##  [6] "SIM-text - AllNOR with W=0.3794252, p=2.347685e-10;"
##  [7] "Sherlock-text - NoPPT with W=0.8784289, p=0.002176162;"
##  [8] "Sherlock-text - RC with W=0.8830677, p=0.00279574;"
##  [9] "Sherlock-text - CCR with W=0.7596318, p=1.004897e-05;"
## [10] "Sherlock-text - AllnoNOR with W=0.4672953, p=1.683508e-09;"
```

Results of Shapiro-Wilk normality test for RSS dataset A1 assigmenet group:

```
## $`Normal distribution`
##  [1] "JPlag-java - NoPPT with W=0.7095627, p=0.01491974;"
##  [2] "JPlag-java - CCR with W=0.8495198, p=0.224609;"
##  [3] "JPlag-java - TE with W=0.960889, p=0.7845021;"
##  [4] "JPlag-java - AllnoNOR with W=0.9318662, p=0.6054105;"
##  [5] "JPlag-java - AllNOR with W=0.9318662, p=0.6054105;"
##  [6] "SIM-java - NoPPT with W=0.7612247, p=0.0489199;"
##  [7] "SIM-java - CCR with W=0.7612247, p=0.0489199;"
##  [8] "SIM-java - TE with W=0.7877798, p=0.08205024;"
##  [9] "SIM-java - AllnoNOR with W=0.700802, p=0.0118625;"
## [10] "SIM-java - AllNOR with W=0.700802, p=0.0118625;"
## [11] "Sherlock-java - NoPPT with W=0.8150224, p=0.1320046;"
## [12] "Sherlock-java - CCR with W=0.8584345, p=0.2546618;"
## [13] "Sherlock-java - TE with W=0.8659565, p=0.2821167;"
## [14] "Sherlock-java - AllnoNOR with W=0.8812872, p=0.3441629;"
## [15] "Sherlock-java - AllNOR with W=0.8837843, p=0.3550508;"
## [16] "JPlag-text - NoPPT with W=0.8507872, p=0.2287199;"
## [17] "JPlag-text - CCR with W=0.8411582, p=0.1988004;"
## [18] "JPlag-text - TE with W=0.8470807, p=0.216846;"
## [19] "JPlag-text - AllnoNOR with W=0.851944, p=0.2325183;"
## [20] "JPlag-text - AllNOR with W=0.851944, p=0.2325183;"
## [21] "SIM-text - NoPPT with W=0.8041968, p=0.1099617;"
## [22] "SIM-text - TE with W=0.8107794, p=0.1229993;"
## [23] "SIM-text - AllnoNOR with W=0.9064628, p=0.4638634;"
## [24] "SIM-text - AllNOR with W=0.9064628, p=0.4638634;"
## [25] "Sherlock-text - NoPPT with W=0.803279, p=0.1082314;"
## [26] "Sherlock-text - CCR with W=0.8640498, p=0.2749732;"
## [27] "Sherlock-text - AllnoNOR with W=0.9359038, p=0.6295144;"
## [28] "Sherlock-text - AllNOR with W=0.9567416, p=0.7584044;"
## $`Non-normal distribution`
## [1] "SIM-text - CCR with W=0.6297763, p=0.001240726;"
## [2] "Sherlock-text - TE with W=0.6726856, p=0.005322119;"
```

All tests were done using Shapiro-Wilk normality test at $p<0.01$. The test was performed using *shapiro.test* function from the package *stats*.

CHAPTER H

MODEL COMPARISONS

This Appendix presents the results of the model comparisons used in the statistical analysis for the different assignments. All models are using an random intercept. The term *Participant* in the SOCO dataset represents one subset assignment (e.g., Dn-1, Dn-2, Dn-3, etc.) from the D1, D2, D3 or D4 group of assignments, and in the student dataset it represents the results from one academic year for the A1 assignment. The models that were compared are:

- NullModel

$$lmer(F1 \sim (1|Participant) + (1|Tool:Participant) + (1|Technique:Participant), \\ data = SOCO.D_n, REML = FALSE) \tag{H.1}$$

- ToolModel:

$$lmer(F1 \sim Tool + \\ (1|Participant) + (1|Tool:Participant) + (1|Technique:Participant), \\ data = SOCO.D_n, REML = FALSE) \tag{H.2}$$

- MainEffectsModel:

$$lmer(F1 \sim Tool + Technique + \\ (1|Participant) + (1|Tool:Participant) + (1|Technique:Participant), \\ data = SOCO.D_n, REML = FALSE) \tag{H.3}$$

- InteractionModel or FullModel:

$$lmer(F1 \sim Tool + Technique + Tool:Technique + \\ (1|Participant) + (1|Tool:Participant) + (1|Technique:Participant), \\ data = SOCO.D_n, REML = FALSE) \tag{H.4}$$

Table H.1: MLM comparison for SOCO D1

	Df	AIC	BIC	logLik	deviance	Chisq	Chi Df	Pr(>Chisq)	p.boot
NullModel	5	-1421.5	-1397.3	715.7	-1431.5	NA	NA	NA	NA
ToolModel	10	-1988.1	-1939.7	1004.0	-2008.1	576.6	5	0.0000	0.0001
MainEffectsModel	14	-2033.9	-1966.2	1030.9	-2061.9	53.8	4	0.0000	0.0001
InteractionModel	34	-3361.6	-3197.2	1714.8	-3429.6	1367.8	20	0.0000	0.0001

Table H.2: MLM comparison for SOCO D2

	Df	AIC	BIC	logLik	deviance	Chisq	Chi Df	Pr(>Chisq)	p.boot
NullModel	5	-1564.6	-1540.4	787.3	-1574.6	NA	NA	NA	NA
ToolModel	10	-2086.6	-2038.2	1053.3	-2106.6	531.9	5	0.0000	0.0001
MainEffectsModel	14	-2119.5	-2051.8	1073.8	-2147.5	41.0	4	0.0000	0.0001
InteractionModel	34	-3351.9	-3187.5	1709.9	-3419.9	1272.4	20	0.0000	0.0001

Table H.3: MLM comparison for SOCO D3

	Df	AIC	BIC	logLik	deviance	Chisq	Chi Df	Pr(>Chisq)	p.boot
NullModel	5	-1751.4	-1727.4	880.7	-1761.4	NA	NA	NA	NA
ToolModel	10	-2431.1	-2383.1	1225.6	-2451.1	689.7	5	0.0000	0.0001
MainEffectsModel	14	-2445.6	-2378.4	1236.8	-2473.6	22.5	4	0.0002	0.0003
InteractionModel	34	-3365.1	-3201.8	1716.5	-3433.1	959.5	20	0.0000	0.0001

Table H.4: MLM comparison for SOCO D4

	Df	AIC	BIC	logLik	deviance	Chisq	Chi Df	Pr(>Chisq)	p.boot
NullModel	5	-1338.8	-1314.7	674.4	-1348.8	NA	NA	NA	NA
ToolModel	10	-1793.7	-1745.3	906.8	-1813.7	464.8	5	0.0000	0.0001
MainEffectsModel	14	-1806.6	-1738.9	917.3	-1834.6	20.9	4	0.0003	0.0006
InteractionModel	34	-1866.1	-1701.7	967.0	-1934.1	99.5	20	0.0000	0.0001

Table H.5: Multi level linear model (MLM) comparison for RSS A1

	Df	AIC	BIC	logLik	deviance	Chisq	Chi Df	Pr(>Chisq)	p.boot
NullModel	5	-68.2	-54.3	39.1	-78.2	NA	NA	NA	NA
ToolModel	10	-72.8	-45.0	46.4	-92.8	14.6	5	0.0121	0.0351
MainEffectsModel	14	-75.9	-36.9	51.9	-103.9	11.0	4	0.0261	0.0637
InteractionModel	34	-73.8	21.0	70.9	-141.8	37.9	20	0.0092	0.0394

CHAPTER I

CONSTRAST EFFECT SIZES

Table I.1: Contrasts effect sizes for SOCO D1

ContrastName	EffectSize	CI.LB	CI.UB
(Intercept)	1.00	NA	NA
Tool.TextvsJava	0.89	0.88	0.91
TT.SHvsOthers	0.86	0.84	0.89
TJ.SHvsOthers	0.94	0.94	0.96
TT.SIMvsJPlag	0.95	0.95	0.96
TJ.SIMvsJPlag	0.80	0.78	0.84
NoPPTvsPPT	0.62	0.26	0.69
SinglevsCombo	0.05	0.00	0.19
RCvsCCR	0.70	0.33	0.75
AnoNvsAN	0.72	0.35	0.77
Tool.TextvsJava:NoPPTvsPPT	0.21	0.13	0.28
TT.SHvsOthers:NoPPTvsPPT	0.71	0.64	0.74
TJ.SHvsOthers:NoPPTvsPPT	0.10	0.02	0.18
TT.SIMvsJPlag:NoPPTvsPPT	0.34	0.26	0.41
TJ.SIMvsJPlag:NoPPTvsPPT	0.00	0.00	0.09
Tool.TextvsJava:SinglevsCombo	0.16	0.08	0.23
TT.SHvsOthers:SinglevsCombo	0.82	0.77	0.84
TJ.SHvsOthers:SinglevsCombo	0.15	0.07	0.22
TT.SIMvsJPlag:SinglevsCombo	0.47	0.38	0.52
TJ.SIMvsJPlag:SinglevsCombo	0.01	0.00	0.09
Tool.TextvsJava:RCvsCCR	0.59	0.52	0.64
TT.SHvsOthers:RCvsCCR	0.61	0.53	0.65
TJ.SHvsOthers:RCvsCCR	0.32	0.24	0.38
TT.SIMvsJPlag:RCvsCCR	0.19	0.11	0.26
TJ.SIMvsJPlag:RCvsCCR	0.00	0.00	0.09
Tool.TextvsJava:AnoNvsAN	0.51	0.43	0.56
TT.SHvsOthers:AnoNvsAN	0.73	0.67	0.77
TJ.SHvsOthers:AnoNvsAN	0.11	0.03	0.19
TT.SIMvsJPlag:AnoNvsAN	0.00	0.00	0.09
TJ.SIMvsJPlag:AnoNvsAN	0.00	0.00	0.09

Note:
TT - ToolText, TJ - ToolJava, SH - Sherlock, AnoN - AllnoNOR, AN - AllNOR

Table I.2: Simple effect analysis effect sizes for SOCO D1

ContrastName	EffectSize	CI.LB	CI.UB
(Intercept)	1.00	NA	NA
TextvsJava	0.89	0.88	0.91
TT.SHvsOthers	0.86	0.84	0.89
TT.SIMvsJPlag	0.95	0.95	0.96
TJ.SHvsOthers	0.94	0.94	0.96
TJ.SIMvsJPlag	0.80	0.78	0.84
TT.SH.NoPPTvsPPT	0.47	0.41	0.53
TT.SH.SinglevsCombo	0.74	0.71	0.78
TT.SH.RCvsCCR	0.13	0.06	0.20
TT.SH.AllnoNORvsAllNOR	0.77	0.74	0.80
TT.JPlag.NoPPTvsPPT	0.34	0.27	0.40
TT.JPlag.SinglevsCombo	0.13	0.06	0.21
TT.JPlag.RCvsCCR	0.51	0.46	0.57
TT.JPlag.AllnoNORvsAllNOR	0.00	0.00	0.08
TT.SIM.NoPPTvsPPT	0.64	0.59	0.68
TT.SIM.SinglevsCombo	0.63	0.59	0.68
TT.SIM.RCvsCCR	0.65	0.61	0.69
TT.SIM.AllnoNORvsAllNOR	0.00	0.00	0.08
TJ.SH.NoPPTvsPPT	0.13	0.06	0.20
TJ.SH.SinglevsCombo	0.18	0.11	0.25
TJ.SH.RCvsCCR	0.34	0.27	0.40
TJ.SH.AllnoNORvsAllNOR	0.13	0.06	0.20
TJ.JPlag.NoPPTvsPPT	0.02	0.00	0.09
TJ.JPlag.SinglevsCombo	0.02	0.00	0.09
TJ.JPlag.RCvsCCR	0.02	0.00	0.09
TJ.JPlag.AllnoNORvsAllNOR	0.00	0.00	0.08
TJ.SIM.NoPPTvsPPT	0.01	0.00	0.09
TJ.SIM.SinglevsCombo	0.01	0.00	0.09
TJ.SIM.RCvsCCR	0.02	0.00	0.09
TJ.SIM.AllnoNORvsAllNOR	0.00	0.00	0.09

Note:
TT - ToolText, TJ - ToolJava, SH - Sherlock

Table I.3: Contrasts effect sizes for SOCO D2

ContrastName	EffectSize	CI.LB	CI.UB
(Intercept)	1.00	NA	NA
Tool.TextvsJava	0.82	0.82	0.86
TT.SHvsOthers	0.80	0.80	0.84
TJ.SHvsOthers	0.93	0.93	0.94
TT.SIMvsJPlag	0.95	0.95	0.97
TJ.SIMvsJPlag	0.78	0.77	0.82
NoPPTvsPPT	0.14	0.08	0.43
SinglevsCombo	0.09	0.02	0.33
RCvsCCR	0.36	0.31	0.73
AnoNvsAN	0.35	0.29	0.72
Tool.TextvsJava:NoPPTvsPPT	0.27	0.21	0.36
TT.SHvsOthers:NoPPTvsPPT	0.68	0.65	0.75
TJ.SHvsOthers:NoPPTvsPPT	0.09	0.02	0.17
TT.SIMvsJPlag:NoPPTvsPPT	0.17	0.11	0.26
TJ.SIMvsJPlag:NoPPTvsPPT	0.05	0.00	0.13
Tool.TextvsJava:SinglevsCombo	0.01	0.00	0.09
TT.SHvsOthers:SinglevsCombo	0.77	0.75	0.83
TJ.SHvsOthers:SinglevsCombo	0.09	0.02	0.17
TT.SIMvsJPlag:SinglevsCombo	0.28	0.22	0.36
TJ.SIMvsJPlag:SinglevsCombo	0.03	0.00	0.11
Tool.TextvsJava:RCvsCCR	0.58	0.54	0.66
TT.SHvsOthers:RCvsCCR	0.61	0.57	0.69
TJ.SHvsOthers:RCvsCCR	0.10	0.03	0.18
TT.SIMvsJPlag:RCvsCCR	0.13	0.06	0.22
TJ.SIMvsJPlag:RCvsCCR	0.05	0.00	0.13
Tool.TextvsJava:AnoNvsAN	0.39	0.33	0.48
TT.SHvsOthers:AnoNvsAN	0.62	0.58	0.69
TJ.SHvsOthers:AnoNvsAN	0.05	0.00	0.13
TT.SIMvsJPlag:AnoNvsAN	0.00	0.00	0.09
TJ.SIMvsJPlag:AnoNvsAN	0.00	0.00	0.09

Note:
TT - ToolText, TJ - ToolJava, SH - Sherlock, AnoN - AllnoNOR, AN - AllNOR

Table I.4: Simple effect analysis effect sizes for SOCO D2

ContrastName	EffectSize	CI.LB	CI.UB
(Intercept)	1.00	NA	NA
TextvsJava	0.82	0.82	0.86
TT.SHvsOthers	0.80	0.80	0.84
TT.SIMvsJPlag	0.95	0.95	0.97
TJ.SHvsOthers	0.93	0.93	0.94
TJ.SIMvsJPlag	0.78	0.77	0.82
TT.SH.NoPPTvsPPT	0.51	0.45	0.56
TT.SH.SinglevsCombo	0.72	0.69	0.75
TT.SH.RCvsCCR	0.18	0.11	0.25
TT.SH.AllnoNORvsAllNOR	0.69	0.66	0.73
TT.JPlag.NoPPTvsPPT	0.39	0.33	0.45
TT.JPlag.SinglevsCombo	0.25	0.18	0.32
TT.JPlag.RCvsCCR	0.56	0.51	0.61
TT.JPlag.AllnoNORvsAllNOR	0.00	0.00	0.08
TT.SIM.NoPPTvsPPT	0.56	0.51	0.61
TT.SIM.SinglevsCombo	0.55	0.50	0.60
TT.SIM.RCvsCCR	0.65	0.61	0.69
TT.SIM.AllnoNORvsAllNOR	0.00	0.00	0.08
TJ.SH.NoPPTvsPPT	0.02	0.00	0.09
TJ.SH.SinglevsCombo	0.04	0.00	0.11
TJ.SH.RCvsCCR	0.21	0.14	0.28
TJ.SH.AllnoNORvsAllNOR	0.06	0.00	0.13
TJ.JPlag.NoPPTvsPPT	0.12	0.05	0.20
TJ.JPlag.SinglevsCombo	0.09	0.02	0.16
TJ.JPlag.RCvsCCR	0.13	0.06	0.20
TJ.JPlag.AllnoNORvsAllNOR	0.00	0.00	0.08
TJ.SIM.NoPPTvsPPT	0.06	0.00	0.13
TJ.SIM.SinglevsCombo	0.05	0.00	0.12
TJ.SIM.RCvsCCR	0.06	0.00	0.13
TJ.SIM.AllnoNORvsAllNOR	0.00	0.00	0.08

Note:
TT - ToolText, TJ - ToolJava, SH - Sherlock

Table I.5: Contrasts effect sizes for SOCO D3

ContrastName	EffectSize	CI.LB	CI.UB
(Intercept)	1.00	NA	NA
Tool.TextvsJava	0.94	0.94	0.96
TT.SHvsOthers	0.91	0.92	0.95
TJ.SHvsOthers	0.96	0.96	0.98
TT.SIMvsJPlag	0.98	0.98	0.99
TJ.SIMvsJPlag	0.64	0.64	0.74
NoPPTvsPPT	0.49	0.19	0.60
SinglevsCombo	0.09	0.00	0.24
RCvsCCR	0.19	0.02	0.33
AnoNvsAN	0.44	0.16	0.56
Tool.TextvsJava:NoPPTvsPPT	0.15	0.06	0.22
TT.SHvsOthers:NoPPTvsPPT	0.47	0.37	0.53
TJ.SHvsOthers:NoPPTvsPPT	0.50	0.40	0.56
TT.SIMvsJPlag:NoPPTvsPPT	0.10	0.02	0.17
TJ.SIMvsJPlag:NoPPTvsPPT	0.13	0.05	0.20
Tool.TextvsJava:SinglevsCombo	0.20	0.11	0.27
TT.SHvsOthers:SinglevsCombo	0.66	0.57	0.71
TJ.SHvsOthers:SinglevsCombo	0.50	0.41	0.56
TT.SIMvsJPlag:SinglevsCombo	0.00	0.00	0.09
TJ.SIMvsJPlag:SinglevsCombo	0.10	0.02	0.18
Tool.TextvsJava:RCvsCCR	0.62	0.53	0.68
TT.SHvsOthers:RCvsCCR	0.25	0.16	0.32
TJ.SHvsOthers:RCvsCCR	0.22	0.14	0.30
TT.SIMvsJPlag:RCvsCCR	0.13	0.04	0.20
TJ.SIMvsJPlag:RCvsCCR	0.14	0.05	0.21
Tool.TextvsJava:AnoNvsAN	0.45	0.36	0.51
TT.SHvsOthers:AnoNvsAN	0.60	0.51	0.66
TJ.SHvsOthers:AnoNvsAN	0.25	0.17	0.33
TT.SIMvsJPlag:AnoNvsAN	0.00	0.00	0.09
TJ.SIMvsJPlag:AnoNvsAN	0.00	0.00	0.09

Note:
TT - ToolText, TJ - ToolJava, SH - Sherlock, AnoN - AllnoNOR, AN - AllNOR

Table I.6: Simple effect analysis effect sizes for SOCO D3

ContrastName	EffectSize	CI.LB	CI.UB
(Intercept)	1.00	NA	NA
TextvsJava	0.94	0.94	0.96
TT.SHvsOthers	0.91	0.92	0.95
TT.SIMvsJPlag	0.98	0.98	0.99
TJ.SHvsOthers	0.96	0.96	0.98
TJ.SIMvsJPlag	0.64	0.64	0.74
TT.SH.NoPPTvsPPT	0.22	0.15	0.30
TT.SH.SinglevsCombo	0.57	0.52	0.64
TT.SH.RCvsCCR	0.13	0.06	0.21
TT.SH.AllnoNORvsAllNOR	0.63	0.58	0.70
TT.JPlag.NoPPTvsPPT	0.38	0.31	0.46
TT.JPlag.SinglevsCombo	0.26	0.19	0.33
TT.JPlag.RCvsCCR	0.45	0.39	0.52
TT.JPlag.AllnoNORvsAllNOR	0.00	0.00	0.09
TT.SIM.NoPPTvsPPT	0.28	0.21	0.36
TT.SIM.SinglevsCombo	0.25	0.18	0.33
TT.SIM.RCvsCCR	0.32	0.25	0.40
TT.SIM.AllnoNORvsAllNOR	0.00	0.00	0.09
TJ.SH.NoPPTvsPPT	0.42	0.36	0.50
TJ.SH.SinglevsCombo	0.46	0.39	0.53
TJ.SH.RCvsCCR	0.39	0.32	0.46
TJ.SH.AllnoNORvsAllNOR	0.28	0.20	0.35
TJ.JPlag.NoPPTvsPPT	0.24	0.17	0.31
TJ.JPlag.SinglevsCombo	0.18	0.11	0.26
TJ.JPlag.RCvsCCR	0.25	0.18	0.33
TJ.JPlag.AllnoNORvsAllNOR	0.00	0.00	0.09
TJ.SIM.NoPPTvsPPT	0.08	0.01	0.15
TJ.SIM.SinglevsCombo	0.06	0.00	0.13
TJ.SIM.RCvsCCR	0.08	0.01	0.16
TJ.SIM.AllnoNORvsAllNOR	0.00	0.00	0.09

Note:
TT - ToolText, TJ - ToolJava, SH - Sherlock

Table I.7: Contrasts effect sizes for SOCO D4

ContrastName	EffectSize	CI.LB	CI.UB
(Intercept)	1.00	NA	NA
Tool.TextvsJava	0.86	0.84	0.91
TT.SHvsOthers	0.81	0.79	0.88
TJ.SHvsOthers	0.90	0.90	0.94
TT.SIMvsJPlag	0.91	0.90	0.94
TJ.SIMvsJPlag	0.52	0.48	0.64
NoPPTvsPPT	0.13	0.01	0.27
SinglevsCombo	0.38	0.11	0.50
RCvsCCR	0.06	0.00	0.20
AnoNvsAN	0.00	0.00	0.17
Tool.TextvsJava:NoPPTvsPPT	0.01	0.00	0.09
TT.SHvsOthers:NoPPTvsPPT	0.09	0.01	0.17
TJ.SHvsOthers:NoPPTvsPPT	0.05	0.00	0.14
TT.SIMvsJPlag:NoPPTvsPPT	0.05	0.00	0.13
TJ.SIMvsJPlag:NoPPTvsPPT	0.08	0.01	0.15
Tool.TextvsJava:SinglevsCombo	0.20	0.11	0.28
TT.SHvsOthers:SinglevsCombo	0.10	0.02	0.18
TJ.SHvsOthers:SinglevsCombo	0.09	0.01	0.17
TT.SIMvsJPlag:SinglevsCombo	0.00	0.00	0.09
TJ.SIMvsJPlag:SinglevsCombo	0.06	0.00	0.13
Tool.TextvsJava:RCvsCCR	0.16	0.07	0.24
TT.SHvsOthers:RCvsCCR	0.08	0.01	0.16
TJ.SHvsOthers:RCvsCCR	0.07	0.01	0.15
TT.SIMvsJPlag:RCvsCCR	0.13	0.05	0.21
TJ.SIMvsJPlag:RCvsCCR	0.08	0.01	0.16
Tool.TextvsJava:AnoNvsAN	0.10	0.02	0.18
TT.SHvsOthers:AnoNvsAN	0.10	0.02	0.18
TJ.SHvsOthers:AnoNvsAN	0.09	0.01	0.17
TT.SIMvsJPlag:AnoNvsAN	0.00	0.00	0.09
TJ.SIMvsJPlag:AnoNvsAN	0.00	0.00	0.09

Note:
TT - ToolText, TJ - ToolJava, SH - Sherlock, AnoN - AllnoNOR, AN - AllNOR

Table I.8: Simple effect analysis effect sizes for SOCO D4

ContrastName	EffectSize	CI.LB	CI.UB
(Intercept)	1.00	1.00	1.00
TextvsJava	0.86	0.84	0.91
TT.SHvsOthers	0.81	0.80	0.88
TT.SIMvsJPlag	0.91	0.90	0.94
TJ.SHvsOthers	0.90	0.90	0.94
TJ.SIMvsJPlag	0.52	0.48	0.64
TT.SH.NoPPTvsPPT	0.10	0.02	0.17
TT.SH.SinglevsCombo	0.22	0.14	0.31
TT.SH.RCvsCCR	0.01	0.00	0.09
TT.SH.AllnoNORvsAllNOR	0.11	0.03	0.19
TT.JPlag.NoPPTvsPPT	0.02	0.00	0.10
TT.JPlag.SinglevsCombo	0.11	0.04	0.19
TT.JPlag.RCvsCCR	0.01	0.00	0.09
TT.JPlag.AllnoNORvsAllNOR	0.00	0.00	0.09
TT.SIM.NoPPTvsPPT	0.04	0.00	0.12
TT.SIM.SinglevsCombo	0.10	0.03	0.18
TT.SIM.RCvsCCR	0.16	0.08	0.25
TT.SIM.AllnoNORvsAllNOR	0.00	0.00	0.09
TJ.SH.NoPPTvsPPT	0.02	0.00	0.10
TJ.SH.SinglevsCombo	0.07	0.01	0.15
TJ.SH.RCvsCCR	0.12	0.05	0.20
TJ.SH.AllnoNORvsAllNOR	0.10	0.03	0.19
TJ.JPlag.NoPPTvsPPT	0.09	0.01	0.16
TJ.JPlag.SinglevsCombo	0.06	0.01	0.14
TJ.JPlag.RCvsCCR	0.09	0.02	0.17
TJ.JPlag.AllnoNORvsAllNOR	0.00	0.00	0.09
TJ.SIM.NoPPTvsPPT	0.01	0.00	0.09
TJ.SIM.SinglevsCombo	0.01	0.00	0.09
TJ.SIM.RCvsCCR	0.01	0.00	0.09
TJ.SIM.AllnoNORvsAllNOR	0.00	0.00	0.09

Note:
TT - ToolText, TJ - ToolJava, SH - Sherlock

Table I.9: Contrasts effect sizes for RSS A1

ContrastName	EffectSize	CI.LB	CI.UB
(Intercept)	0.89	0.87	0.92
Tool.TextvsJava	0.29	0.14	0.47
TT.SHvsOthers	0.48	0.37	0.63
TJ.SHvsOthers	0.52	0.40	0.66
TT.SIMvsJPlag	0.46	0.34	0.61
TJ.SIMvsJPlag	0.21	0.05	0.39
NoPPTvsPPT	0.57	0.43	0.76
SinglevsCombo	0.50	0.34	0.71
CCRvsTE	0.38	0.17	0.62
AnoNvsAN	0.08	0.01	0.35
Tool.TextvsJava:NoPPTvsPPT	0.16	0.01	0.40
TT.SHvsOthers:NoPPTvsPPT	0.26	0.04	0.48
TJ.SHvsOthers:NoPPTvsPPT	0.10	0.01	0.34
TT.SIMvsJPlag:NoPPTvsPPT	0.17	0.01	0.41
TJ.SIMvsJPlag:NoPPTvsPPT	0.22	0.02	0.45
Tool.TextvsJava:SinglevsCombo	0.10	0.01	0.35
TT.SHvsOthers:SinglevsCombo	0.19	0.01	0.42
TJ.SHvsOthers:SinglevsCombo	0.13	0.01	0.36
TT.SIMvsJPlag:SinglevsCombo	0.12	0.01	0.36
TJ.SIMvsJPlag:SinglevsCombo	0.06	0.00	0.30
Tool.TextvsJava:CCRvsTE	0.25	0.03	0.47
TT.SHvsOthers:CCRvsTE	0.31	0.08	0.52
TJ.SHvsOthers:CCRvsTE	0.11	0.01	0.35
TT.SIMvsJPlag:CCRvsTE	0.29	0.05	0.50
TJ.SIMvsJPlag:CCRvsTE	0.04	0.00	0.29
Tool.TextvsJava:AnoNvsAN	0.11	0.01	0.35
TT.SHvsOthers:AnoNvsAN	0.03	0.00	0.29
TJ.SHvsOthers:AnoNvsAN	0.19	0.01	0.43
TT.SIMvsJPlag:AnoNvsAN	0.00	0.00	0.29
TJ.SIMvsJPlag:AnoNvsAN	0.00	0.00	0.29

Note:
TT - ToolText, TJ - ToolJava, SH - Sherlock, AnoN - AllnoNOR, AN - AllNOR

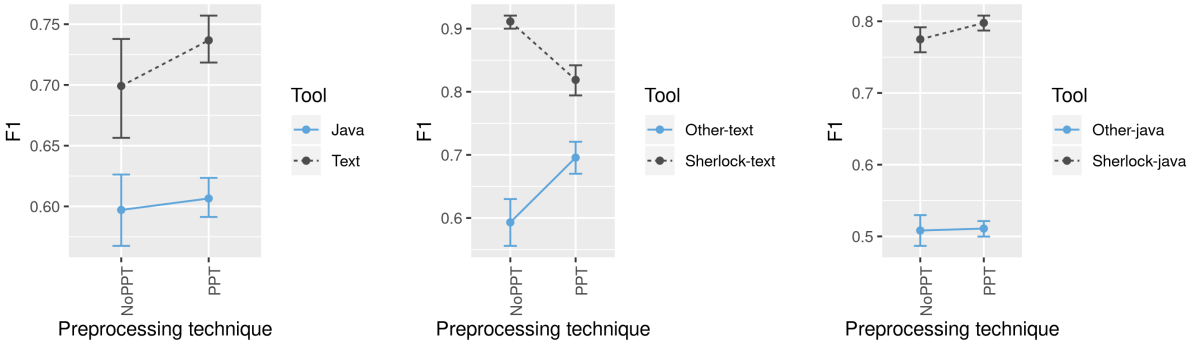
Table I.10: Simple effect analysis effect sizes for RSS A1

ContrastName	EffectSize	CI.LB	CI.UB
(Intercept)	0.89	0.87	0.92
TextvsJava	0.29	0.14	0.47
TT.SHvsOthers	0.48	0.37	0.63
TT.SIMvsJPlag	0.46	0.33	0.61
TJ.SHvsOthers	0.52	0.40	0.66
TJ.SIMvsJPlag	0.21	0.05	0.39
TT.SH.NoPPTvsPPT	0.13	0.01	0.36
TT.SH.SinglevsCombo	0.12	0.01	0.35
TT.SH.CCRvsTE	0.01	0.00	0.27
TT.SH.AllnoNORvsAllNOR	0.04	0.00	0.28
TT.JPlag.NoPPTvsPPT	0.49	0.30	0.65
TT.JPlag.SinglevsCombo	0.26	0.04	0.46
TT.JPlag.CCRvsTE	0.51	0.32	0.66
TT.JPlag.AllnoNORvsAllNOR	0.00	0.00	0.27
TT.SIM.NoPPTvsPPT	0.31	0.09	0.51
TT.SIM.SinglevsCombo	0.39	0.18	0.57
TT.SIM.CCRvsTE	0.18	0.01	0.40
TT.SIM.AllnoNORvsAllNOR	0.00	0.00	0.27
TJ.SH.NoPPTvsPPT	0.14	0.01	0.36
TJ.SH.SinglevsCombo	0.28	0.06	0.48
TJ.SH.CCRvsTE	0.02	0.00	0.27
TJ.SH.AllnoNORvsAllNOR	0.22	0.02	0.44
TJ.JPlag.NoPPTvsPPT	0.38	0.17	0.56
TJ.JPlag.SinglevsCombo	0.18	0.01	0.40
TJ.JPlag.CCRvsTE	0.14	0.01	0.36
TJ.JPlag.AllnoNORvsAllNOR	0.00	0.00	0.27
TJ.SIM.NoPPTvsPPT	0.10	0.01	0.33
TJ.SIM.SinglevsCombo	0.11	0.01	0.34
TJ.SIM.CCRvsTE	0.08	0.00	0.31
TJ.SIM.AllnoNORvsAllNOR	0.00	0.00	0.27

Note:
TT - ToolText, TJ - ToolJava, SH - Sherlock

CHAPTER J

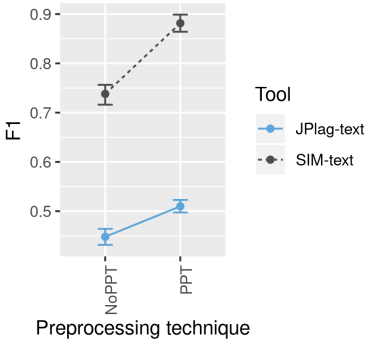
INTERACTION GRAPHS



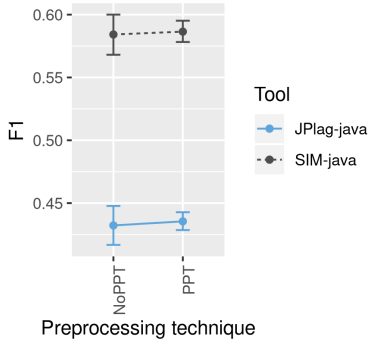
(a) Interaction 1

(b) Interaction 2

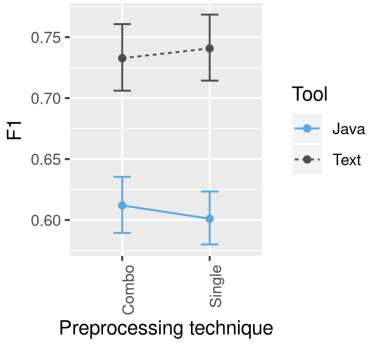
(c) Interaction 3



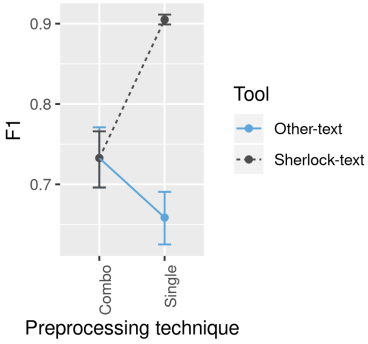
(d) Interaction 4



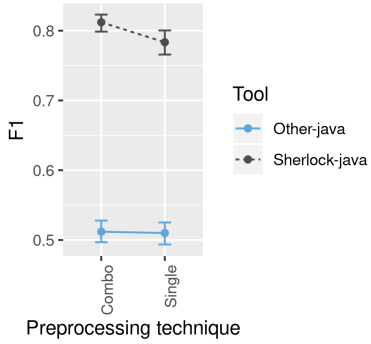
(e) Interaction 5



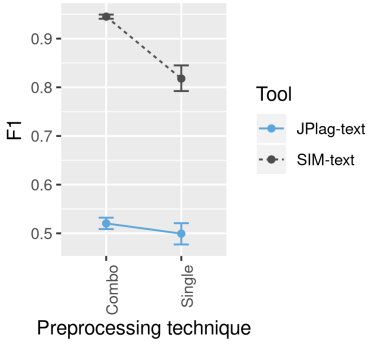
(f) Interaction 6



(g) Interaction 7

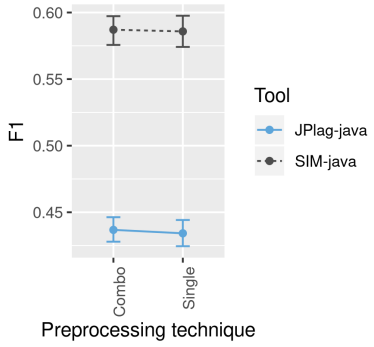


(h) Interaction 8

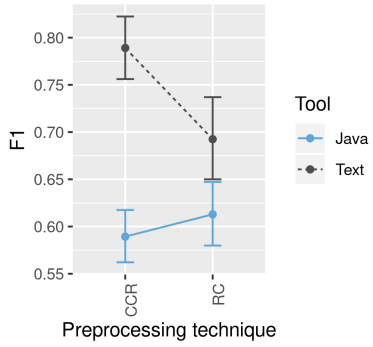


(i) Interaction 9

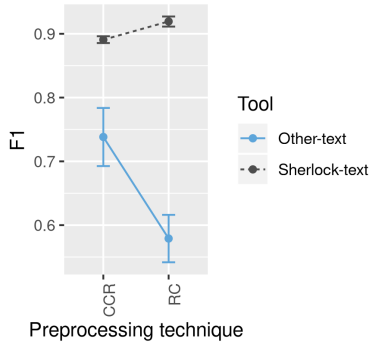
Figure J.1: Interaction graphs for SOCO D1 - part 1



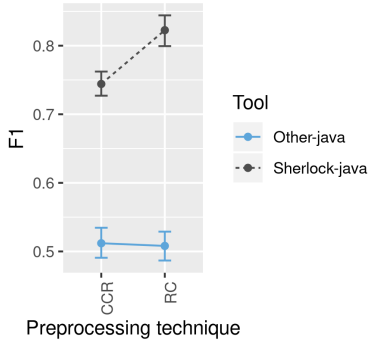
(a) Interaction 10



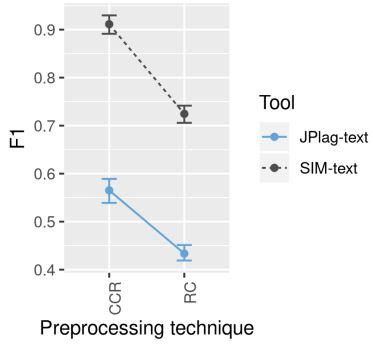
(b) Interaction 11



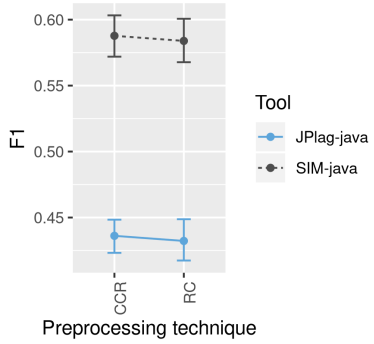
(c) Interaction 12



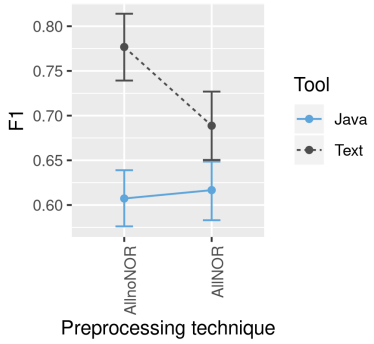
(d) Interaction 13



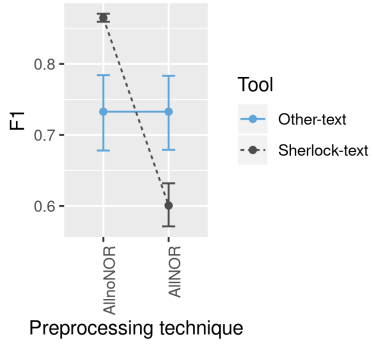
(e) Interaction 14



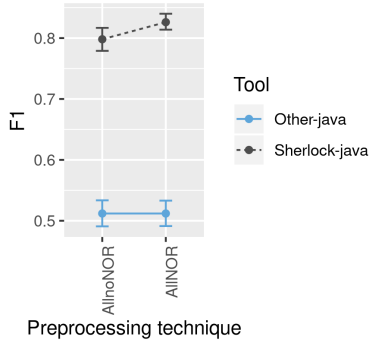
(f) Interaction 15



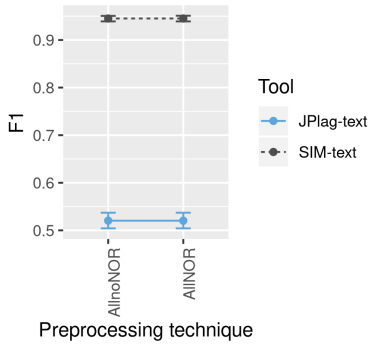
(g) Interaction 16



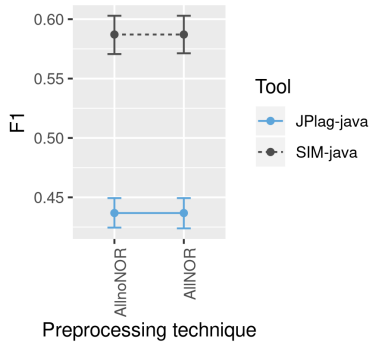
(h) Interaction 17



(i) Interaction 18

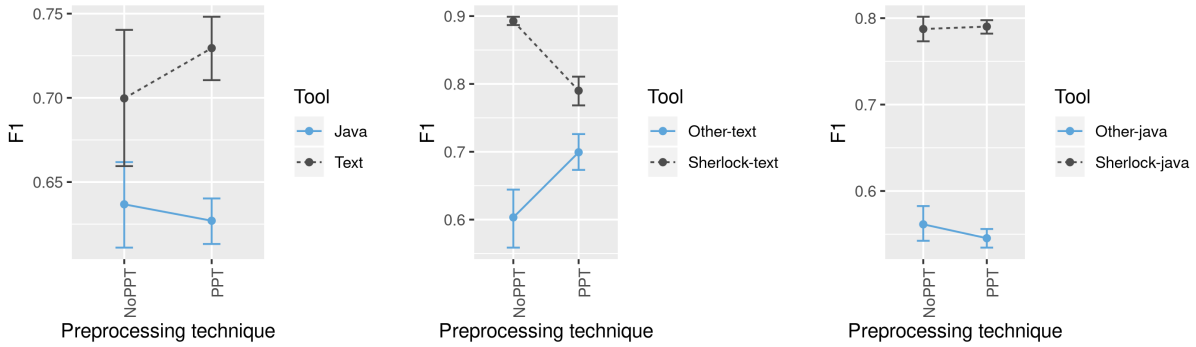


(j) Interaction 19



(k) Interaction 20

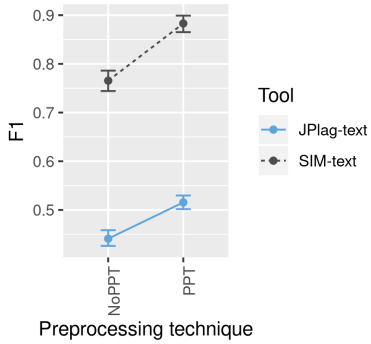
Figure J.2: Interaction graphs for SOCO D1 - part 2



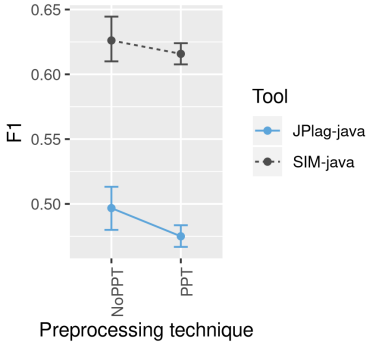
(a) Interaction 1

(b) Interaction 2

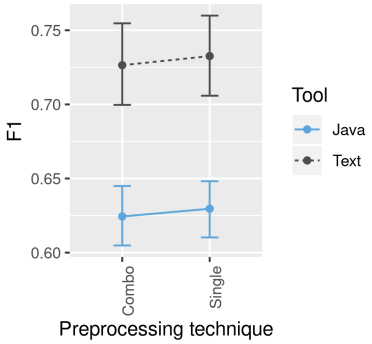
(c) Interaction 3



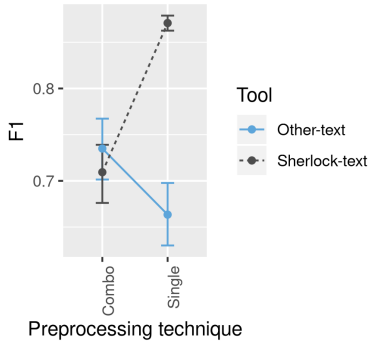
(d) Interaction 4



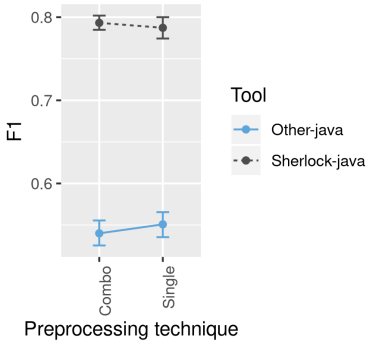
(e) Interaction 5



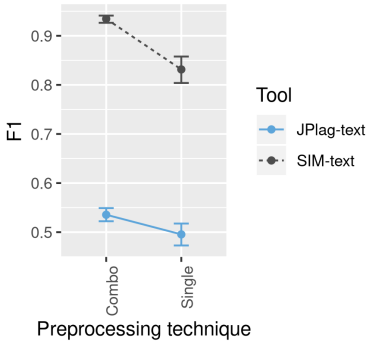
(f) Interaction 6



(g) Interaction 7

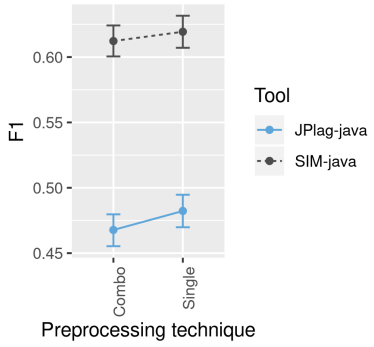


(h) Interaction 8

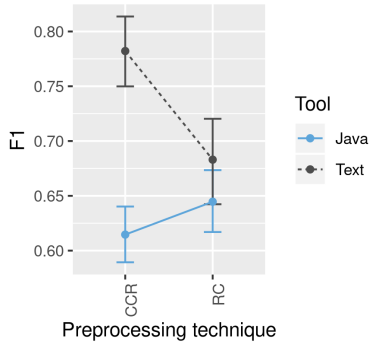


(i) Interaction 9

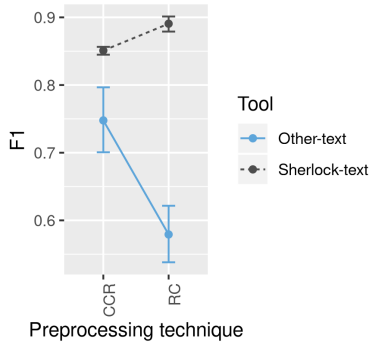
Figure J.3: Interaction graphs for SOCO D2 - part 1



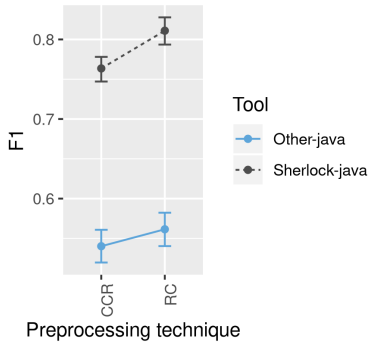
(a) Interaction 10



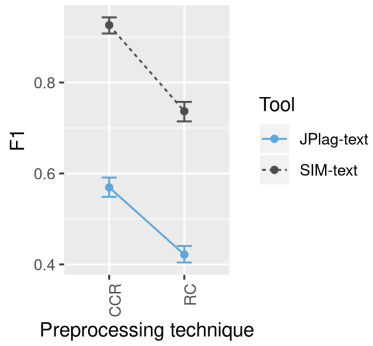
(b) Interaction 11



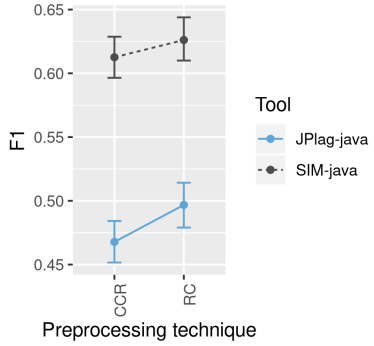
(c) Interaction 12



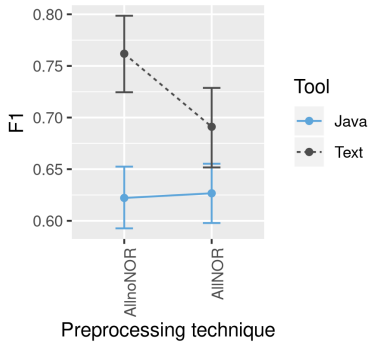
(d) Interaction 13



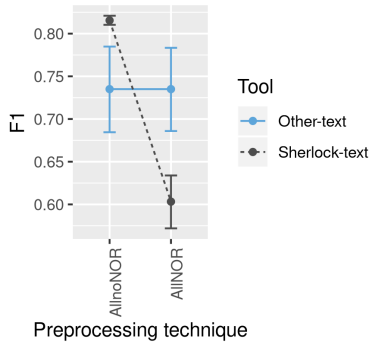
(e) Interaction 14



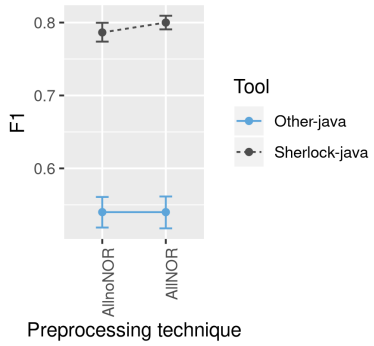
(f) Interaction 15



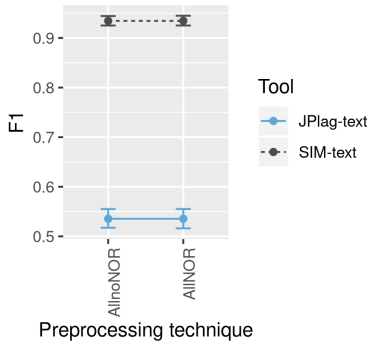
(g) Interaction 16



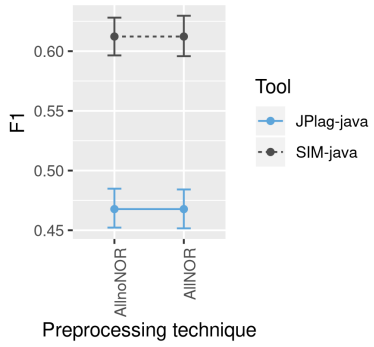
(h) Interaction 17



(i) Interaction 18

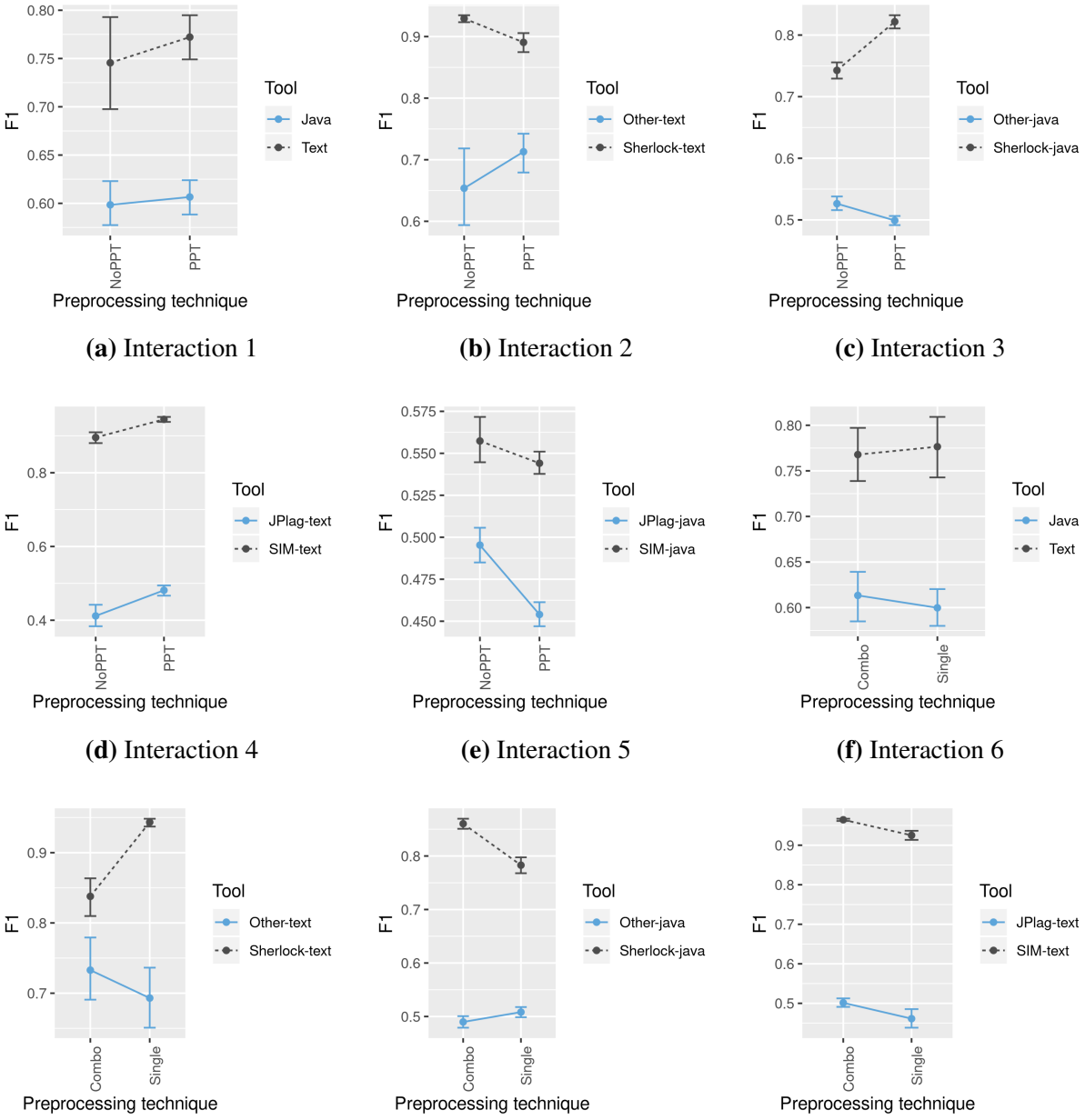


(j) Interaction 19



(k) Interaction 20

Figure J.4: Interaction graphs for SOCO D2 - part 2

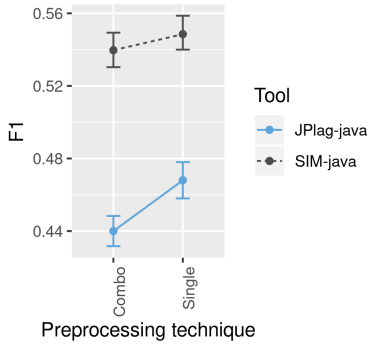


(a) Interaction 1 **(b)** Interaction 2 **(c)** Interaction 3

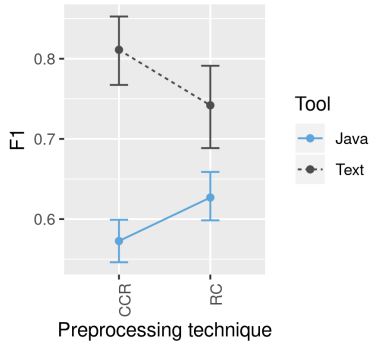
(d) Interaction 4 **(e)** Interaction 5 **(f)** Interaction 6

(g) Interaction 7 **(h)** Interaction 8 **(i)** Interaction 9

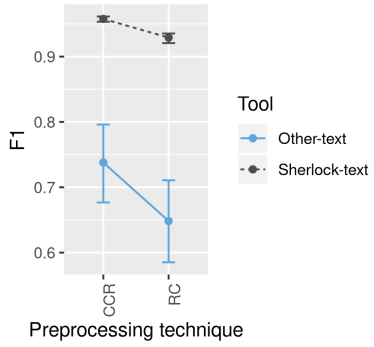
Figure J.5: Interaction graphs for SOCO D3 - part 1



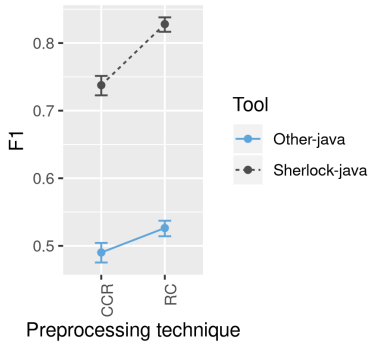
(a) Interaction 10



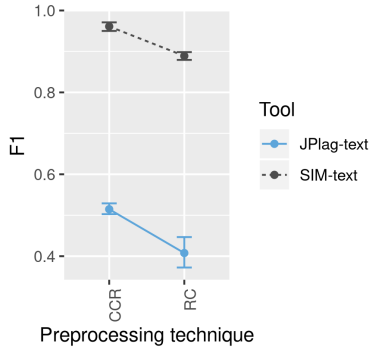
(b) Interaction 11



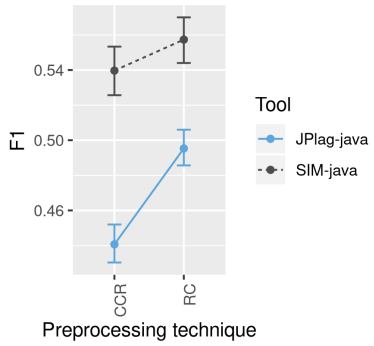
(c) Interaction 12



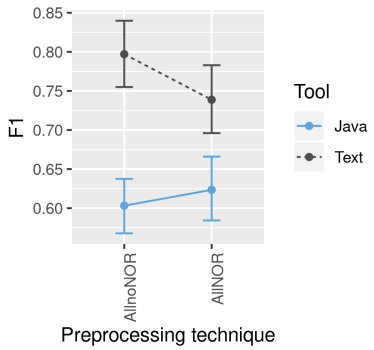
(d) Interaction 13



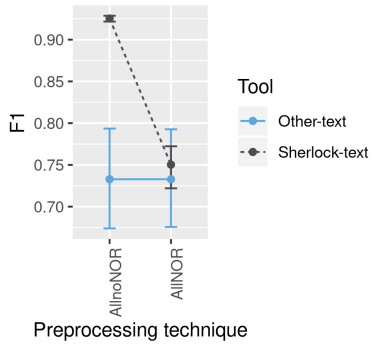
(e) Interaction 14



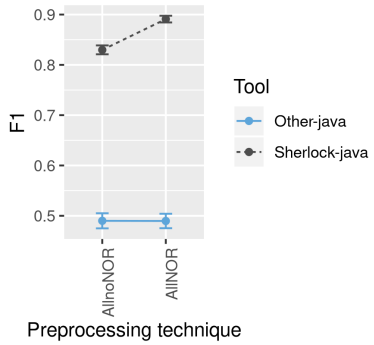
(f) Interaction 15



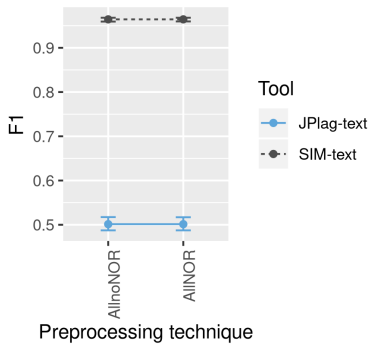
(g) Interaction 16



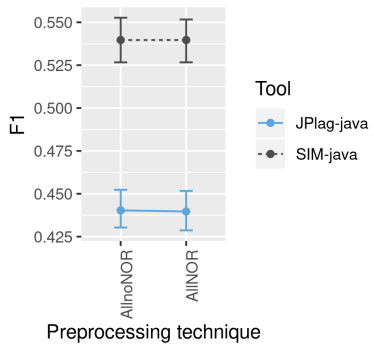
(h) Interaction 17



(i) Interaction 18

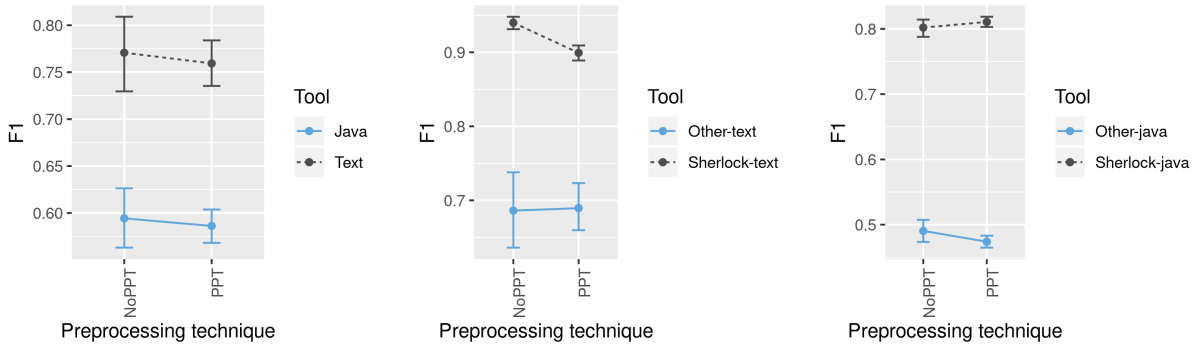


(j) Interaction 19



(k) Interaction 20

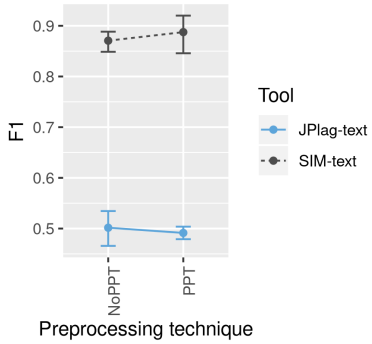
Figure J.6: Interaction graphs for SOCO D3 - part 2



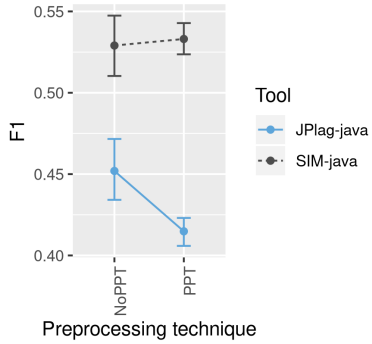
(a) Interaction 1

(b) Interaction 2

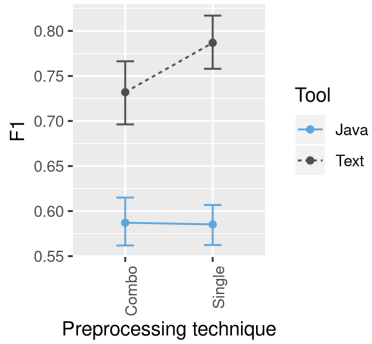
(c) Interaction 3



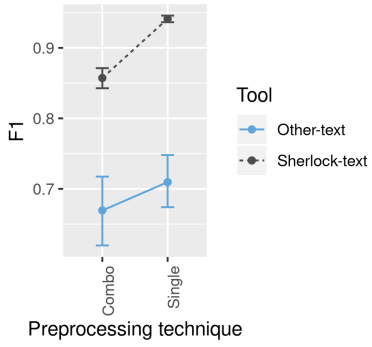
(d) Interaction 4



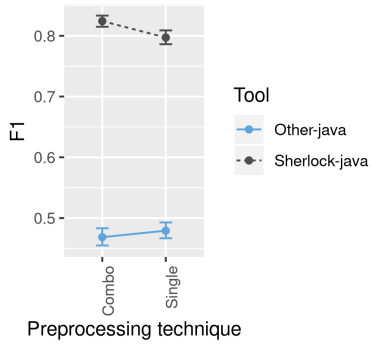
(e) Interaction 5



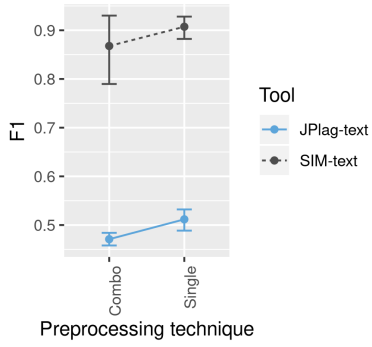
(f) Interaction 6



(g) Interaction 7

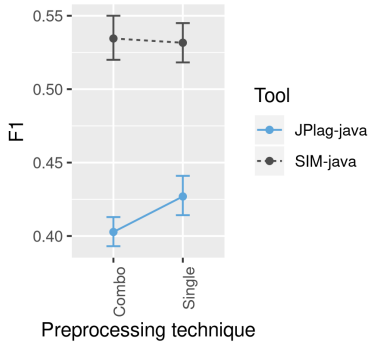


(h) Interaction 8

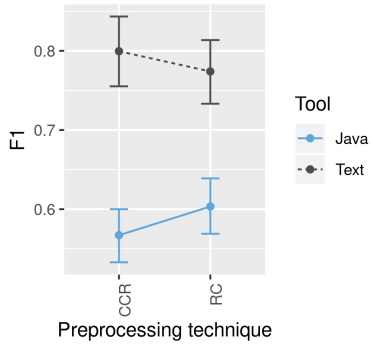


(i) Interaction 9

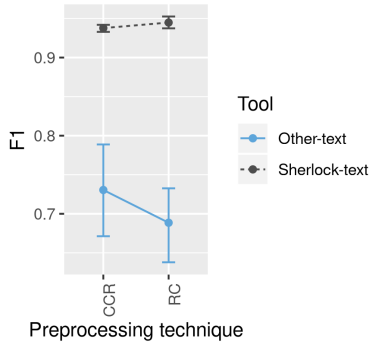
Figure J.7: Interaction graphs for SOCO D4 - part 1



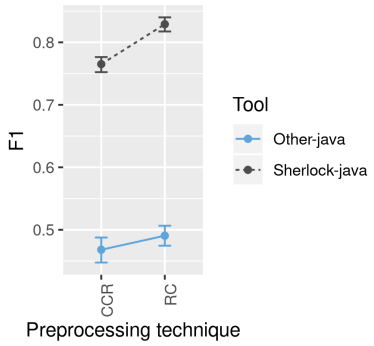
(a) Interaction 10



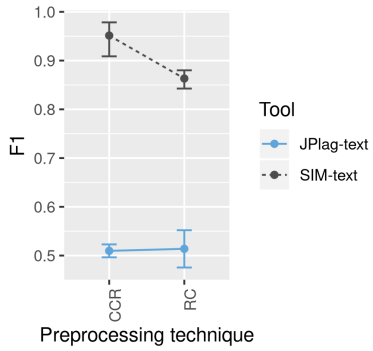
(b) Interaction 11



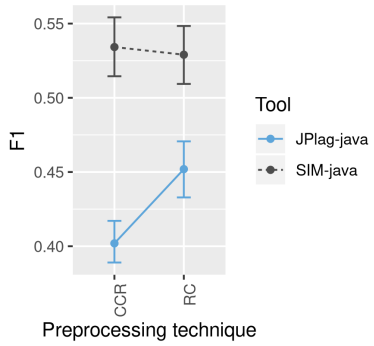
(c) Interaction 12



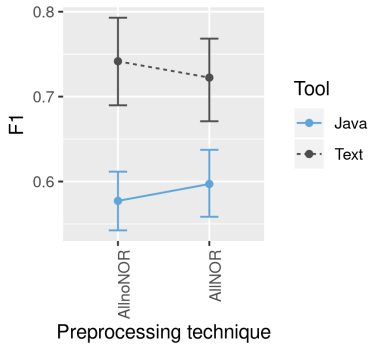
(d) Interaction 13



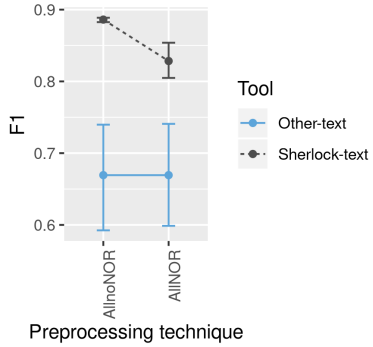
(e) Interaction 14



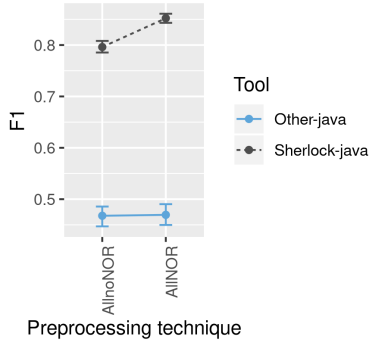
(f) Interaction 15



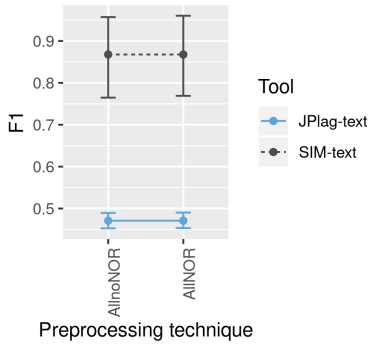
(g) Interaction 16



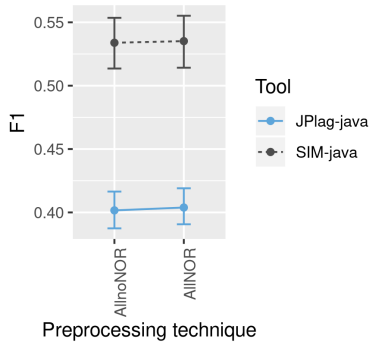
(h) Interaction 17



(i) Interaction 18



(j) Interaction 19



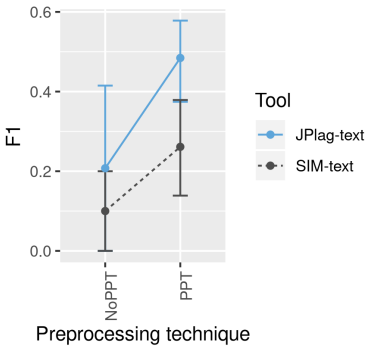
(k) Interaction 20

Figure J.8: Interaction graphs for SOCO D4 - part 2

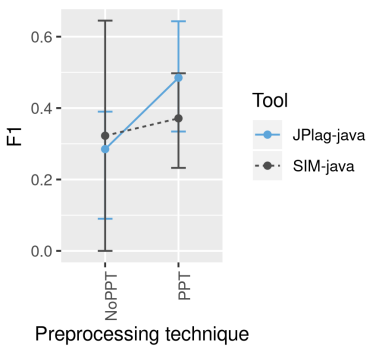
(a) Interaction 1

(b) Interaction 2

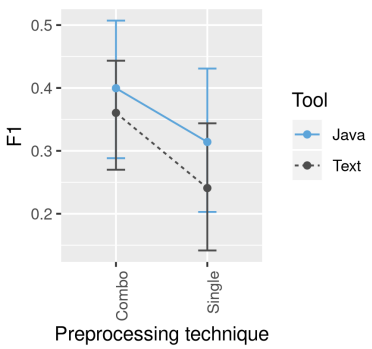
(c) Interaction 3



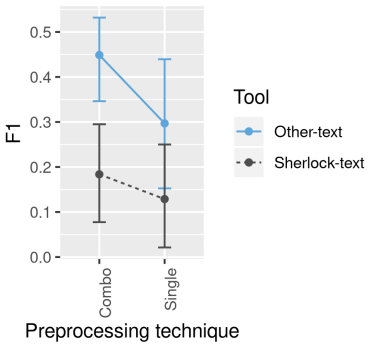
(d) Interaction 4



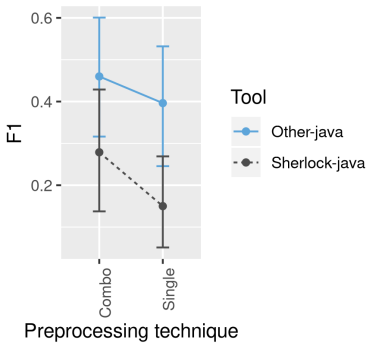
(e) Interaction 5



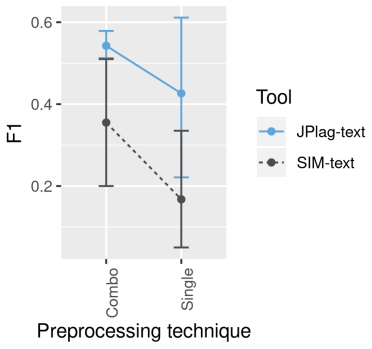
(f) Interaction 6



(g) Interaction 7

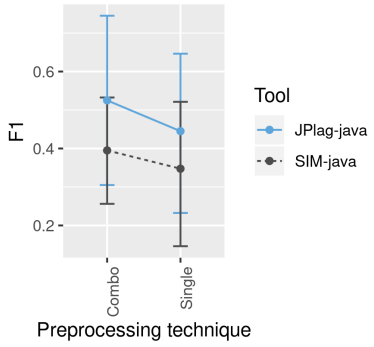


(h) Interaction 8

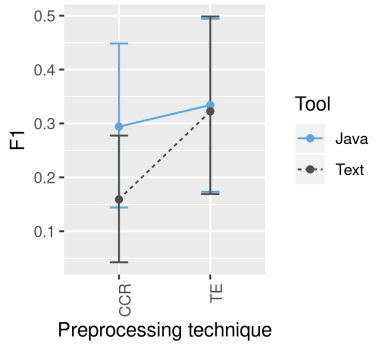


(i) Interaction 9

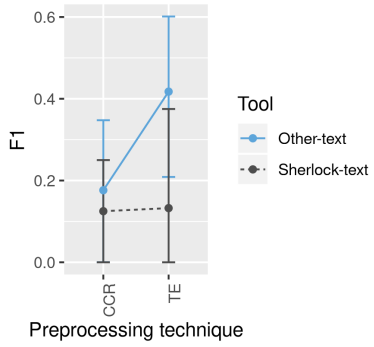
Figure J.9: Interaction graphs for RSS A1 - part 1



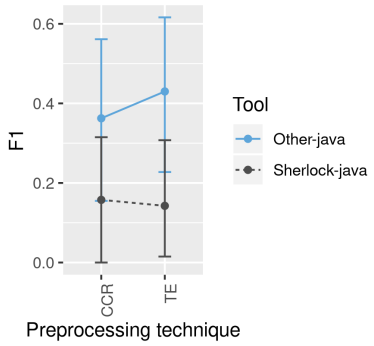
(a) Interaction 10



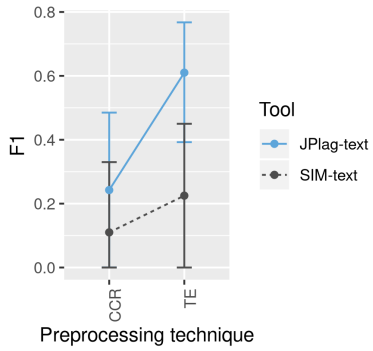
(b) Interaction 11



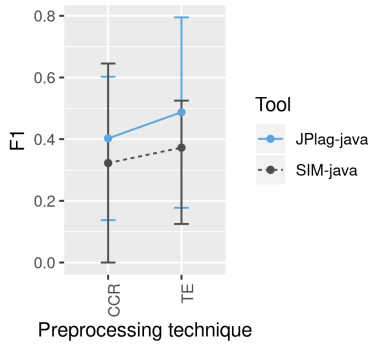
(c) Interaction 12



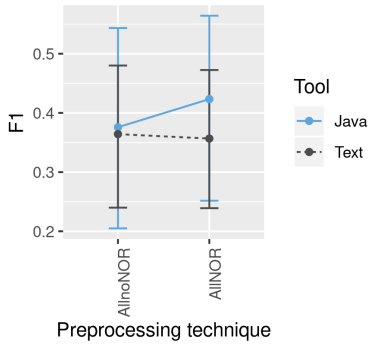
(d) Interaction 13



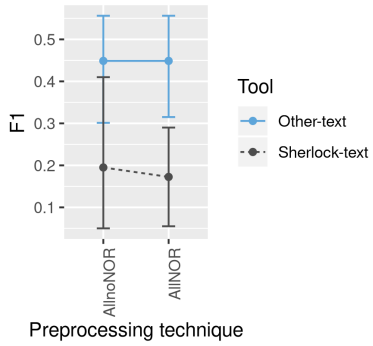
(e) Interaction 14



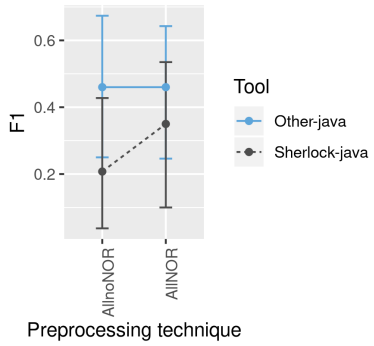
(f) Interaction 15



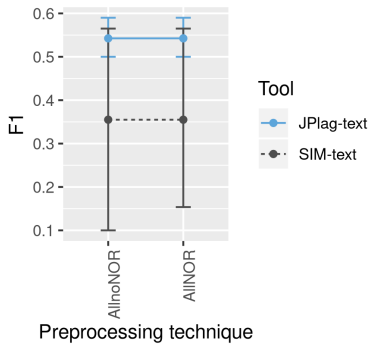
(g) Interaction 16



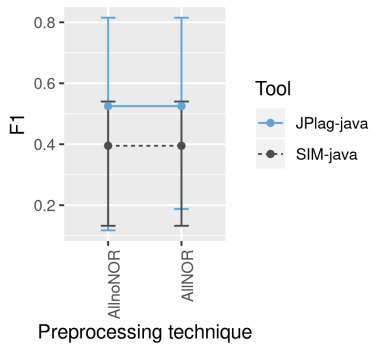
(h) Interaction 17



(i) Interaction 18



(j) Interaction 19



(k) Interaction 20

Figure J.10: Interaction graphs for RSS A1 - part 2

CHAPTER K

PRECISION AND RECALL FOR RSS DATASET

The folowing figures present the Precision and Recall for the RSS A1 dataset in the academic years: 2012-2013 (Figure K.1), 2013-2014 (Figure K.2), 2014-2015 (Figure K.3), and 2017-2018 Figure K.4.

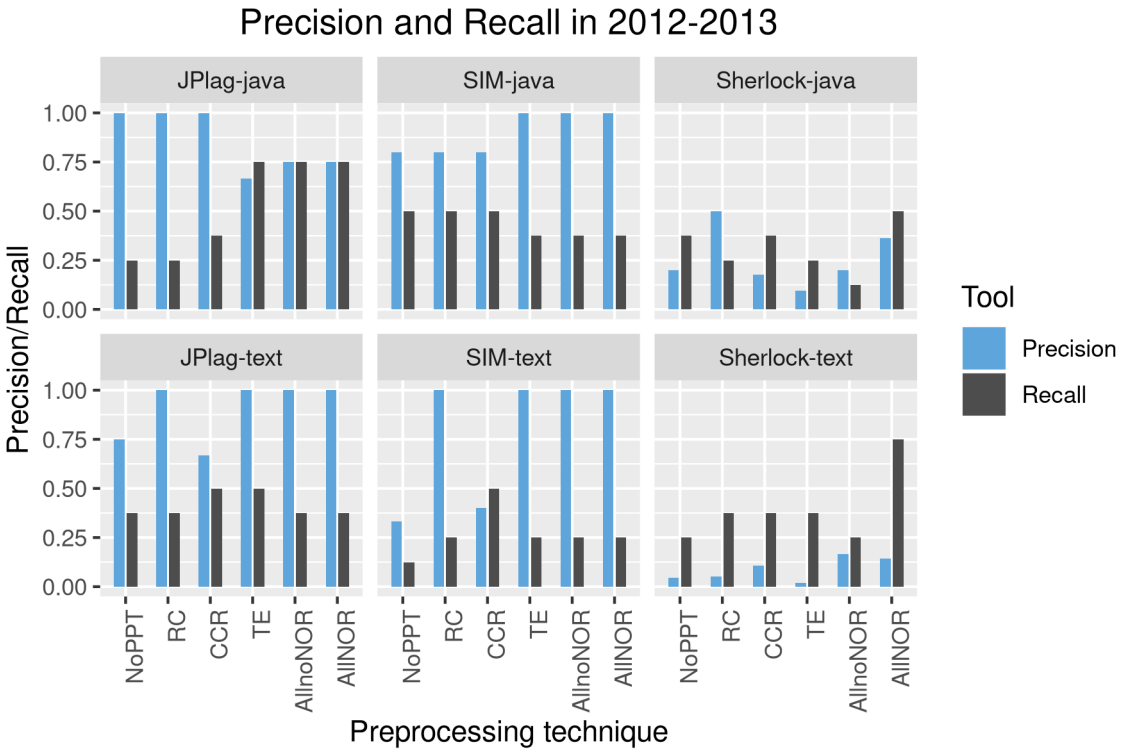


Figure K.1: Precision and Recall for RSS A1 assigment in academic year 2012-2013

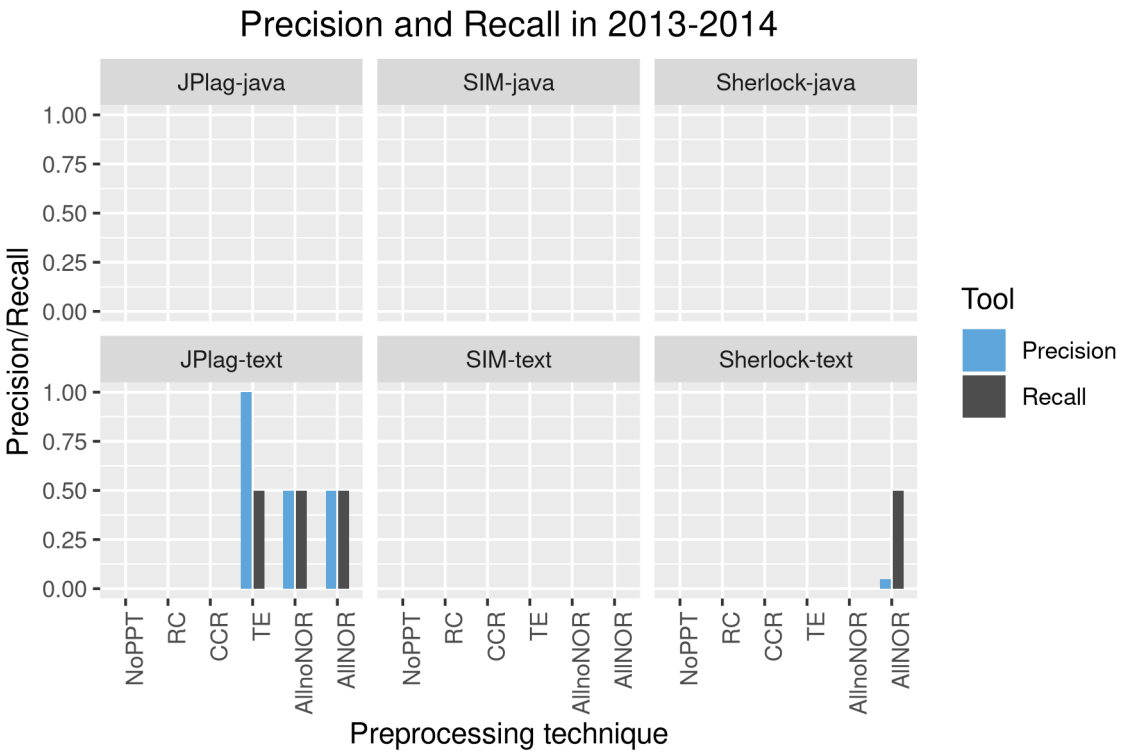


Figure K.2: Precision and Recall for RSS A1 assigment in academic year 2013-2014

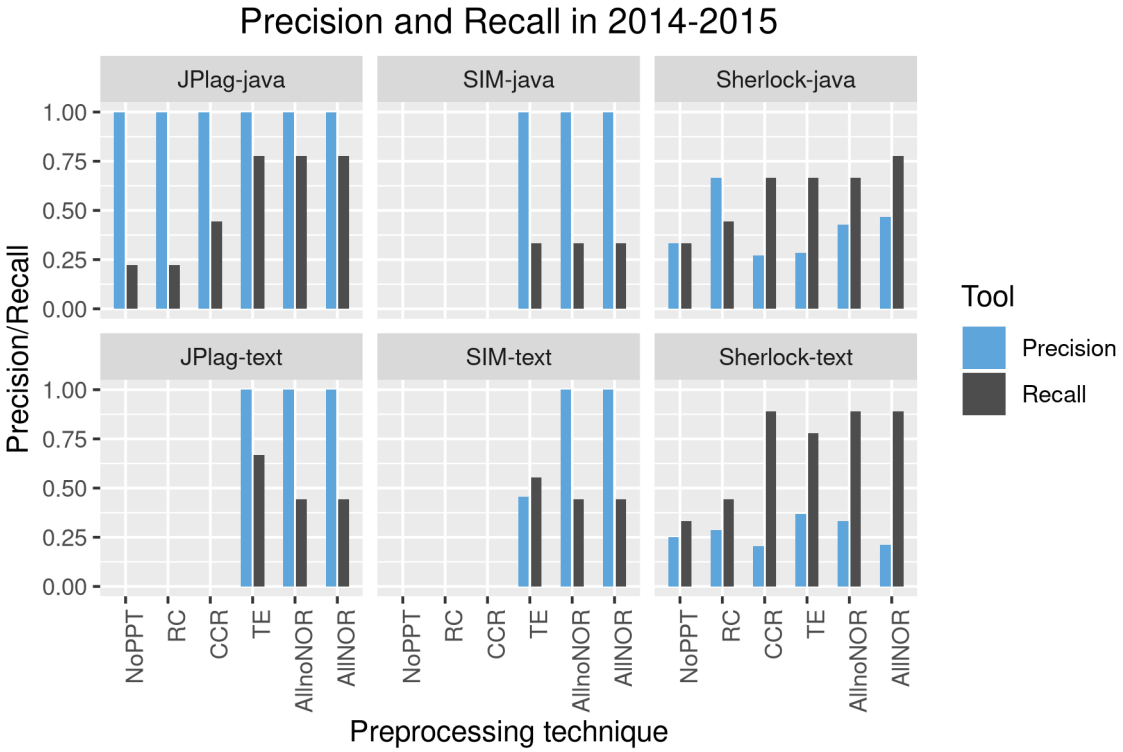


Figure K.3: Precision and Recall for RSS A1 assigment in academic year 2014-2015

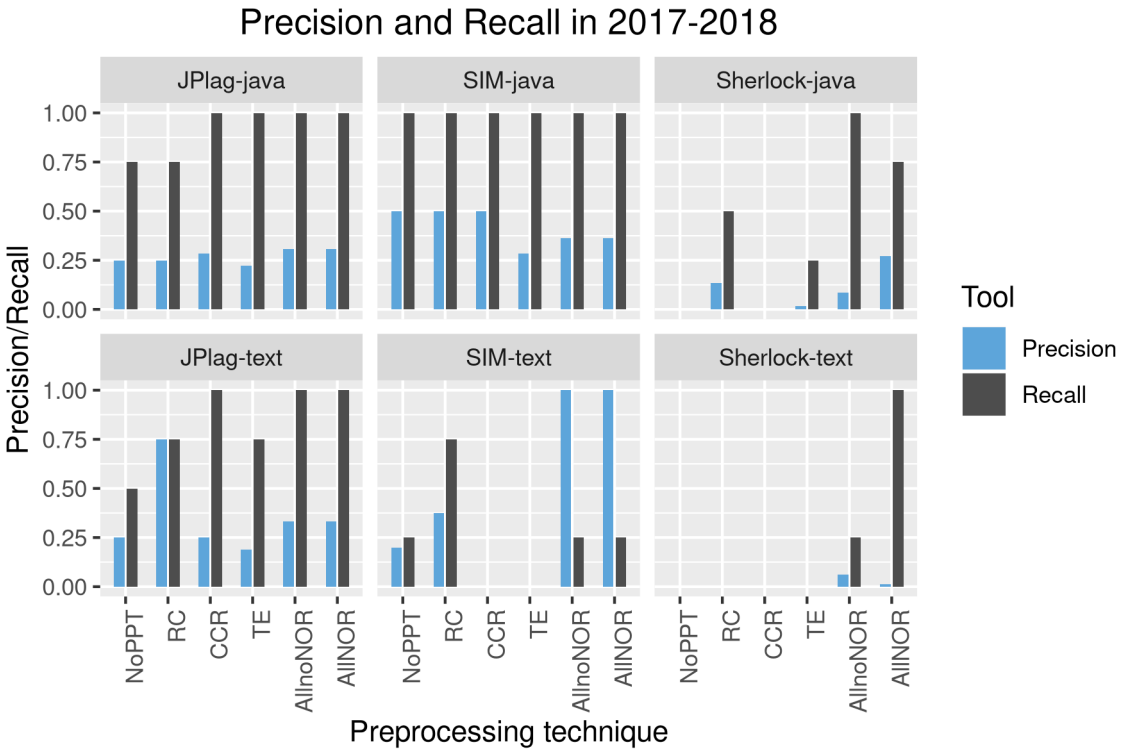


Figure K.4: Precision and Recall for RSS A1 assigment in academic year 2017-2018

CHAPTER L

LIST OF USED PACKAGES IN R

Here is a listing of the R session and loaded packages to perfom the various statistical analysis, table and graph creation.

```
  sessionInfo()

## R version 3.6.2 (2019-12-12)
## Platform: x86_64-pc-linux-gnu (64-bit)
## Running under: Arch Linux
##
## Matrix products: default
## BLAS:   /usr/lib/libblas.so.3.9.0
## LAPACK: /usr/lib/liblapack.so.3.9.0
##
## locale:
##  [1] LC_CTYPE=en_US.UTF-8       LC_NUMERIC=C
##  [3] LC_TIME=en_US.UTF-8        LC_COLLATE=en_US.UTF-8
##  [5] LC_MONETARY=en_US.UTF-8    LC_MESSAGES=en_US.UTF-8
##  [7] LC_PAPER=en_US.UTF-8       LC_NAME=C
##  [9] LC_ADDRESS=C               LC_TELEPHONE=C
## [11] LC_MEASUREMENT=en_US.UTF-8 LC_IDENTIFICATION=C
##
## attached base packages:
## [1] stats     graphics  grDevices utils     datasets  methods   base
##
## other attached packages:
##  [1] pbkrtest_0.4-7       qqplotr_0.0.3        boot_1.3-24
##  [4] lmerTest_3.1-1       lme4_1.1-21          Matrix_1.2-18
##  [7] pgirmess_1.6.9       compute.es_0.2-4     reshape_0.8.8
## [10] multcomp_1.4-11      TH.data_1.0-10       MASS_7.3-51.4
## [13] survival_3.1-8       mvtnorm_1.0-11       ez_4.4-0
## [16] nlme_3.1-142         car_3.0-6            carData_3.0-3
## [19] pastecs_1.3.21       dplyr_0.8.3          wordcloud_2.6
## [22] RColorBrewer_1.1-2   tm_0.7-7             NLP_0.2-0
## [25] UtilityFunctions_1.5 ggplot2_3.2.1        testthat_2.3.1
## [28] kableExtra_1.1.0     stringr_1.4.0        xtable_1.8-4
```

```
## [31] knitr_1.26
##
## loaded via a namespace (and not attached):
##   [1] readxl_1.3.1        backports_1.1.5     Hmisc_4.3-0
##   [4] plyr_1.8.5          lazyeval_0.2.2      sp_1.3-2
##   [7] splines_3.6.2       usethis_1.5.1       digest_0.6.23
##  [10] htmltools_0.4.0     gdata_2.18.0        fansi_0.4.0
##  [13] checkmate_1.9.4     magrittr_1.5        memoise_1.1.0
##  [16] cluster_2.1.0       openxlsx_4.1.4      remotes_2.1.0
##  [19] readr_1.3.1         gmodels_2.18.1      sandwich_2.5-1
##  [22] prettyunits_1.0.2   jpeg_0.1-8.1        colorspace_1.4-1
##  [25] rvest_0.3.5         haven_2.2.0         xfun_0.11
##  [28] rgdal_1.4-8         callr_3.4.0         crayon_1.3.4
##  [31] zeallot_0.1.0       zoo_1.8-6           glue_1.3.1
##  [34] gtable_0.3.0        webshot_0.5.2       pkgbuild_1.0.6
##  [37] DEoptimR_1.0-8      abind_1.4-5         scales_1.1.0
##  [40] DBI_1.1.0           Rcpp_1.0.3          htmlTable_1.13.3
##  [43] viridisLite_0.3.0   spData_0.3.2        units_0.6-5
##  [46] foreign_0.8-72      spdep_1.1-3         Formula_1.2-3
##  [49] htmlwidgets_1.5.1   httr_1.4.1          acepack_1.4.1
##  [52] ellipsis_0.3.0      pkgconfig_2.0.3     farver_2.0.1
##  [55] nnet_7.3-12         deldir_0.1-23       tidyselect_0.2.5
##  [58] labeling_0.3        rlang_0.4.2         reshape2_1.4.3
##  [61] munsell_0.5.0       cellranger_1.1.0    tools_3.6.2
##  [64] cli_2.0.0           splancs_2.01-40     devtools_2.2.1
##  [67] evaluate_0.14       processx_3.4.1      fs_1.3.1
##  [70] zip_2.0.4           robustbase_0.93-5   purrr_0.3.3
##  [73] slam_0.1-47         xml2_1.2.2          compiler_3.6.2
##  [76] rstudioapi_0.10     png_0.1-7           curl_4.3
##  [79] e1071_1.7-3         tibble_2.1.3        stringi_1.4.3
##  [82] ps_1.3.0            desc_1.2.0          forcats_0.4.0
##  [85] rgeos_0.5-2         lattice_0.20-38     classInt_0.4-2
##  [88] nloptr_1.2.1        vctrs_0.2.1         pillar_1.4.3
##  [91] LearnBayes_2.15.1   lifecycle_0.1.0     data.table_1.12.8
##  [94] maptools_0.9-9      R6_2.4.1            latticeExtra_0.6-29
##  [97] gridExtra_2.3       KernSmooth_2.23-16  rio_0.5.16
## [100] sessioninfo_1.1.1   codetools_0.2-16    gtools_3.8.1
## [103] assertthat_0.2.1    pkgload_1.0.2       rprojroot_1.3-2
```

```
## [106] withr_2.1.2         mgcv_1.8-31         expm_0.999-4
## [109] parallel_3.6.2      hms_0.5.2           rpart_4.1-15
## [112] grid_3.6.2          coda_0.19-3         class_7.3-15
## [115] minqa_1.2.4         rmarkdown_2.0       sf_0.8-0
## [118] numDeriv_2016.8-1.1 base64enc_0.1-3
```

BIBLIOGRAPHY

[1] C. Aasheim, P. Rutner, L. Li, and S. Williams, "Plagiarism and programming: A survey of student attitudes", *Journal of Information Systems Education*, vol. 23, no. 3, pp. 297–314, 2012, ISSN: 1055-3096.

[2] G. Acampora and G. Cosma, "A Fuzzy-based approach to programming language independent source-code plagiarism detection", in *IEEE International Conference on Fuzzy Systems*, Istanbul, Turkey: IEEE, 2015, pp. 1–8, ISBN: 978-1-4673-7428-6. DOI: 10.1109/FUZZ-IEEE.2015.7337935.

[3] A. Ahtiainen, S. Surakka, and M. Rahikainen, "Plaggie: GNU-licensed source code plagiarism detection engine for Java exercises.", in *Proceedings of the 6th Baltic Sea conference on Computing education research Koli Calling*, vol. 276, New York, USA: ACM Press, 2006, p. 141. DOI: 10.1145/1315803.1315831.

[4] M. Arevalillo-Herráez and J. M. Claver, "Assessment Technique to Encourage Cooperative Learning in a Computer Programming Course", *International Journal of Engineering Education*, vol. 27, no. 4, pp. 867–874, 2011, ISSN: 0949-149X.

[5] C. Arwin and S. M. M. Tahaghoghi, "Plagiarism detection across programming languages", in *29th Australasian Computer Science Conference*, vol. 48, Melbourne, Australia: Australian Computer Society, Inc., 2006, pp. 277–286, ISBN: 978-1-9206-8230-9.

[6] A. Asadullah, M. Basavaraju, I. Stern, and V. D. Bhat, "Design Patterns Based Preprocessing of Source Code for Plagiarism Detection", in *19th Asia-Pacific Software Engineering Conference*, vol. 2, Bangalore, India: IEEE, 2012, pp. 128–135, ISBN: 978-1-4673-4930-7. DOI: 10.1109/APSEC.2012.141.

[7] M. J. Austin and L. D. Brown, "Internet Plagiarism: Developing Strategies to Curb Student Academic Dishonesty", *The Internet and Higher Education*, vol. 2, no. 1, pp. 21–33, 1999, ISSN: 1096-7516. DOI: 10.1016/S1096-7516(99)00004-4.

[8] W. Badke, "Training plagiarism detectives: The law and order approach", *Online Magazine at questia.com*, vol. 31, no. 6, pp. 50–52, 2007, ISSN: 0146-5422.

[9] E. S. Banjanovic and J. W. Osborne, "Confidence intervals for effect sizes: Applying bootstrap resampling", *Practical Assessment, Research & Evaluation*, vol. 21, no. 5, pp. 1–20, 2016, ISSN: 1531-7714.

[10] E. Barnes, "Student Honor: A Study in Cheating", *The International Journal of Ethics*, vol. 14, no. 4, pp. 481–488, 1904, ISSN: 1526-422X. DOI: 10.1086/intejethi.14.4.2376257.

[11] R. Barrett and A. L. Cox, "At least they're learning something: the hazy line between collaboration and collusion", *Assessment & Evaluation in Higher Education*, vol. 30, no. 2, pp. 107–122, 2005, ISSN: 0260-2938. DOI: 10.1080/0260293042000264226.

[12] M. Bartoszuk and M. Gagolewski, "A Fuzzy R Code Similarity Detection Algorithm", in *15th International Conference on Information Processing and Management of Uncertainty in Knowledge-based Systems*, Part 3, vol. 444 CCIS, Warsaw, Poland: Springer Verlag, 2014, pp. 21–30, ISBN: 978-3-3190-8851-8. DOI: 10.1007/978-3-319-08852-5_3.

[13] K. Beck, *Test-driven development: by example*. Addison-Wesley Professional, 2003, ISBN: 978-0-3211-4653-3.

[14] A. M. Bejarano, L. E. García, and E. E. Zurek, "Detection of source code similitude in academic environments", *Computer Applications in Engineering Education*, vol. 23, no. 1, pp. 13–22, 2015, ISSN: 1061-3773. DOI: 10.1002/cae.21571.

[15] S. Bellon, R. Koschke, G. Antoniol, J. Krinke, and E. Merlo, "Comparison and Evaluation of Clone Detection Tools", *IEEE Transactions on Software Engineering*, vol. 33, no. 9, pp. 577–591, 2007, ISSN: 0098-5589. DOI: 10.1109/TSE.2007.70725.

[16] J. Bloch, *Effective java (3rd Edition)*. Addison-Wesley Professional, 2018, ISBN: 978-0-1346-8599-1.

[17] R. Brixtel, M. Fontaine, B. Lesner, C. Bazin, and R. Robbes, "Language-Independent Clone Detection Applied to Plagiarism Detection", in *10th IEEE Working Conference on Source Code Analysis and Manipulation*, Caen, France: IEEE, 2010, pp. 77–86, ISBN: 978-1-4244-8655-7. DOI: 10.1109/SCAM.2010.19.

[18] S. Burrows and S. M. M. Tahaghoghi, "Source code authorship attribution using n-grams", in *12th Australasian Document Computing Symposium*, RMIT University, Melbourne, Australia, 2007, pp. 32–39, ISBN: 978-0-6464-8437-2.

[19] S. Burrows, A. L. Uitdenbogerd, and A. Turpin, "Temporally Robust Software Features for Authorship Attribution", in *33rd Annual IEEE International Computer Software and Applications Conference*, vol. 1, 2009, pp. 599–606, ISBN: 978-0-7695-3726-9. DOI: 10.1109/COMPSAC.2009.85.

[20] S. Burrows, S. M. M. Tahaghoghi, and J. Zobel, "Efficient plagiarism detection for large code repositories", *Software: Practice and Experience*, vol. 37, no. 2, pp. 151–175, 2007, ISSN: 0038-0644. DOI: 10.1002/spe.750.

[21] S. Butakov and V. Shcherbinin, "On the number of search queries required for Internet plagiarism detection", in *9th IEEE International Conference on Advanced Learning Technologies*, IEEE, 2009, pp. 482–483, ISBN: 978-0-7695-3711-5. DOI: 10.1109/ICALT.2009.78.

[22] M Cebrian, M. Alfonseca, and A. Ortega, "Towards the Validation of Plagiarism Detection Tools by Means of Grammar Evolution", *IEEE Transactions on Evolutionary Computation*, vol. 13, no. 3, pp. 477–485, 2009, ISSN: 1941-0026. DOI: 10.1109/TEVC.2008.2008797.

[23] G. Chen, Y. Zhang, and X. Wang, "Analysis on Identification Technologies of Program Code Similarity", in *International Conference of Information Technology, Computer Engineering and Management Sciences*, vol. 1, Nanjing, Jiangsu, China: IEEE, 2011, pp. 188–191, ISBN: 978-1-4577-1419-1. DOI: 10.1109/ICM.2011.240.

[24] X. Chen, B. Francia, M. Li, B. McKinnon, and A. Seker, "Shared Information and Program Plagiarism Detection", *IEEE Transactions on Information Theory*, vol. 50, no. 7, pp. 1545–1551, 2004, ISSN: 0018-9448. DOI: 10.1109/TIT.2004.830793.

[25] J. Choma, E. M. Guerra, and T. S. da Silva, "Developers' Initial Perceptions on TDD Practice: A Thematic Analysis with Distinct Domains and Languages", in *International Conference on Agile Software Development*, 2018, pp. 68–85, ISBN: 978-3-319-91601-9. DOI: 10.1007/978-3-319-91602-6_5.

[26] D. Chuda and P. Navrat, "Support for checking plagiarism in e-learning", *Procedia - Social and Behavioral Sciences*, vol. 2, no. 2, pp. 3140–3144, 2010, ISSN: 1877-0428. DOI: 10.1016/j.sbspro.2010.03.478.

[27] D. Chuda, P. Navrat, B. Kovacova, and P. Humay, "The Issue of (Software) Plagiarism: A Student View", *IEEE Transactions on Education*, vol. 55, no. 1, pp. 22–28, 2012, ISSN: 0018-9359. DOI: 10.1109/TE.2011.2112768.

[28] V. Ciesielski, N. Wu, and S. Tahaghoghi, "Evolving similarity functions for code plagiarism detection", in *Proceedings of the 10th annual conference on Genetic and evolutionary computation*, Georgia, USA: ACM Press, 2008, pp. 1453–1460, ISBN: 978-1-6055-8130-9. DOI: 10.1145/1389095.1389380.

[29] F. Cirillo, *The Pomodoro Technique: The Acclaimed Time-Management System That Has Transformed How We Work*. Currency, 2018, ISBN: 978-1-5247-6070-0.

[30] P. Clough, *Plagiarism in natural and programming languages: an overview of current tools and technologies*, Research Memoranda: CS-00-05, Department of Computer Science, University of Sheffield, Accessed on 2016-07-25 Available at http://ir.shef.ac.uk/cloughie/papers/plagiarism2000.pdf, Sheffield, UK, 2000.

[31] G. Cosma and M. Joy, "An Approach to Source-Code Plagiarism Detection and Investigation Using Latent Semantic Analysis", *IEEE Transactions on Computers*, vol. 61, no. 3, pp. 379–394, 2012, ISSN: 0018-9340. DOI: 10.1109/TC.2011.223.

[32] G. Cosma and M. Joy, "Evaluating the performance of LSA for source-code plagiarism detection", *Informatica (Slovenia)*, vol. 36, no. 4, pp. 409–424, 2012, ISSN: 0350-5596.

[33] G. Cosma, "An approach to source-code plagiarism detection and investigation using latent semantic analysis", PhD thesis, University of Warwick, Accessed on 2016-07-25 Available at http://wrap.warwick.ac.uk/3575/, 2008.

[34] G. Cosma and M. Joy, "Towards a Definition of Source-Code Plagiarism", *IEEE Transactions on Education*, vol. 51, no. 2, pp. 195–200, 2008, ISSN: 0018-9359. DOI: 10.1109/TE.2007.906776.

[35] A. Ramírez-de-la Cruz, G. Ramírez-de-la Rosa, C. Sánchez-Sánchez, and H. Jiménez-Salazar, "On the Importance of Lexicon, Structure and Style for Identifying Source Code Plagiarism", in *Proceedings of the Forum for Information Retrieval Evaluation*, ser. FIRE '14, New York, USA: ACM Press, 2015, pp. 31–38, ISBN: 978-1-4503-3755-7. DOI: 10.1145/2824864.2824879.

[36] C. Daly, "A Technique for Detecting Plagiarism in Computer Code", *The Computer Journal*, vol. 48, no. 6, pp. 662–666, 2005, ISSN: 0010-4620. DOI: 10.1093/comjnl/bxh139.

[37] E. L. Deci and R. M. Ryan, "A motivational approach to self: Integration in personality", *Perspectives on motivation*, vol. 38, no. 237, pp. 237–288, 1990.

[38] Z. Đurić and D. Gašević, "A Source Code Similarity System for Plagiarism Detection", *The Computer Journal*, vol. 56, no. 1, pp. 70–86, 2013, ISSN: 0010-4620. DOI: 10.1093/comjnl/bxs018.

[39] J. Dobša, D. Mladenić, J. Rupnik, D. Radošević, and I. Magdalenić, "Cross-language information retrieval by reduced k-means", *International Journal of Computer Information Systems and Industrial Managerment Applications*, vol. 10, no. 1, pp. 314–322, 2018, ISSN: 2150-7988.

[40] C. Domin, H. Pohl, and M. Krause, "Improving Plagiarism Detection in Coding Assignments by Dynamic Removal of Common Ground", in *Proceedings of the 2016 CHI Conference Extended Abstracts on Human Factors in Computing Systems*, ser. CHI EA '16, New York, USA: ACM Press, 2016, pp. 1173–1179, ISBN: 978-1-4503-4082-3. DOI: 10.1145/2851581.2892512.

[41] J. L. Donaldson, A.-M. Lancaster, and P. H. Sposato, "A plagiarism detection system", *ACM SIGCSE Bulletin*, SIGCSE '81, vol. 13, no. 1, pp. 21–25, 1981, ISSN: 0097-8418. DOI: 10.1145/953049.800955.

[42] R. A. Easterlin, "Income and happiness: Towards a unified theory", *The economic journal*, vol. 111, no. 473, pp. 465–484, 2001, ISSN: 0013-0133. DOI: 10.1111/1468-0297.00646.

[43] Encyclopædia Britannica Inc, *Plagiarism*, Accessed on 2015-09-10 Available at http://www.britannica.com/topic/plagiarism, 2015.

[44] J. Faidhi and S. Robinson, "An empirical approach for detecting program similarity and plagiarism within a university programming environment", *Computers & Education*, vol. 11, no. 1, pp. 11–19, 1987, ISSN: 0360-1315. DOI: 10.1016/0360-1315(87)90042-X.

[45] Y. Fain, *Java Programming 24-Hour Trainer*. Wiley Publishing, Inc., 2011, ISBN: 978-0-470-88964-0.

[46] A. Field, J. Miles, and Z. Field, *Discovering statistics using R*. SAGE Publications Ltd, 2012, ISBN: 978-1-4462-0046-9.

[47] R. Finnie, W. Poirier, E. Bozkurt, J. B. Peterson, T. Fricker, and M. Pratt, "Using Future Authoring to Improve Student Outcomes", Higher Education Quality Council of Ontario, Toronto, Canada, Tech. Rep., 2017, Accessed on 2016-07-25 Available at http://www.heqco.ca/en-ca/Research/ResPub/Pages/Using-Future-Authoring-to-Improve-Student-Outcomes-.aspx, pp. 1–50.

[48] E. Flores, A. Barrón-Cedeño, L. Moreno, and P. Rosso, "Cross-language source code re-use detection using latent semantic analysis", *Journal of Universal Computer Science*, vol. 21, no. 13, pp. 1708–1725, 2015, ISSN: 0948-695X.

[49] E. Flores, P. Rosso, L. Moreno, and E. Villatoro-Tello, "Pan@ fire 2015: Overview of cl-soco track on the detection of cross-language source code re-use", in *Proceedings of the Seventh Forum for Information Retrieval Evaluation*, Gandhinagar, India, 2015, pp. 4–6.

[50] E. Flores, A. Barrón-Cedeño, L. Moreno, and P. Rosso, "Uncovering source code reuse in large-scale academic environments", *Computer Applications in Engineering Education*, vol. 23, no. 3, pp. 383–390, 2015, ISSN: 1061-3773. DOI: 10.1002/cae.21608.

[51] E. Flores, L. Moreno, and P. Rosso, "Detecting Source Code Re-Use with Ensemble Models", in *Proceedings of the 4th Spanish Conference on Information Retrieval*, New York, USA: ACM Press, 2016, pp. 1–7, ISBN: 9781450341417. DOI: 10.1145/2934732.2934738.

[52] E. Flores, P. Rosso, L. Moreno, and E. Villatoro-Tello, "On the Detection of SOurce COde Re-use", in *Proceedings of the Forum for Information Retrieval Evaluation*, New York, USA: ACM Press, 2015, pp. 21–30, ISBN: 978-1-4503-3755-7. DOI: 10.1145/2824864.2824878.

[53] M. Fowler, K. Beck, J. Brant, W. Opdyke, and D. Roberts, *Refactoring: improving the design of existing code*. Addison-Wesley Professional, 1999, ISBN: 978-0-2014-8567-7.

[54] R. Fraser, "Collaboration, Collusion and Plagiarism in Computer Science Coursework", *Informatics in Education*, vol. 13, no. 2, pp. 179–195, 2014, ISSN: 16485831. DOI: 10.15388/infedu.2014.01.

[55] S. Freeman and N. Pryce, *Growing Object-oriented Software: Guided by Tests*. Pearson Education India, 2009, ISBN: 978-0-3215-0362-6.

[56] M. Freire, "Visualizing program similarity in the Ac plagiarism detection system", in *Proceedings of the working conference on Advanced visual interfaces*, New York, USA: ACM Press, 2008, pp. 404–407, ISBN: 978-1-6055-8141-5. DOI: 10.1145/1385569.1385644.

[57] E. Gamma, R. Helm, R. Johnson, and J. Vlissides, *Design Patterns: Elements of Reusable Object-Oriented Software*. Pearson Education, 2005, ISBN: 978-1-4058-3730-9.

[58] D. Ganguly and G. J. F. Jones, "DCU@FIRE-2014: An Information Retrieval Approach for Source Code Plagiarism Detection", in *Proceedings of the Forum for Information Retrieval Evaluation*, ser. FIRE '14, vol. 05-07-Dec-, New York, USA: ACM Press, 2015, pp. 39–42, ISBN: 978-1-4503-3755-7. DOI: 10.1145/2824864.2824887.

[59] D. Gitchell and N. Tran, "Sim: A utility for detecting similarity in computer programs", in *The proceedings of the thirtieth SIGCSE technical symposium on Computer science education*, vol. 31, New York, USA: ACM Press, 1999, pp. 266–270, ISBN: 987-1-58113-085-6. DOI: 10.1145/299649.299783.

[60] S. Grier, "A tool that detects plagiarism in Pascal programs", in *Proceedings of the twelfth SIGCSE technical symposium on Computer science education*, ser. SIGCSE '81, New York, USA: ACM Press, 1981, pp. 15–20, ISBN: 978-0-8979-1036-1. DOI: 10.1145/800037.800954.

[61] D. Grune and M. Huntjens, "Het detecteren van kopieen bij informatica-practica", *Informatie*, vol. 31, no. 11, pp. 864–867, 1989.

[62] J. Hage, "Programmeerplagiaatdetectie met Marble - Technical Report UU-CS-2006-062", Department of Information and Computing Sciences, Utrecht University, Department of Information and Computing Sciences, Utrecht University, Tech. Rep., 2006, Accessed on 2016-07-25 Available at http://www.cs.uu.nl/research/techreps/repo/CS-2006/2006-062.pdf.

[63] J. Hage, P. Rademaker, and N. van Vugt, "Plagiarism Detection for Java: A Tool Comparison", in *Computer Science Education Research Conference*, ser. CSERC '11, Heerlen, The Netherlands: Open Universiteit, Heerlen, 2011, pp. 33–46, ISBN: 978-9-0358-1987-0.

[64] J. Hage, B. Vermeer, and G. Verburg, "Research Paper: Plagiarism Detection for Haskell with Holmes", in *Proceedings of the 3rd Computer Science Education Research Conference on Computer Science Education Research*, ser. CSERC '13, Heerlen, The Netherlands: Open Universiteit, Heerlen, 2013, pp. 19–30.

[65] M. H. Halstead, "Natural laws controlling algorithm structure?", *ACM SIGPLAN Notices*, vol. 7, no. 2, pp. 19–26, 1972, ISSN: 0362-1340. DOI: 10.1145/953363.953366.

[66] M. H. Halstead, *Elements of Software Science (Operating and Programming Systems Series)*. New York, USA: Elsevier Science Inc., 1977, ISBN: 978-0-4440-0205-1.

[67] J. Hamblen, A. Parker, and S. Wachtel, "A new undergraduate computer arithmetic software laboratory", *IEEE Transactions on Education*, vol. 31, no. 3, pp. 177–180, 1988, ISSN: 0018-9359. DOI: 10.1109/13.2309.

[68] B. Hart, M. Joy, W. Smith, R. Pitt, A. Ward, W. Zhang, T. Mak, and D. White, *Sherlock Help Guide*, Department of Computer Science, University of Warwick, Accessed on 2018-09-13 Available at https://warwick.ac.uk/fac/sci/dcs/research/ias/software/sherlock/sherlock.jar, Coventry, UK, 2003.

[69] J. H. Hayes and J. Offutt, "Recognizing authors: An examination of the consistent programmer hypothesis", *Software Testing Verification and Reliability*, vol. 20, no. 4, pp. 329–356, 2010, ISSN: 0960-0833. DOI: 10.1002/stvr.412.

[70] D. O. Hebb and W. Thompson, "The social significance of animal studies", in *G. Lindzey & E. Aronson, Handbook of social psychology*, 3rd ed, New York: Random House, 1985, pp. 729–774, ISBN: 978-0-8985-9720-2.

[71] D. Heres and J. Hage, "A Quantitative Comparison of Program Plagiarism Detection Tools", in *Proceedings of the 6th Computer Science Education Research Conference*, ser. CSERC '17, New York, USA: ACM, 2017, pp. 73–82, ISBN: 978-1-4503-6338-9. DOI: 10.1145/3162087.3162101.

[72] M. J. Heron and P. Belford, "Musings on misconduct: A Practitioner Reflection on the Ethical Investigation of Plagiarism within Programming Modules", *ACM SIGCAS Computers and Society*, vol. 45, no. 3, pp. 438–444, 2016, ISSN: 0095-2737. DOI: 10.1145/2874239.2874304.

[73] L. Hertzog, *Introduction to bootstrap with applications to mixed-effect models*, R-bloggers, Accessed on 2018-12-19 Available at https://www.r-bloggers.com/introduction-to-bootstrap-with-applications-to-mixed-effect-models/, 2015.

[74] T. C. Hoad and J. Zobel, "Methods for identifying versioned and plagiarized documents", *Journal of the American Society for Information Science and Technology*, vol. 54, no. 3, pp. 203–215, 2003, ISSN: 1532-2882. DOI: 10.1002/asi.10170.

[75] C.-H. Hsiao, M. Cafarella, and S. Narayanasamy, "Using web corpus statistics for program analysis", in *Proceedings of the 2014 ACM International Conference on Object Oriented Programming Systems Languages & Applications*, vol. 49, New York, USA: ACM Press, 2014, pp. 49–65, ISBN: 978-1-4503-2585-1. DOI: 10.1145/2660193.2660226.

[76] L. Huang, S. Shi, and H. Huang, “A new method for Code Similarity Detection”, in *IEEE International Conference on Progress in Informatics and Computing*, vol. 2, Nanjing, China: IEEE, 2010, pp. 1015–1018, ISBN: 978-1-4244-6788-4. DOI: 10.1109/PIC.2010.5687856.

[77] A. Hunt and D. Thomas, *The pragmatic programmer: from journeyman to master*. Addison-Wesley Professional, 2000, ISBN: 978-0-2016-1622-4.

[78] A. Jadalla and A. Elnagar, “PDE4Java: Plagiarism Detection Engine for Java source code: a clustering approach”, *International Journal of Business Intelligence and Data Mining*, vol. 3, no. 2, p. 121, 2008, ISSN: 1743-8187. DOI: 10.1504/IJBIDM.2008.020514.

[79] J.-H. Ji, S.-H. Park, G Woo, and H.-G. Cho, “Generating pylogenetic tree of homogeneous source code in a plagiarism detection system”, *International Journal of Control, Automation and Systems*, vol. 6, no. 6, pp. 809–817, 2008, ISSN: 1598-6446.

[80] J.-H. Ji, G. Woo, and H.-G. Cho, “A Plagiarism Detection Technique for Java Program Using Bytecode Analysis”, in *Third International Conference on Convergence and Hybrid Information Technology*, vol. 1, Busan, South Korea: IEEE, 2008, pp. 1092–1098, ISBN: 978-0-7695-3407-7. DOI: 10.1109/ICCIT.2008.267.

[81] S. Jia, L. Dongsheng, L. Zhang, and C. Liu, “A Research on Plagiarism Detecting Method Based on XML Similarity and Clustering”, in *International Workshop on Internet of Things*, vol. 312 CCIS, Hohhot, China, 2012, pp. 619–626, ISBN: 978-3-6423-2426-0. DOI: 10.1007/978-3-642-32427-7_88.

[82] M. C. Johnson, C. Watson, S. Davidson, and D. Eschbach, “Gene sequence inspired design plagiarism screening”, in *Annual Conference and Exposition, "Engineering Researchs New Heights"*, Purdue University, USA, 2004, pp. 6087–6105.

[83] E. L. Jones, “Metrics Based Plagarism Monitoring”, *Journal of Computing Sciences in Colleges*, vol. 16, no. 4, pp. 253–261, 2001, ISSN: 1937-4771.

[84] I. Jonyer, P. Apiratikul, and J. Thomas, “Source code fingerprinting using graph grammar induction”, in *Recent Advances in Artifical Intelligence - Eighteenth International Florida Artificial Intelligence Research Society Conference*, Tulsa, USA, 2005, pp. 468–473, ISBN: 978-1-5773-5234-1.

[85] M. Joy and M. Luck, “Plagiarism in programming assignments”, *IEEE Transactions on Education*, vol. 42, no. 2, pp. 129–133, 1999, ISSN: 0018-9359. DOI: 10.1109/13.762946.

[86] M. Joy, G. Cosma, J. Y.-K. Yau, and J. Sinclair, “Source Code Plagiarism - A Student Perspective”, *IEEE Transactions on Education*, vol. 54, no. 1, pp. 125–132, 2011, ISSN: 0018-9359. DOI: 10.1109/TE.2010.2046664.

[87] R. I. Kabacoff, *R in Action Data analysis and graphics with R*. New York, USA: Manning Publications Co., 2011, ISBN: 978-1-93518-2399.

[88] D. C. Kar, "Detection of Plagiarism in Computer Programming Assignments", *Journal of Computing Sciences in Colleges*, vol. 15, no. 3, pp. 266–276, 2000, ISSN: 1937-4771.

[89] B Kaučič, D Sraka, M Ramsăk, and M Krašna, "Observations on plagiarism in programming courses", in *2nd International Conference on Computer Supported Education*, vol. 2, Ljubljana, Slovenia, 2010, pp. 181–184, ISBN: 978-9-8967-4023-8.

[90] R. Kaushal and A. Singh, "Automated evaluation of programming assignments", in *IEEE International Conference on Engineering Education: Innovative Practices and Future Trends*, Kottayam, India: IEEE, 2012, pp. 1–5, ISBN: 978-1-4673-2269-0. DOI: 10.1109/AICERA.2012.6306707.

[91] M. Kaya and S. A. Özel, "Integrating an online compiler and a plagiarism detection tool into the Moodle distance education system for easy assessment of programming assignments", *Computer Applications in Engineering Education*, vol. 23, no. 3, pp. 363–373, 2015, ISSN: 1061-3773. DOI: 10.1002/cae.21606.

[92] R. Kelley and B. Dooley, "The technology of cheating", in *IEEE International Symposium on Ethics in Science, Technology and Engineering*, Chicago, IL, USA: IEEE, 2014, pp. 1–4, ISBN: 978-1-4799-4992-2. DOI: 10.1109/ETHICS.2014.6893442.

[93] D. Kermek, M. Novak, and M. Kaniški, "Two years of gamification of the course — Lessons learned", in *41st International Convention on Information and Communication Technology, Electronics and Microelectronics*, Opatija, Croatia: IEEE, 2018, pp. 754–759, ISBN: 978-953-233-095-3. DOI: 10.23919/MIPRO.2018.8400140.

[94] D. Kermek, D. Strmečki, M. Novak, and M. Kaniški, "Preparation of a hybrid e-learning course for gamification", in *2016 39th International Convention on Information and Communication Technology, Electronics and Microelectronics*, IEEE, 2016, pp. 829–834, ISBN: 978-953-233-086-1. DOI: 10.1109/MIPRO.2016.7522254.

[95] D. Kermek and M. Novak, "Process Model Improvement for Source Code Plagiarism Detection in Student Programming Assignments", *Informatics in Education*, vol. 15, no. 1, pp. 103–126, 2016, ISSN: 2335-8971. DOI: 10.15388/infedu.2016.06.

[96] A. Kleiman and T. Kowaltowski, "Qualitative Analysis and Comparison of Plagiarism-Detection Systems in Student Programs - Technical Report IC-09-08", Institu to de Computação, Universidade Estadual de Campinas, Tech. Rep., 2009, Accessed on 2016-07-25 Available at http://www.ic.unicamp.br/ reltech/2009/09-08.pdf.

[97] M. Konecki, T. Orehovački, and A. Lovrenčić, "Detecting computer code plagiarism in higher education", in *31st International Conference on Information Technology Interfaces*, Dubrovnik, Croatia: IEEE, 2009, pp. 409–414, ISBN: 978-953-7138-15-8. DOI: 10.1109/ITI.2009.5196118.

[98] J. Y. Kuo and F. C. Huang, "Code analyzer for an online course management system", *Journal of Systems and Software*, vol. 83, no. 12, pp. 2478–2486, 2010, ISSN: 0164-1212. DOI: 10.1016/j.jss.2010.07.037.

[99] J. Y. Kuo, F. C. Huang, C. Hung, and L. H. Z. Yang, "The Study of Plagiarism Detection for Object-Oriented Programming", in *Sixth International Conference on Genetic and Evolutionary Computing*, Kitakushu, Japan: IEEE, 2012, pp. 188–191, ISBN: 978-1-4673-2138-9. DOI: 10.1109/ICGEC.2012.145.

[100] T. Lancaster, "Effective and Efficient Plagiarism Detection", PhD thesis, South Bank University, Accessed on 2016-07-25 Available at https://www.academia.edu/168972/Effective_and_Efficient_Plagiarism_Detection, 2003.

[101] T. Lancaster and F. Culwin, "A Comparison of Source Code Plagiarism Detection Engines", *Computer Science Education*, vol. 14, no. 2, pp. 101–112, 2004, ISSN: 0899-3408. DOI: 10.1080/08993400412331363843.

[102] Y.-J. Lee, J.-S. Lim, J.-H. Ji, H.-G. Cho, and G. Woo, "Plagiarism Detection among Source Codes using Adaptive Methods", *KSII Transactions on Internet and Information Systems*, vol. 6, no. 6, pp. 1627–1648, 2012, ISSN: 1976-7277. DOI: 10.3837/tiis.2012.06.008.

[103] A. M. Leitao, "Detection of redundant code using (RD2)-D-2", *Software quality journal*, vol. 12, no. 4, pp. 361–382, 2004, ISSN: 0963-9314. DOI: 10.1023/B:SQJO.0000039793.31052.72.

[104] B. Lesner, R. Brixtel, C. Bazin, and G. Bagan, "A novel framework to detect source code plagiarism", in *Proceedings of the 2010 ACM Symposium on Applied Computing*, New York, USA: ACM Press, 2010, pp. 57–58, ISBN: 978-1-6055-8639-7. DOI: 10.1145/1774088.1774101.

[105] X. Li and X. J. Zhong, "The Source Code Plagiarism Detection Using AST", in *International Symposium on Intelligence Information Processing and Trusted Computing*, Chengdu, China: IEEE, 2010, pp. 406–408, ISBN: 978-1-4244-8148-4. DOI: 10.1109/IPTC.2010.90.

[106] C. Liu, C. Chen, J. Han, and P. S. Yu, "GPLAG: Detection of software plagiarism by program dependence graph analysis", in *12th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, vol. 2006, University of Illinois-UC, Illinois, USA, 2006, pp. 872–881, ISBN: 978-1-5959-3339-3.

[107] X. Liu, C. Xu, and B. Ouyang, "Plagiarism Detection Algorithm for Source Code in Computer Science Education", *International Journal of Distance Education Technologies*, vol. 13, no. 4, pp. 29–39, 2015, ISSN: 1539-3100. DOI: 10.4018/IJDET.2015100102.

[108] Y. T. Liu, H. R. Zhang, T. W. Chen, and W. G. Teng, "Extending web search for online plagiarism detection", in *IEEE International Conference on Information Reuse and Integration*, Las Vegas, IL, USA: IEEE, 2007, pp. 164–169, ISBN: 978-1-4244-1499-4. DOI: 10.1109/IRI.2007.4296615.

[109] S. Mancuso, *Model-based bootstrapped ANOVA and ANCOVA*, Accessed on 2018-12-19 Available at https://sammancuso.com/2017/11/01/model-based-bootstrapped-anova-and-ancova/, 2017.

[110] S. Mann and Z. Frew, "Similarity and originality in code: Plagiarism and normal variation in student assignments", in *Conferences in Research and Practice in Information Technology Series*, vol. 52, Hobart, TAS, Australia, 2006, pp. 143–150, ISBN: 978-1-9206-8234-7.

[111] E. Marais, U. Minnaar, and D. Argles, "Plagiarism in e-Learning Systems: Identifying and Solving the Problem for Practical Assignments", in *Sixth IEEE International Conference on Advanced Learning Technologies*, Kerkrade, Netherlands: IEEE, 2006, pp. 822–824, ISBN: 0-7695-2632-2. DOI: 10.1109/ICALT.2006.1652567.

[112] L. Mariani and D. Micucci, "AuDeNTES: Automatic Detection of teNtative plagiarism according to a rEference Solution", *ACM Transactions on Computing Education*, vol. 12, no. 1, pp. 1–26, 2012, ISSN: 1946-6226. DOI: 10.1145/2133797.2133799.

[113] R. C. Martin, *Clean Architecture: A Craftsman's Guide to Software Structure and Design*, ser. Robert C. Martin Series. Pearson Education, 2017, ISBN: 978-0-1344-9432-6.

[114] R. C. Martin, *Clean code: a handbook of agile software craftsmanship*. Pearson Education, 2009, ISBN: 978-0-1323-5088-4.

[115] R. C. Martin, *The clean coder: a code of conduct for professional programmers*. Pearson Education, 2011, ISBN: 978-0-1370-8107-3.

[116] V. T. Martins, D. Fonte, P. R. Henriques, and D. da Cruz, "Plagiarism detection: A tool survey and comparison", in *3rd Symposium on Languages, Applications and Technologies*, vol. 38, Gualtar, Portugal: Schloss Dagstuhl- Leibniz-Zentrum fur Informatik GmbH, Dagstuhl Publishing, 2014, pp. 143–158, ISBN: 978-3-9398-9768-2. DOI: 10.4230/OASIcs.SLATE.2014.143.

[117] V. T. V. Martins, "Detection of Plagiarism in Software in an Academic Environment", Master's thesis, University of Minho, Accessed on 2016-07-25 Available at http://hdl.handle.net/1822/42735, 2015.

[118] V. T. Martins, P. R. Henriques, and D. da Cruz, "An AST-based Tool, Spector, for Plagiarism Detection: The Approach, Functionality, and Implementation", *Communications in Computer and Information Science*, vol. 563, pp. 153–159, 2015, ISSN: 1865-0929. DOI: 10.1007/978-3-319-27653-3_15.

[119] J. A. McCart and J. Jarman, "A technological tool to detect plagiarized projects in microsoft access", *IEEE Transactions on Education*, vol. 51, no. 2, pp. 166–173, 2008, ISSN: 0018-9359. DOI: 10.1109/TE.2007.906312.

[120] Z. Mei and L. Dongsheng, "An XML plagiarism detection algorithm for Procedural Programming Languages", in *International Conference on Educational and Information Technology*, vol. 3, Chongqing, China: IEEE, 2010, pp. V3–427–V3–431, ISBN: 978-1-4244-8033-3. DOI: 10.1109/ICEIT.2010.5608336.

[121] Z. Mei and L. Dongsheng, "An XML plagiarism detection model for C program", in *3rd International Conference on Advanced Computer Theory and Engineering*, vol. 1, Chengdu, China: IEEE, 2010, pp. V1–460–V1–464, ISBN: 978-1-4244-6539-2. DOI: 10.1109/ICACTE.2010.5578975.

[122] G. Meszaros, *xUnit test patterns: Refactoring test code*. Pearson Education, 2007, ISBN: 978-0-1314-9505-0.

[123] C. Meyer, C. Heeren, E. Shaffer, and J. Tedesco, "CoMoTo: the collaboration modeling toolkit", in *Proceedings of the 16th annual joint conference on Innovation and technology in computer science education*, New York, USA: ACM Press, 2011, p. 143, ISBN: 978-1-4503-0697-3. DOI: 10.1145/1999747.1999789.

[124] G. Milliken and D. Johnson, *Analysis of messy data Volume 1 Designed Experiments*. CRC Press, 2009, p. 690, ISBN: 978-1-5848-8334-0.

[125] O. Mirza, M. Joy, and G. Cosma, "Style Analysis for Source Code Plagiarism Detection", in *The 17th IEEE International Conference on Advanced Learning Technologies*, Timisoara, Romania: IEEE, 2017, pp. 53–61, ISBN: 978-1-5386-3870-5. DOI: 10.1109/ICALT.2017.117.

[126] M. Mišić, Z. Siustran, and J. Protić, "A comparison of software tools for plagiarism detection in programming assignments", *International Journal of Engineering Education*, vol. 32, no. 2, pp. 738–748, 2016, ISSN: 0949149X.

[127] M. J. Mišić, J. Protić, and M. V. Tomašević, "Improving source code plagiarism detection: Lessons learned", in *25th Telecommunication Forum*, Belgrade, Serbia: IEEE, 2018, pp. 1–8, ISBN: 978-1-5386-3072-3. DOI: 10.1109/TELFOR.2017.8249481.

[128] H. Mohamad Judi, S. Mohd Sallen, N. Hussin, and S. Idris, "The Use of Assignment Programming Activity Log to Study Novice Programmers' Behavior between Non-Plagiarized and Plagiarized Groups", *Information Technology Journal*, vol. 9, no. 1, pp. 98–106, 2010, ISSN: 1812-5638. DOI: 10.3923/itj.2010.98.106.

[129] C. Z. Mooney and R. D. Duval, *Bootstrapping: A Nonparametric Approach to Statistical Inference*. California, USA: SAGE Publications, Inc., 1993, ISBN: 978-0-8039-5381-9.

[130] L. Moussiades, “PDetect: A Clustering Approach for Detecting Plagiarism in Source Code Datasets”, *The Computer Journal*, vol. 48, no. 6, pp. 651–661, 2005, ISSN: 0010-4620. DOI: 10.1093/comjnl/bxh119.

[131] M. Mozgovoy, “Desktop tools for offline plagiarism detection in computer programs”, *Informatics in Education*, vol. 5, no. 1, pp. 97–112, 2006, ISSN: 1648-5831.

[132] M. Mozgovoy, “Enhancing computer-aided plagiarsim detection”, PhD thesis, University of Joensuu, Accessed on 2016-07-25 Available at http://epublications.uef.fi/pub/urn_isbn_978952219050-5/index_en.html, 2007, ISBN: 978-9-5221-9049-9.

[133] M. Mozgovoy, K. Fredriksson, D. White, M. Joy, and E. Sutinen, “Fast Plagiarism Detection System”, in *Lecture Notes in Computer Science*, ser. SPIRE’05, vol. 3772 LNCS, Berlin, Heidelberg: Springer-Verlag, 2005, pp. 267–270, ISBN: 978-3-5402-9740-6. DOI: 10.1007/11575832_30.

[134] B. Muddu, A. Asadullah, and V. Bhat, “CPDP: A robust technique for plagiarism detection in source code”, in *2013 7th International Workshop on Software Clones*, Infosys Labs., Bangalore, India: IEEE, 2013, pp. 39–45, ISBN: 978-1-4673-6445-4. DOI: 10.1109/IWSC.2013.6613041.

[135] O. Müller and S. Strickroth, “GATE - Ein System zur Verbesserung der Programmierausbildung und zur Unterstützung von Tutoren”, in *Proceedings of the First Workshop "Automatische Bewertung von Programmieraufgaben"*, Hannover, Germany: CEUR-WS, 2013.

[136] S. Narayanan and S. Simi, “Source code plagiarism detection and performance analysis using fingerprint based distance measure method”, in *7th International Conference on Computer Science & Education*, Cochin, India: IEEE, 2012, pp. 1065–1068, ISBN: 978-1-4673-0242-5. DOI: 10.1109/ICCSE.2012.6295247.

[137] S. C. Ng, S. O. Choy, and R. Kwan, “An intelligent online assessment system for programming courses”, in *Enhancing Learning Through Technology*, World Scientific Publishing Co. Pte. Ltd, 2008, pp. 217–231, ISBN: 978-9-8127-9944-9. DOI: 10.1142/9789812799456_0014.

[138] M. Novak, D. Kermek, and M. Joy, “Calibration of source-code similarity detection tools for objective comparisons”, in *41st International Convention on Information and Communication Technology, Electronics and Microelectronics*, Opatija, Croatia: IEEE, 2018, pp. 794–799, ISBN: 978-953-233-095-3. DOI: 10.23919/MIPRO.2018.8400147.

[139] M. Novak, “Review of source-code plagiarism detection in academia”, in *39th International Convention on Information and Communication Technology, Electronics and Microelectronics*, Opatija, Croatia: IEEE, 2016, pp. 796–801, ISBN: 978-953-233-086-1. DOI: 10.1109/MIPRO.2016.7522248.

[140] M. Novak, M. Joy, and D. Kermek, "Source-code similarity detection and detection tools used in academia: A systematic review", *ACM Transactions on Computing Education*, vol. 19, no. 3, 27:1–27:37, 2019, ISSN: 1946-6226. DOI: 10.1145/3313290.

[141] M. Novak and D. Kermek, "Profesional programmer: knowledge, attitude and misconceptions", in *CASE30-Razvoj poslovnih i informatičkih sustava*, Zagreb, Croatia: CASE d.o.o., 2018, pp. 12–18.

[142] M. Novak and D. Oreški, "Fuzzy knowledge-based system for calculating course difficulty based on student perception", *Computer Applications in Engineering Education*, vol. 24, no. 2, pp. 225–233, 2016, ISSN: 1061-3773. DOI: 10.1002/cae.21700.

[143] T. Ohmann and I. Rahal, "Efficient clustering-based source code plagiarism detection using PIY", *Knowledge and Information Systems*, vol. 43, no. 2, pp. 445–472, 2015, ISSN: 0219-1377. DOI: 10.1007/s10115-014-0742-2.

[144] A. Ohno and H. Murao, "A two-step in-class source code plagiarism detection method utilizing improved CM algorithm and SIM", *International Journal of Innovative Computing, Information and Control*, vol. 7, no. 8, pp. 4729–4739, 2011, ISSN: 1349-4198.

[145] C. Oprisa, G. Cabau, and A. Colesa, "From Plagiarism to Malware Detection", in *15th International Symposium on Symbolic and Numeric Algorithms for Scientific Computing*, Bitdefender, Romania: IEEE, 2013, pp. 227–234, ISBN: 978-1-4799-3036-4. DOI: 10.1109/SYNASC.2013.37.

[146] K. J. Ottenstein, "An algorithmic approach to the detection and prevention of plagiarism", *ACM SIGCSE Bulletin*, vol. 8, no. 4, pp. 30–41, 1976, ISSN: 0097-8418. DOI: 10.1145/382222.382462.

[147] A. Parker and J. Hamblen, "Computer algorithms for plagiarism detection", *IEEE Transactions on Education*, vol. 32, no. 2, pp. 94–99, 1989, ISSN: 0018-9359. DOI: 10.1109/13.28038.

[148] J. B. Peterson, *Maps of meaning: The architecture of belief*. Routledge, 2002, ISBN: 978-0-4159-2222-7.

[149] J. B. Peterson, *12 Rules for Life: An Antidote to Chaos*. Random House Canada, 2018, ISBN: 978-0-1419-8851-1.

[150] J. Y. Poon, K. Sugiyama, Y. F. Tan, and M.-Y. Kan, "Instructor-centric source code plagiarism detection and plagiarism corpus", in *Proceedings of the 17th ACM annual conference on Innovation and technology in computer science education*, New York, USA: ACM Press, 2012, p. 122, ISBN: 978-1-4503-1246-2. DOI: 10.1145/2325296.2325328.

[151] D. A. Popescu and D. Nicolae, "Determining the Similarity of Two Web Applications Using the Edit Distance", *Advances in Intelligent Systems and Computing*, vol. 356, pp. 681–690, 2016, ISSN: 2194-5357. DOI: 10.1007/978-3-319-18296-4_53.

[152] J. L. Popyack, N. Herrmann, P. Zoski, B. Char, C. Cera, and R. N. Lass, "Academic dishonesty in a high-tech environment", in *Proceedings of the 34th SIGCSE Technical Symposium on Computer Science Education*, Drexel University, Philadelphia, USA, 2003, pp. 357–358, ISBN: 978-1-58113-648-X. DOI: 10.1145/611892.611916.

[153] A. O. Portillo-Dominguez, V. Ayala-Rivera, E. Murphy, and J. Murphy, "A unified approach to automate the usage of plagiarism detection tools in programming courses", in *12th International Conference on Computer Science and Education*, Houston, TX, USA: IEEE, 2017, pp. 18–23, ISBN: 978-1-5090-2508-4. DOI: 10.1109/ICCSE.2017.8085456.

[154] L. Prechelt, G. Malpohl, and M. Philippsen, "Finding plagiarisms among a set of programs with JPlag", *Journal of Universal Computer Science*, vol. 8, no. 11, pp. 1016–1038, 2002, ISSN: 0958-695X. DOI: 10.3217/jucs-008-11-1016.

[155] D. Qiu, J. Sun, and H. Li, "Improving Similarity Measure for Java Programs Based on Optimal Matching of Control Flow Graphs", *International Journal of Software Engineering and Knowledge Engineering*, vol. 25, no. 07, pp. 1171–1197, 2015, ISSN: 0218-1940. DOI: 10.1142/S0218194015500229.

[156] F. S. Rabbani and O. Karnalim, "Detecting Source Code Plagiarism on .NET Programming Languages using Low-level Representation and Adaptive Local Alignment", *Journal of Information and Organizational Sciences*, vol. 42, no. 1, pp. 105–123, 2017, ISSN: 1846-9418. DOI: 10.31341/jios.41.1.7.

[157] C. Ragkhitwetsagul, J. Krinke, and D. Clark, "Similarity of Source Code in the Presence of Pervasive Modifications", in *IEEE 16th International Working Conference on Source Code Analysis and Manipulation*, IEEE, 2016, pp. 117–126, ISBN: 978-1-5090-3848-0. DOI: 10.1109/SCAM.2016.13.

[158] S. S. Robinson and M. L. Soffa, "An instructional aid for student programs", *ACM SIGCSE Bulletin*, vol. 12, no. 1, pp. 118–129, 1980, ISSN: 0097-8418. DOI: 10.1145/953032.804623.

[159] R. E. Roxas, N. R. Lim, and N. Bautista, "Automatic Generation of Plagiarism Detection Among Student Programs", in *7th International Conference on Information Technology Based Higher Education and Training*, Ultimo, NSW, Australia: IEEE, 2006, pp. 226–235, ISBN: 1-4244-0405-3. DOI: 10.1109/ITHET.2006.339768.

[160] R. Saikkonen, L. Malmi, and A. Korhonen, “Fully automatic assessment of programming exercises”, in *Proceedings of the Conference on Integrating Technology into Computer Science Education*, Cantenbury, UK: ACM, 2001, pp. 133–136, ISBN: 987-1-58113-330-8. DOI: 10.1145/377435.377666.

[161] J. Sant, “Code Repurposing as an Assessment Tool”, in *IEEE/ACM 37th IEEE International Conference on Software Engineering*, ser. ICSE ’15, vol. 2, Piscataway, NJ, USA: IEEE, 2015, pp. 295–298, ISBN: 978-1-4799-1934-5. DOI: 10.1109/ICSE.2015.158.

[162] S. Saxon, “Comparison of plagiarism detection techniques applied to student code”, Part II . Computer Science project, Trinity College, Cambridge, UK, Tech. Rep., 2000.

[163] S. Schleimer, D. S. Wilkerson, and A. Aiken, “Winnowing: local algorithms for document fingerprinting”, in *Proceedings of the 2003 ACM SIGMOD international conference on on Management of data*, New York, USA: ACM Press, 2003, pp. 76–85, ISBN: 158113634X. DOI: 10.1145/872757.872770.

[164] N. Shah, S. Modha, and D. Dave, “Differential Weight Based Hybrid Approach to Detect Software Plagiarism”, *Advances in Intelligent Systems and Computing*, vol. 409, pp. 645–653, 2016, ISSN: 2194-5357. DOI: 10.1007/978-981-10-0135-2_62.

[165] S. Shan, F. Guo, and J. Ren, “Similarity Detection Method Based on Assembly Language and String Matching”, *Advances in Intelligent and Soft Computing*, vol. 1, no. 148 AISC, pp. 363–367, 2012, ISSN: 1867-5662. DOI: 10.1007/978-3-642-28655-1_57.

[166] D. Silverman and A. Marvasti, *Doing Qualitative Research: A Comprehensive Guide*. SAGE Publications, 2008, ISBN: 978-1-4129-2639-3.

[167] Simon, “Academic Integrity in Non-Text Based Disciplines”, in *Handbook of Academic Integrity*, Singapore: Springer Singapore, 2016, pp. 763–782, ISBN: 978-981-287-098-8. DOI: 10.1007/978-981-287-098-8_61.

[168] Simon, B. Cook, J. Sheard, A. Carbone, and C. Johnson, “Academic integrity perceptions regarding computing assessments and essays”, in *Proceedings of the tenth annual conference on International computing education research*, ser. ICER ’14, New York, USA: ACM Press, 2014, pp. 107–114, ISBN: 978-1-4503-2755-8. DOI: 10.1145/2632320.2632342.

[169] A. Singh, G. Mangalaraj, and A. Taneja, “An approach to detecting plagiarism in spreadsheet assignments: A digital answer to digital cheating”, *Journal of Accounting Education*, vol. 29, no. 2–3, pp. 142–152, 2011, ISSN: 0748-5751. DOI: 10.1016/j.jaccedu.2012.02.002.

[170] H.-J. Song, S.-B. Park, and S. Y. Park, “Computation of Program Source Code Similarity by Composition of Parse Tree and Call Graph”, *Mathematical Problems in Engineering*, vol. 2015, pp. 1–12, 2015, ISSN: 1024-123X. DOI: 10.1155/2015/429807.

[171] D. Sraka and B. Kaučić, "Source code plagiarism", in *31st International Conference on Information Technology Interfaces*, Ljbuljana, Slovenia: IEEE, 2009, pp. 461–466, ISBN: 978-953-7138-15-8. DOI: 10.1109/ITI.2009.5196127.

[172] M. F. Tennyson and F. J. Mitropoulos, "Choosing a profile length in the SCAP method of source code authorship attribution", in *IEEE SoutheastCon 2014*, Bradley University Peoria, Illinois, USA: Institute of Electrical and Electronics Engineers Inc., 2014, ISBN: 978-1-4799-6585-4. DOI: 10.1109/SECON.2014.6950705.

[173] J. Traxler, "Plagiarism in Programming: A Review and Discussion of the Factors", in *Proceedings of the Second International Conference on Software Engineering in Higher Education II*, ser. SEHE '95, Billerica, MA, USA: Computational Mechanics, Inc., 1995, pp. 131–138, ISBN: 1-56252-309-0. DOI: 10.2495/SEHE950171.

[174] K. Ueta and H. Tominaga, "A development and application of similarity detection methods for plagiarism of online reports", in *9th International Conference on Information Technology Based Higher Education and Training*, Hayashi-cho, Takamatsu, Japan: IEEE, 2010, pp. 363–371, ISBN: 978-1-4244-4810-4. DOI: 10.1109/ITHET.2010.5480091.

[175] J. Underwood and A. Szabo, "Academic offences and e-learning: Individual propensities in cheating", *British Journal of Educational Technology*, vol. 34, no. 4, pp. 467–477, 2003, ISSN: 0007-1013. DOI: 10.1111/1467-8535.00343.

[176] University of Sydney, *The Sherlock Plagiarism Detector*, Accessed on 2016-01-26 Available at http://sydney.edu.au/engineering/it/ scilect/sherlock/.

[177] P. Vamplew and J. Dermoudy, "An Anti-Plagiarism Editor for Software Development Courses", in *7th Australasian Computing Education Conference*, vol. 42, University of Tasmania, Tasmania, Australia, 2005, pp. 83–90, ISBN: 978-1-9206-8224-8.

[178] K. L. Verco and M. J. Wise, "Plagiarism à la mode: A comparison of automated systems for detecting suspected plagiarism", *Computer Journal*, vol. 39, no. 9, pp. 749–750, 1996, ISSN: 0010-4620. DOI: 10.1093/comjnl/39.9.741.

[179] K. L. Verco and M. J. Wise, "Software for detecting suspected plagiarism: comparing structure and attribute-counting systems", in *Proceedings of the first Australasian conference on Computer science education*, New York, USA: ACM Press, 1996, pp. 81–88, ISBN: 978-0-8979-1845-9. DOI: 10.1145/369585.369598.

[180] D. Vogts, "Plagiarising of source code by novice programmers a "cry for help"?", in *Proceedings of the 2009 Annual Research Conference of the South African Institute of Computer Scientists and Information Technologists*, New York, USA: ACM Press, 2009, pp. 141–149, ISBN: 978-1-6055-8643-4. DOI: 10.1145/1632149.1632168.

[181] R. Všianský, D. Dlabolová, and T. Foltýnek, "Source Code Plagiarism Detection for PHP Language", vol. 3, no. 2, pp. 106–117, 2017. DOI: 10.11118/ejobsat.v3i2.100.

[182] C. Wang, Z. Liu, and L. Dongsheng, “Preventing and Detecting Plagiarism in Programming Course”, *International Journal of Security and Its Applications*, vol. 7, no. 5, pp. 269–278, 2013, ISSN: 1738-9976. DOI: 10.14257/ijsia.2013.7.5.25.

[183] D. Weber-Wulff, K. Köhler, and C. Möller, “Collusion Detection System Test Report 2012”, Hochschule für Technik und Wirtchaft, Berlin, Tech. Rep., 2012, Accessed on 2016-08-11 Available at http://plagiat.htw-berlin.de/collusion-test-2012/.

[184] D. Weber-Wulff, C. Möller, J. Touras, and E. Zincke, “Plagiarism Detection Software Test 2013”, Hochschule für Technik und Wirtchaft, Berlin, Tech. Rep., 2013, Accessed on 2016-07-25 Available at http://plagiat.htw-berlin.de/software-en/test2013/.

[185] G. Whale, “Identification of program similarity in large populations”, *Computer Journal*, vol. 33, no. 2, pp. 140–146, 1990, ISSN: 0010-4620. DOI: 10.1093/comjnl/33.2.140.

[186] D. R. White and M. S. Joy, “Sentence-based natural language plagiarism detection”, *ACM Journal on Educational Resources in Computing*, vol. 4, no. 4, 2004, ISSN: 1531-4278. DOI: 10.1145/1086339.1086341.

[187] M. J. Wise, “Detection of similarities in student programs: YAP’ing may be preferable to plague’ing”, in *ACM SIGCSE Bulletin*, vol. 24, Kansas City, MO, USA: ACM, 1992, pp. 268–271. DOI: 10.1145/135250.134564.

[188] M. J. Wise, “YAP3: improved detection of similarities in computer program and other texts”, in *Proceedings of the twenty-seventh SIGCSE technical symposium on Computer science education*, vol. 28, New York, USA: ACM Press, 1996, pp. 130–134, ISBN: 0-89791-757-X. DOI: 10.1145/236452.236525.

[189] D. Wollschlaeger, *Mixed-effects models for repeated-measures ANOVA*, R Examples Repository, Accessed on 2018-12-19 Available at http://dwoll.de/rexrepos/posts/anovaMixed.html, 2013.

[190] H. Xiong, H. Yan, Z. Li, and H. Li, “BUAA_AntiPlagiarism: A System To Detect Plagiarism for C Source Code”, in *International Conference on Computational Intelligence and Software Engineering*, Beijing, China: IEEE, 2009, pp. 1–5, ISBN: 978-1-4244-4507-3. DOI: 10.1109/CISE.2009.5366790.

[191] H.-B. Yang, L. Chen, and L.-F. Yang, “Study on the static analysis and the similarity comparing of SQL code”, in *2010 3rd International Conference on Advanced Computer Theory and Engineering*, vol. 6, IEEE, 2010, pp. V6–138–V6–141, ISBN: 978-1-4244-6539-2. DOI: 10.1109/ICACTE.2010.5579400.

[192] S. Yang and X. Wang, “A Visual Domain Recognition Method Based on Function Mode for Source Code Plagiarism”, in *Third International Symposium on Intelligent Information Technology and Security Informatics*, Dalian, China: IEEE, 2010, pp. 580–584, ISBN: 978-1-4244-6730-3. DOI: 10.1109/IITSI.2010.114.

[193] S. Yang, X. Wang, C. Shao, and P. Zhang, "Recognition on source codes similarity with weighted attributes eigenvector", in *International Conference on Intelligent Control and Information Processing*, Dalian, China: IEEE, 2010, pp. 539–543, ISBN: 978-1-4244-7047-1. DOI: 10.1109/ICICIP.2010.5565209.

[194] B. Zeidman, "Detecting source-code plagiarism", *Dr. Dobb's Journal*, vol. 29, no. 7, pp. 57–60, 2004, ISSN: 1044-789X.

[195] A. Zeller, "Making students read and review code", in *Proceedings of the Conference on Integrating Technology into Computer Science Education*, Helsinki, Finl: ACM, New York, USA, 2000, pp. 89–92, ISBN: 978-1-58113-207-7. DOI: 10.1145/353519.343090.

[196] D. Zhang, M. Joy, G. Cosma, R. Boyatt, J. Sinclair, and J. Yau, "Source-code plagiarism in universities: a comparative study of student perspectives in China and the UK", *Assessment & Evaluation in Higher Education*, vol. 39, no. 6, pp. 743–758, 2014, ISSN: 0260-2938. DOI: 10.1080/02602938.2013.870122.

[197] L. Zhang and L. Dongsheng, "AST-based multi-language plagiarism detection method", in *IEEE 4th International Conference on Software Engineering and Service Science*, IEEE, Hohhot, China: IEEE, 2013, pp. 738–742, ISBN: 978-1-4673-5000-6. DOI: 10.1109/ICSESS.2013.6615411.

[198] L. Zhang, Y.-t. Zhuang, and Z.-m. Yuan, "A Program Plagiarism Detection Model Based on Information Distance and Clustering", in *The 2007 International Conference on Intelligent Pervasive Computing*, Jeju City, South Korea: IEEE, 2007, pp. 431–436, ISBN: 978-0-7695-3006-2. DOI: 10.1109/IPC.2007.10.

[199] J. Zhao, G. Zhan, and J. Feng, "Disputed authorship in C program code after detection of plagiarism", in *International Conference on Computer Science and Software Engineering*, vol. 1, School Hang Zhou Normal University, Hang Zhou, China, 2008, pp. 86–89, ISBN: 978-0-7695-3336-0. DOI: 10.1109/CSSE.2008.620.

[200] H. M. Zhu, L. Zhang, W. Sun, and Y. X. Sun, "A Token Oriented Measurement Method of Source Code Similarity", *Applied Mechanics and Materials*, vol. 668-669, pp. 899–902, 2014, ISSN: 1662-7482. DOI: 10.4028/www.scientific.net/AMM.668-669.899.

[201] J. Zhu, Z. Wu, Z. Guan, and Z. Chen, "Appearance similarity evaluation for Android applications", in *Seventh International Conference on Advanced Computational Intelligence*, IEEE, 2015, pp. 323–328, ISBN: 978-1-4799-7257-9. DOI: 10.1109/ICACI.2015.7184722.

CURRICULUM VITAE

Matija Novak was born on 16^{th} March 1987 in Čakovec and currently he lives in Strahoninec. He completed his secondary education in 2005, in Technical, commercial and industrial school in Čakovec, field "Computer Technician". That same year, he enrolled the Faculty of Organization and Informatics in Varaždin at University of Zagreb. He gained Bachelor's degree: "Bachelor of Informatics" in 2008 with honorary mention, and in 2010 he gained the master degree: "Master of Informatics" with honorary mention at the same faculty.

After finishing master's degree he worked for two years in NTH Group in Varaždin, first as voice service administrator and later as Product manager for voice platform and Business Consultant for mobile applications for Swiss and German territory where he gained experience on international projects. His job included gathering of customer requirements, their specification and documentation. Later on, he worked for one year at MCS d.o.o in Strahoninec as system architect for mobile and web platforms. One of his main product was developing Cadastral web for Croatian State Geodetic Administration that provides an insight into cadastral data from all over the Republic of Croatia. During his work in these two companies, he gained deep knowledge and understanding of Web and mobile applications development and the process required for their implementation.

From November 2013 he is a PhD student at University of Zagreb - Faculty of Organization and Informatics. Matija is working as a teaching assistant at the same faculty where he is teaching courses Web design and programming, Building a Web application and Advanced Web technologies and services. He is an author of multiple of technical and scientific articles listed bellow in the field of software engineering and education.

Published Research

1. M. Novak, M. Joy, and D. Kermek, "Source-code similarity detection and detection tools used in academia: A systematic review", *ACM Transactions on Computing Education*, vol. 19, no. 3, 27:1–27:37, 2019, ISSN: 1946-6226. DOI: 10.1145/3313290
2. D. Andročec, M. Novak, and D. Oreški, "Using Semantic Web for Internet of Things Interoperability", *International Journal on Semantic Web and Information Systems*, vol. 14, no. 4, pp. 147–171, 2018, ISSN: 1552-6283. DOI: 10.4018/IJSWIS.2018100108
3. M. Novak, M. Kaniški, and D. Kermek, "Course gamification with LevelUp Plugin", in *MoodleMoot 2019*, Zagreb, Croatia, 2019
4. M. Novak and D. Kermek, "Profesional programmer: knowledge, attitude and misconceptions", in *CASE30-Razvoj poslovnih i informatičkih sustava*, Zagreb, Croatia: CASE d.o.o., 2018, pp. 12–18

5. D. Kermek, M. Novak, and M. Kaniški, "Two years of gamification of the course — Lessons learned", in *41st International Convention on Information and Communication Technology, Electronics and Microelectronics*, Opatija, Croatia: IEEE, 2018, pp. 754–759, ISBN: 978-953-233-095-3. DOI: 10.23919/MIPRO.2018.8400140

6. M. Novak, D. Kermek, and M. Joy, "Calibration of source-code similarity detection tools for objective comparisons", in *41st International Convention on Information and Communication Technology, Electronics and Microelectronics*, Opatija, Croatia: IEEE, 2018, pp. 794–799, ISBN: 978-953-233-095-3. DOI: 10.23919/MIPRO.2018.8400147

7. M. Novak, "Review of source-code plagiarism detection in academia", in *39th International Convention on Information and Communication Technology, Electronics and Microelectronics*, Opatija, Croatia: IEEE, 2016, pp. 796–801, ISBN: 978-953-233-086-1. DOI: 10.1109/MIPRO.2016.7522248

8. D. Kermek and M. Novak, "Process Model Improvement for Source Code Plagiarism Detection in Student Programming Assignments", *Informatics in Education*, vol. 15, no. 1, pp. 103–126, 2016, ISSN: 2335-8971. DOI: 10.15388/infedu.2016.06

9. D. Kermek, D. Strmečki, M. Novak, and M. Kaniški, "Preparation of a hybrid e-learning course for gamification", in *2016 39th International Convention on Information and Communication Technology, Electronics and Microelectronics*, IEEE, 2016, pp. 829–834, ISBN: 978-953-233-086-1. DOI: 10.1109/MIPRO.2016.7522254

10. D. Kermek and M. Novak, "Course gemification for programming education in Moodle", in *MoodleMoot 2016*, Zagreb, Croatia, 2016

11. M. Novak, D. Strmečki, and D. Oreški, "Linked Open Data: Realization, Trends and Application Overview", in *CEE e| Dem and e| Gov Days 2016*, Budapest, Hungary, 2016, pp. 303–314

12. D. Kermek, M. Kaniški, and M. Novak, "IoT Protocols overivew", in *CASE28-Razvoj poslovnih i informatičkih sustava*, Zagreb, Croatia: Case d.o.o, 2016, pp. 71–78

13. R. Tomašković, E. Popović, and M. Novak, "PHP7. 0-Innovations review, testing and transition", in *CASE28-Razvoj poslovnih i informatičkih sustava*, Zagreb, Croatia: Case d.o.o, 2016, pp. 27–36

14. T. Balint, D. Jakovljević, and M. Novak, "Phaser game development", in *CASE28-Razvoj poslovnih i informatičkih sustava*, Zagreb, Croatia: CASE d.o.o., 2016, pp. 21–26

15. M. Novak and I. Švogor, "Current usage of Component based Principles for Developing Web Applications with Frameworks: A Literature Review", *Interdisciplinary Description of Complex Systems*, vol. 14, no. 2, pp. 253–276, 2016, ISSN: 1334-4676. DOI: 10.7906/indecs.14.2.14

16. M. Novak and D. Oreški, "Fuzzy knowledge-based system for calculating course difficulty

based on student perception", *Computer Applications in Engineering Education*, vol. 24, no. 2, pp. 225–233, 2016, ISSN: 1061-3773. DOI: 10.1002/cae.21700

17. M. Novak, I. Magdalenić, and D. Radošević, "Common Metamodel of Component Diagram and Feature Diagram in Generative Programming", *Journal of Computer Science*, vol. 12, no. 10, pp. 517–526, 2016, ISSN: 1549-3636. DOI: 10.3844/jcssp.2016.517.526
18. M. Šestak, Martina; Rabuzin, Kornelije; Novak, "Integrity constraints in graph databases - implementation challenges", in *Central European Conference on Information and Intelligent Systems*, Varaždin, Croatia: Faculty of Organization and Informatics, University of Zagreb, 2016, pp. 23–30
19. D. Oreški, M. Konecki, and M. Novak, "Identifying impacts of social networks usage on student population", in *IAC-SSaH 2015*, Prague, Cezch Republic, 2015, pp. 252–257
20. M. Novak, "Source-code preprocess model for improved plagiarism detectionin student programming assignments", in *10th International Doctoral Seminar*, Varaždin, Croatia: Faculty of Organization and Informatics, University of Zagreb, 2015, pp. 41–44
21. M. Novak and D. Kermek, "Home Automation Using Raspberry Pi", in *Central European Conference on Information and Intelligent Systems*, Varaždin, Croatia: Faculty of Organization and Informatics, University of Zagreb, 2015, pp. 239–245, ISBN: 978-1-4673-7910-6
22. M. Novak and D. Kermek, "Internet of things with RPi and Java", in *JavaCro'15 - 4. international Java conference in Croatia*, Rovinj, Croatia, 2015
23. D. Kermek and M. Novak, "Home automation using Raspberry PI", in *CASE 27 - Razvoj poslovnih i informatičkih sustava*, Zagreb, Croatia: Case d.o.o, 2015, pp. 41–47
24. K. Rabuzin and M. Novak, "WebETL Tool–A Prototype in Action", in *The Ninth International Multi-Conference on Computing in the Global Information Technology*, Sevilja, Spain, 2014, pp. 67–71, ISBN: 978-1-6120-8346-9
25. M. Novak and K. Rabuzin, "Prototype of a Web ETL Tool", *International Journal of Advanced Computer Science and Applications*, vol. 5, no. 6, pp. 97–103, 2014, ISSN: 2156-5570. DOI: 10.14569/IJACSA.2014.050614
26. K. Rabuzin and M. Novak, "Data warehouses and ETL", in *Case 22 - Metode i alati za razvoj poslovnih i informatičkih sustava*, Zagreb, Croatia: Case d.o.o, 2010, pp. 85–89
27. M. Novak, "A tool for data extraction, transformation and load", Graduate thesis, University of Zagreb, Faculty of Organization and Informatics, 2010, p. 140
28. M. Novak, "Databases on the Web", Undergraduate thesis, University of Zagreb, Faculty of Organization and Informatics, 2008, p. 65