



UNIVERSITEIT•STELLENBOSCH•UNIVERSITY
jou kennisvennoot • your knowledge partner

A Hybrid Taxonomy & Model for Detecting, Source Code, Authorship Misconduct: Classifying Code Duplication as Self-Plagiarism

by

Christo Goosen
15857255

Socio-Informatics Honours 2014

Research Project

Department of Information Science
Stellenbosch University

Supervisor: Mr RJ Barnett

October 2014

Declaration

By submitting this report electronically, I declare that the entirety of the work contained therein is my own, original work, that I am the owner of the copyright thereof (unless to the extent explicitly otherwise stated) and that I have not previously in its entirety or in part submitted it for obtaining any qualification.

Date:

Copyright © 2014 Stellenbosch University
All rights reserved.

Abstract

This research project proposes a hybrid and single taxonomy and model for classifying and detecting plagiarism and code duplication in software source code. This was developed due to the fields of computer and information science, classifying and studying the two distinct phenomena separately. Within academic environments the previously mentioned misconduct is addressed and penalized separately, if not focused solely on plagiarism only. These two phenomena or cases of authorship misconduct are not only ethical, copy right and intellectual property issues; but also could lead to future academic or industry misconduct. Academia has a unique opportunity to not only penalized misconduct if necessary, but also educate potential programmers on best-practices and misconduct implications. This research projects identifies the problematic division and shortcomings, identified in literature, between the classification and detection of plagiarism and code duplication.. The research project tests the following hypothesis of : Does the combination and hybridization of plagiarism and code duplication, on the basis of the latter classified as self-plagiarism, improve classification and detection of misconduct; through a hybrid taxonomy and model. This hypothesis was tested by proposing new taxonomies based on existing literature, combining the new taxonomies into a single proposed taxonomy, proposing a model and testing a limited version of the model. The research project identified that a single taxonomy and modular model vastly improved the classification of the two similar phenomena, with additional classifications identifying the behaviour applied by the author. The research project additionally identified that a system could be implemented applying the taxonomy and model to detect and distinguish the type of misconduct by the author. The research project finally identifies improvements and additions to increase the sophistication of the detection, that would in turn improve the sophistication or intelligent forms of plagiarism detected.

Contents

Declaration	i
Abstract	ii
Contents	iii
List of Figures	v
List of Tables	vi
1 Introduction	1
2 Literature Review	3
2.1 Introduction	3
2.2 Plagiarism	3
2.3 Taxonomy - Plagiarism	4
2.3.1 Literal Plagiarism	5
2.3.2 Intelligent Plagiarism	5
2.4 Detection Methods - Plagiarism	6
2.5 Code Duplication	6
2.6 Taxonomy - Code Duplication	8
2.6.1 Metamodel	10
2.7 Detection Frameworks	10
2.7.1 Plagiarism Detection Frameworks	11
2.7.2 Code Duplication Frameworks	12
2.8 Authorship Identification/Forensics	13
2.9 Algorithms	14
2.9.1 Rabin-Karp Algorithm	14
2.9.2 Vector Similarity Metric	14
2.9.3 String Similarity Metric	14
2.10 Conclusion	15
3 Hybrid Plagiarism & Code Duplication Taxonomy	17
3.1 Introduction	17
3.2 Code Duplication	17
3.2.1 Within File Duplication (wfD)	18
3.2.2 Across File Duplication (afD)	19
3.2.3 Complete File Duplication (cfD)	19
3.3 Plagiarism	19
3.3.1 Literal Plagiarism (LitP)	19
3.3.2 Intelligent Plagiarism (IntelP)	20
3.4 The need for Reviewing and Combining the two taxonomies	21

3.4.1	Copy-and-Paste	21
3.4.2	New definition for code duplication as form of plagiarism	22
3.4.3	Penalties for plagiarism	22
3.4.4	Penalties for self-plagiarism	23
3.5	New proposed hybrid detection taxonomy	25
3.5.1	Similarity Function	25
4	Proposed Model	27
4.1	Introduction	27
4.2	Pre-Processing	28
4.3	Extraction	28
4.4	Storage	28
4.5	Analysis	29
4.6	Reporting	29
4.7	Conclusion	29
5	Conclusion	30
5.1	Introduction	30
5.2	Future Work	30
	Bibliography	31

List of Figures

2.1	Three plagiarism detection frameworks: (a)extrinsic, (b) intrinsic, (c)multi-lingual (Alzahrani et al., 2012)	12
3.1	Representation of a proposed Code Duplication Taxonomy. The types are based on the RoyRoy (2009) metamodel	18
3.2	Representation of a proposed Plagiarism Taxonomy	20
3.3	Representation of a proposed hybrid plagiarism & code duplication taxonomy . .	25
3.4	Representation of a original and altered source as illustration of partial use (Wattedal, 2008)	26
4.1	Representation of a proposed hybrid detection model	27

List of Tables

2.1	Detection methods and their characteristics (Alzahrani et al., 2012)	6
2.2	Table: Vector Similarity Metrics (Alzahrani et al., 2012)	15
2.3	Table: String Similarity Metrics (Alzahrani et al., 2012)	16

Chapter 1

Introduction

Plagiarism and code duplication in software source code has been studied since the 1970s, but remains a recurring issue in both academia and industry (Ramya & Venkatalakshmi, 2013). The World Wide Web, digital media and increasing access to published source code has resulted in opportunities for students, academics and developers alike to learn technical skills, enquire on forums for advice, but also commit plagiarism (Alzahrani et al., 2012). Increasing pressure in software and academic environments further encourages plagiarism and the duplication of source code, increasing the code line count and the perceived performance of individual programmers (Ducasse et al., 1999b). Evaluating a programmer by the amount of code produced not only incentivizes code duplication and plagiarism, but also increases the risk of performance issues, software bugs and security vulnerabilities (Ducasse et al., 1999b). Students are also under pressure to produce code for projects amidst other subjects and time constraints. Besides technical issues, increasing financial and legal costs arise for organizations and businesses, with large corporations using litigation as a weapon to remove competitors from the market (Judy Anderson, 1998). Businesses and the Free and Open Source Software (FOSS) community face devastating legal risks in the event that programmer negligence is identified by competitors and exploited through litigation (Al-ekram et al., 2005). The FOSS community's decentralized organizational structures and knowledge sharing poses a high-risk factor for programmer negligence and the ensuing fallout (Al-ekram et al., 2005). Additionally the recent controversy around bugs in software like Heartbleed and Shellshock highlights the dangers from programmer misconduct or negligence and the widespread impact on business critical systems (Jackson & PhishLabs, 2014).

Plagiarism and code duplication in software source code is a recurring issue within the Academic community, with a potential and devastating spill-over into the software industry. Detection and the delicate reprimanding of such infractions or mistakes by students, can be as much of an educational learning process as attending classes. Ethical and professionally educated programmers could potentially protect the software industry of software bugs, security exploits and devastating lawsuits.

Academic studies and commercial products for the last half century have primarily focused on specific or single detection methods(Ramya & Venkatalakshmi, 2013). The increasing sophistication of detection evasion techniques put these methods at risk and possibly incites the increase of plagiarism(Alzahrani et al., 2012). As the nature or threat of plagiarism changes so too must the nature of plagiarism detection. Additionally detecting code duplication within the same system allows educators to catch, discipline and educate students on best practices in programming.

Structure of research project

To analyse the state of source code analysis, this paper will first investigate literature on the subject. Much of the literature focuses on the definitions, taxonomies and frameworks applied in the field. A taxonomy is a valuable tool as plagiarism and code duplication share similarities in the behaviour of the perpetrator. With a strong taxonomy laid out this research project will investigate current methods for detection in source code and review existing systems that are widely used.

****Figure out how to add to section above** The following literature review is will begin reviewing Plagiarism as a whole, within source code and the proposed taxonomies to classify the forms of plagiarism. Code duplication as a section will review the reason for the phenomena, its repercussions, its prevalence in the software industry and a taxonomy for its classification. The next section on detection methods will review the literature and studies of the different types of detection methods and algorithms available from current and historical research. Based partly on the previously reviewed methods, a review of existing and currently used detection systems will investigate the effective application of such methods. The final section will review algorithms useful in detection, analysis and increasing the efficiency (processing time, resource usage etc.) of a detection system.

Following the investigation the paper will propose two previously proposed taxonomies and a new hybrid taxonomy and model; which would use multiple existing and proven methods instead of proposing a new singular method model. A hybrid model would combine the advantages of different methods to increase the accuracy of detection and the resistance to detection evasion techniques. The proposed model will employ multiple detection levels, providing a accumulated result and both plagiarism and code duplication detection. The model would include methods to resist evasion techniques and measures used to obfuscate plagiarism and code duplication. Finally the paper will conclude by discussing the viability of such a model and the possible application in both academia and the software industry.

Chapter 2

Literature Review

2.1 Introduction

The majority of the literature is divided amongst either plagiarism or code duplication, literature covering both topics is either non-existent or scarce. Adding to the large amount of academic literature is the study of software forensics, authorship analysis and the study of plagiarism in natural languages (Kilgour et al., 1998). Research in Software forensics and linguistics can be applied in my current research as they share methods or can contribute additional methods to the field. The wide, rich and vast nature of the literature available has the potential to be both positive and negative in the scope of the research. Most of the reviewed literature focuses on Computer Science and Information Science research in code duplication and plagiarism in source code. Some of the articles build on the previously mentioned fields of study in areas like: linguistics, Software forensics, proposed taxonomies, open source software, etc.

2.2 Plagiarism

Before further investigation into plagiarism, a general definition of plagiarism is warranted. The Association for Computing Machinery (ACM) defines plagiarism as misrepresenting words, ideas, source code or creative expression as your own (ACM, 2014). The ACM continues to identify plagiarism as an unethical act, but also the occurrence of self-plagiarism when a near verbatim copy is made of an author's own work without referencing the original source (ACM, 2014). The ACM (2014) further defines self-plagiarism as "Self plagiarism or redundant, duplicative publication (verbatim or near-verbatim reuse of significant portions of one's own copyrighted work in subsequent papers where the authors have not disclosed in the subsequent paper the previous publication)". For the purpose of this research, the ACM's definition of plagiarism will form the basis of the research and an individual engaging in plagiarism will be referred to as a plagiarist (Alzahrani et al., 2012).

Plagiarism amongst students, lecturers and researchers is on the rise and without detection and consequences; it will follow graduates into their work and professional environments (Boisvert & Irwin, 2006). Plagiarism has also become a rising issue for Journals and publishers with the ACM stating the escalation of plagiarism to annual occurrences, compared to its first 40 years of publishing (Boisvert & Irwin, 2006). A policy on plagiarism, first issued in 2005, has enabled the ACM to deal with such cases, but might not account for the increasing sophistication of evading detection (Boisvert & Irwin, 2006). The ethical dilemma of plagiarism in academia is highlighted by the following statement: “Ethical writing demands that ideas, data, and conclusions are borrowed from others and used as the foundation of one’s own contributions to the literature, must be properly acknowledged” (Alzahrani et al., 2012). Research backed by literature is supposed to build upon the existing body of knowledge and increase it, not steal from it.

Adding to the challenges of detecting plagiarism, in source code additional factors exist, including third party source code, code generators, common identifier names and commonly used algorithms (Zeidman, 2007). Amidst challenges in both natural language and source code plagiarism, an effective taxonomy of plagiarism is needed to support a system of detection (Ramya & Venkatalakshmi, 2013). The taxonomy of plagiarism is based on the behaviour of the plagiarist, which enables classification and detection (Alzahrani et al., 2012).

2.3 Taxonomy - Plagiarism

Reviewing plagiarism is a challenging exercise based on its different natures and the behaviours of application by the plagiarist. Studying the characteristics of plagiarism is best approached through a taxonomy, based on existing literature. Alzahrani et al. (2012) wrote a extensive article on understanding the patterns of plagiarism within linguistics and the detection thereof. Alzahrani et al. (2012) provides a good summary of their research in the following statement: “This paper brings patterns of plagiarism together with textual features for characterization of each pattern and computerized methods for detection”. The most notable feature of their research is the connection of the behaviours of plagiarists with a taxonomy of the various forms of plagiarism. Not only is it essential to classify the types of plagiarism, but also highlighting the human behaviour producing the plagiarism (Alzahrani et al., 2012). Adding to the behaviour factor is the factor of how plagiarism has changed over time with influences like the world wide web and technology.

Alzahrani et al. (2012) highlights the changing characteristics of plagiarism by reviewing a existing taxonomy. The existing taxonomy has two types of plagiarism of either an “exact copy” or “modified copy”, with slight variations in the two types depending on the extent of the document plagiarized and the whether or not the author has referenced (Alzahrani et al., 2012). The taxonomy proposed by Alzahrani et al. (2012) introduces the concept of “intel-

ligerent” behaviours by dividing the taxonomy into two main parts: “literal” or “intelligent” plagiarism.

2.3.1 Literal Plagiarism

Literal plagiarism is defined as “copying the text nearly as it is” by Alzahrani et al. (2012) or as the “practice of verbatim or cut and paste of the text from one document to another (might be words or ideas)” as defined by Alzahrani et al. (2011).

Literal plagiarism is identified by the “near verbatim” copy of another author’s work, or simply it is the exact copy (ACM, 2014). Literal plagiarism is common and has a high detection rate due to the lack of obscurification by the plagiarist (Ramya & Venkatalakshmi, 2013). Although the plagiarist might embed the plagiarised work within an existing structure, no proper reference or quotation has been attempted to credit the obvious original source (ACM, 2014).

2.3.2 Intelligent Plagiarism

The definition for *Intelligent Plagiarism* by Alzahrani et al. (2012) is the “illegal practices of changing text to hide the offences including restructuring, paraphrasing, summarizing, and translating”. Intelligent plagiarism is characterized separately from literal plagiarism, based on the sophistication of a plagiarist, to obscure the borrowing of another author’s source as their own (Alzahrani et al., 2012).

Intelligent plagiarism is a growing trend whereby a plagiarist deceives a reader, through modifying contributions from other authors (Alzahrani et al., 2012). Intelligent plagiarism is difficult to detect due to the differing levels of deception and obscurification used to mimic borrowed ideas as a plagiarist’s own (Alzahrani et al., 2012). Intelligent plagiarism modifies and embeds original sources through text “manipulation”, “translation” and “idea adoption” (Alzahrani et al., 2012). Intelligent plagiarism is especially effective through a knowledge in linguistics or programming language styles; applied to the original source (Alzahrani et al., 2012). A holistic view of Intelligent plagiarism describes it as the modification or “context-based adaptation” of an author’s original source (Alzahrani et al., 2012).

Within Intelligent plagiarism Alzahrani et al. (2012) classifies “text manipulation”, “translation” and “idea adoption” as methods applied by plagiarists. An interesting part of the taxonomy is within the adoption of ideas as it is the most difficult form to detect and can be applied either through “semantic based meaning”, “section based importance” or “context based adaptation” (Alzahrani et al., 2012). Idea adoption is a serious infraction of ethical guidelines and policies and plagiarists can apply it upon a entire source for use as their own.

2.4 Detection Methods - Plagiarism

The next important section from the work of Alzahrani et al. (2012), is the section on plagiarism detection methods. The methods are interesting as they are applied over both areas of textual and source code detection. Alzahrani et al. (2012) identifies the following methods for plagiarism detection: “Char-Based, Vector-Based, Syntax-Based, Semantic-Based, Fuzzy-Based, Structural-Based, Stylometric-Based and Cross-Lingual”. These methods differ in their tasks, type of plagiarism detected and language specific or cross-lingual abilities (Alzahrani et al., 2012). The following Table 2.1 depicts the various methods, their abilities and the types of plagiarism detected.

TABLE VI
PLAGIARISM DETECTION METHODS AND THEIR EFFICIENCY IN DETECTING DIFFERENT PLAGIARISM TYPES

Technique	Tasks		IR		Language(s)	Plagiarism Type(s)								Reference
	extrinsic	intrinsic	mono-lingual	cross-lingual		Literal			Intelligent					
						copy	near copy	restructuring	paraphrasing	summarising	translating	idea (section)	idea (context)	
Char-Based (CNG)	☑		☑		any	☑	☑							[1-6]
Vector-Based (VEC)	☑		☑		any	☑	☑	☑						[7-11]
Syntax-Based (SYN)	☑		☑		specific	☑	☑	☑						[6, 12, 13]
Semantic-Based (SEM)	☑		☑		specific	☑	☑	☑	☑	☐				[14, 15]
Fuzzy-Based (FUZZY)	☑		☑		specific	☑	☑	☑	☑	☐				[16-19]
Structural-Based (STRUC)	☑		☑		specific	☑	☑	☑	☐	☐		☐	☐	[21, 29]
Stylometric-Based (STYLE)		☑	☑		specific	☑	☑	☑						[22, 23, 32-35]
Cross-Lingual (CROSS)	☑			☑	cross						☑			[31, 36-38]

The notions in the table indicate the following: ☑ means include/support by evidence from research stated in the references column, ☐ means possibility to include/support but need further research for proof.

Table 2.1: Detection methods and their characteristics (Alzahrani et al., 2012)

2.5 Code Duplication

Kapser (2009) states the following in his doctoral thesis on “software code cloning”: “the intentional or unintentional introduction of similar code segments as it occurs in medium-sized to large software systems”. Code duplication or cloning in its simplest form can be described as the programmer applying “copy-and-paste” to segments of code where similar functionality is needed (Kapser, 2009). The copying and pasting of code is not the only means of duplication, as the programmer can modify libraries, file and code segments from previous code sources to fulfil similar functionality within the segment, file or directory (Kapser, 2009). Code duplication is creates concern in large software projects as it is “detrimental to the quality of the source code”, “can cause additional maintenance effort”, increase compiling time and increase both file size and code length (Kapser, 2009). Kapser (2009) states that although code duplication from bad practices and programmer’s laziness will have a run-off effect on the source code, when programmer’s “intentionally create code

clones for principled engineering reasons such as mitigating risk, supporting flexible evolution, or preventing fragmentation of concepts, then code cloning may have a positive impact on the system quality”.

Research by Kapser & Godfrey (2003b) towards identifying and classifying code duplication or software clones, was conducted by reviewing the “Linux kernel file-system subsystem”. The Linux file-system is ideal for study as it has a layered design to incorporate most of the commercially and open-source available file systems (Kapsner & Godfrey, 2003b). The subsystem’s top layer is called a “*Virtual File System (VFS)*” which enables the operating system to have a common interface for interacting with various types of file systems (Kapsner & Godfrey, 2003b). This allows the Linux Kernel to mount and work across different file systems like NTFS, ext2, ext3, ext4, HFS, HFS+ etc. through creating a root directory (Kapsner & Godfrey, 2003b). The design of the VFS lead Kapser & Godfrey (2003b) to hypothesise that code duplication would be detected, due to the various file systems needing similar functionality when interacting with the VFS. The file-system subsystem consists of the VFS and 42 implementations of the various file-system (Kapsner & Godfrey, 2003b). File systems like ext2, ext3 and ext4 have a high number of similarities and would likely have many files or functions copied across the directories and files (Kapsner & Godfrey, 2003b). Kapser & Godfrey (2003b) used tools like “CCFinder” and python scripts to identify code duplication in Linux’s mostly C or C++ source code.

Ducasse et al. (1999a) conducted research into a “A Language Independent Approach for Detecting Duplicated Code” through a visual approach, but also proposed reasoning behind programmers conducting code duplication in their work. Ducasse et al. (1999a) identified the following reasons for programmers applying such a practice: “simpler and faster than writing the code from scratch”, “Evaluating the performance of a programmer by the amount of code he or she produces gives a natural incentive for copying code” and “Efficiency considerations may make the cost of a procedure call or method invocation seem too high a price”. Except for identifying the reasons for the phenomena, Ducasse et al. (1999a) also identified that up to “70%” of software development consists of maintenance. Maintenance is further complicated by duplication in the following: “If one repairs a bug in a system with duplicated code, all possible duplications of that bug must be checked”, “Code duplication increases the size of the code, extending compile time and expanding the size of the executable”, “Code duplication often indicates design problems like missing inheritance or missing procedural abstraction” (Ducasse et al., 1999a).

Research by Roy (2009) identified the importance of code duplication detection due to “near-miss software clones”. “Near-miss” clones are copied fragments that have minor to extensive modifications to the original source adding to the need for advanced detection methods (Roy, 2009). The focus of the study was based on “open source systems” and that “software engineering tasks, such as program understanding (clones may carry domain knowledge), code quality analysis (fewer clones may mean better quality code), aspect mining

(clones may indicate the presence of an aspect), plagiarism detection, copyright infringement investigation, software evolution analysis, code compaction (for example, in mobile devices), virus detection, and bug detection may require the extraction of syntactically or semantically similar code fragments” (Roy, 2009). Based on research and a case study in open source software Roy (2009) propose a metamodel and framework for effective code duplication detection.

2.6 Taxonomy - Code Duplication

Code duplication or software clones, is a field of research without an generally accepted or proposed taxonomy (Kapsner & Godfrey, 2003a). Although a major a major problem in the software industry, educating best practice in coding instead of detecting duplication, is the recurring trend within academia (Kapsner & Godfrey, 2003b). Most of the literature encountered classified code duplication in a general classification, with others proposing the shift towards a taxonomy (Kapsner & Godfrey, 2003b). Such a proposal was made by Kapsner & Godfrey (2003a) and in a recent article the importance of such a taxonomy was highlighted: ”It is our position that generating a taxonomy of clones will provide several important contributions to clone detection research”.By using metrics and different detection tools Kapsner & Godfrey (2003b) was able to identify a taxonomy based on the different types of duplication detected. Kapsner & Godfrey (2003b) used tools like ”CCFinder” to tokenize the source files in the Linux file-system subsystem and quantify such characteristics like line count, levels of nesting and parameters used or modified. Their taxonomy allows the standardisation of research in the code duplication and allows accurate detection tools to be modelled based on their taxonomy of the different kinds of code duplication.

The taxonomy proposed by Kapsner & Godfrey (2003b) is based on their Linux file-system case study, but depicts recurring practice in the software industry. Linux has become a widely used kernel for many Unix-based distributions and the size of the kernel alone provides a good case study for code duplication detection (Kapsner & Godfrey, 2003b). The taxonomy proposed by Kapsner & Godfrey (2003b) is called a Taxonomy of Clones and is divided into 7 different clone types namely: ”Duplicated blocks within same function”, ”Similar functions, same file”, ”Functions cloned between files within the same directory”, ”Functions cloned across directories”, ”Cloned files”, ”Blocks across files” and ”Initialization and finalization clones”.

”*Duplicated blocks within same function*” was the first classification identified by Kapsner & Godfrey (2003b) when reviewing the results from their Linux kernel case study. Kapsner & Godfrey (2003b) identifies this type of code duplication as ”repeated blocks of code within the same function, these blocks are of non-trivial size (such as 5 to 127 lines of code) and each copy expresses the same semantic idea”. It was identified that generally a few of the variables were changed, but most of the original source was maintained(Kapsner & Godfrey,

2003b). The type of code duplication is prevalent in the file-system subsystem and amongst programmers usually copying the structure for *"IF"*, *"SWITCH"*, etc. statements (Kapser & Godfrey, 2003b).

The second classification is called *"Similar functions, same file"*. Kapser & Godfrey (2003b) classifies this through identifying two or more functions with similar tasks or functionality and only slight modification from the original source. As stated by Kapser & Godfrey (2003b) : "These types of clones are often characterized by changing only a few function calls, variable initializations, constants, or other minor things".

The third classification *"Functions cloned between files within the same directory"* is identified when multiple files require the same functionality (Kapser & Godfrey, 2003b). The files are located within the same directory and the programmer will distribute the functions to different files, with minor or no modifications, instead of writing a commonly shared library (Kapser & Godfrey, 2003b). A common library is especially effective in reducing code size when exact copies of functions are used across files (Kapser & Godfrey, 2003b).

Similar to the previous classification, the fourth classification occurs across multiple directories and is called: *"Functions cloned across directories"* (Kapser & Godfrey, 2003b). Commonly in this practice, the programmer will apply no to minor modifications to the functions (Kapser & Godfrey, 2003b). This is prevalent in such "tasks as parsing options or outputting errors" and when a new subsystem is added based on the "design and implementations" of another similar subsystem (Kapser & Godfrey, 2003b). Kapser & Godfrey (2003b) identified this behaviour in the file-system subsystem in situations like implementing ext4 from similar ext3 design and functionalities.

"Cloned files" is the fifth classification identified by Kapser & Godfrey (2003b) and is an interesting behaviour prevalent in the Linux kernel case study. Cases identified of the classification are similar to the ext4 and ext3 subsystems sharing various functions or parts (Kapser & Godfrey, 2003b). Kapser & Godfrey (2003b) identifies the behaviour as a "type of clone occurs when a new problem arises with requirements that are very similar to those of an existing software system, and the source code is readily available". With the source files open, the implementation of ext3 was based on the previous ext2 file-system and copied files like *"buffer.c"* in its implementation (Kapser & Godfrey, 2003b).

The second last classification of *"Blocks Across"* has a similar characteristic to the first, but "occurs in different files within the same directories or across file systems" (Kapser & Godfrey, 2003b). Kapser & Godfrey (2003b) also state that with "cloning blocks across directories, we see that the cloned block is in fact the remains of what appears to be a cloned function". The programmer modifies the copied function according to a personal or preferred style and the specific needs of the project or file in which it will be utilised (Kapser & Godfrey, 2003b).

The last classification proposed by Kapser & Godfrey (2003b) is the *"Initialization and finalization clones"*. Kapser & Godfrey (2003b) defines these types of clones as occurring "across file systems when initializing data parameters or cleaning up at end of function" and that it "usually occurs when using the same data types or when performing the same tasks such as memory allocation and de-allocation or variable initialization". With finalization clones specifically, they usually contain "exit conditions and logging" that the programmer copied from another source (Kapser & Godfrey, 2003b).

2.6.1 Metamodel

The metamodel proposed by Roy (2009) is based on two definitions namely "code fragment" and "software clone". A code fragment consists of a code line sequence and a software clone is the pairing or classing of code fragments based on a "given definition of similarity" (Roy, 2009). A software clone is represented as " $f(CF1) = f(CF2)$ " where f represents the "similarity function" and CF a code fragment. Referring to more than two code fragments it could be represented as $f(CF1) = f(CFn)$. These two definitions are utilized within four types of software clones to form the Roy (2009) metamodel. The metamodel, as proposed by Roy (2009), is as follows:

Type 1 "Identical code fragments except for variations in whitespace, layout and comments."

Type 2 "Syntactically identical fragments except for variations in identifiers, literals, types, whitespace, layout and comments."

Type 3 "Copied fragments with further modifications such as changed, added or removed statements, in addition to variations in identifiers, literals, types, whitespace, layout and comments."

Type 4 "Two or more code fragments that perform the same computation but are implemented by different syntactic variants."

These four types proposed by Roy (2009) effectively classify different code duplication situations and the specific characteristics of each.

2.7 Detection Frameworks

Frameworks are essential for the analysis of source code. Frameworks allow the breakdown of the process to smaller, logical processes to be completed in an ordered fashion. As emphasized by Chudá et al. (2011) using methods like "text pre-processing" can have an impact on the result of analysis on the suspected text.

2.7.1 Plagiarism Detection Frameworks

Based on the proposed taxonomy for plagiarism, Alzahrani et al. (2012) reviews the aspects of a "Plagiarism detection systems frameworks based" on different types of tasks during detection. The need for a frameworks is based on the limitations of human detection and the vast amount of information needed to compare suspected sources(Alzahrani et al., 2012). The use of frameworks are necessary as most existing and commonly used detection systems are only reviewed within a "Black-box" framework(Alzahrani et al., 2012). The first part of the framework review is focused on the tasks applied to the detection system. The first task of extrinsic plagiarism detection refers to the detection system evaluating the suspected source to a single or multiple additional sources(Alzahrani et al., 2012). When applying detection to plagiarism though extrinsic analysis, the sources necessary are either procured from within a local source or an external source like the World Wide Web, for computer based analysis(Alzahrani et al., 2012). An intrinsic analysis would evaluate the document in isolation and is an area in which human detection can identify variations in the style of writing(Alzahrani et al., 2012). When applied to a computerised system, the author's style can be quantified and questionable authorship can be verified(Alzahrani et al., 2012). Another part of the frameworks reviewed is characterised by language dependence and the differing "monolingual" and "cross-lingual" detection systems(Alzahrani et al., 2012). Fig 2.1. represents the main frameworks presented by Alzahrani et al. (2012).

Lesner et al. (2010) conducted a case study to measure "similarity distance" between two documents. The similarity distance was achieved by transforming a document into smaller parts and calculating cost of changing one document to another through a "document-wise based distance" (Lesner et al., 2010). In the context of their case study a document relates to the source code of a student's computer science project. By calculating the cost of changing the segmented parts to that of the other document Lesner et al. (2010) was able to calculate the similarity.

The proposed method or framework by Lesner et al. (2010) followed a "bottoms-up" approach with each next step reaching a new level of abstraction. The method is built on the following six steps: "pre-filtering, segmentation and similarity measurement stage, segment matching, post- filtering, document-wise distance evaluation, and corpus analysis presentation". The pre-filtering stage is especially interesting in its approach to transform the original text into a easily quantifiable measure (Lesner et al., 2010). The pre-filtering stage achieves this through the following approach: "replacing each alphanumerical string (eg. variable name or keywords) by a single symbol" (Lesner et al., 2010). The pre-filtering stage relates to the work of Heintze (1996), using hashing and fingerprinting to prepare documents for similarity analysis.

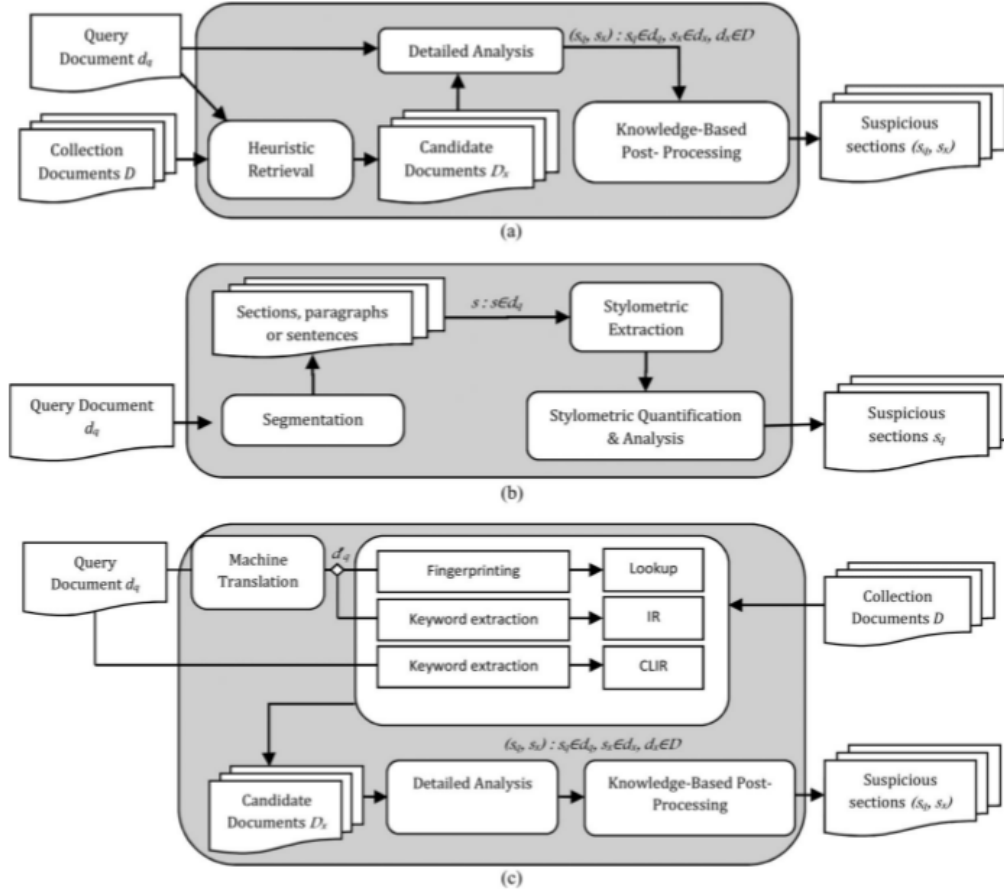


Figure 2.1: Three plagiarism detection frameworks: (a) extrinsic, (b) intrinsic, (c) multi-lingual (Alzahrani et al., 2012)

2.7.2 Code Duplication Frameworks

Ducasse et al. (1999b) proposes a language independent code duplication framework or approach. To achieve the language independent approach Ducasse et al. (1999b) uses the following three methods: “algorithms used to compute the comparisons data, visualization of the comparison data, and pattern matching to condense the data”. Before the comparison data can be produced by algorithms, the document is prepared to remove code comments and white spaces (Ducasse et al., 1999b). The condensing of files not only allows the algorithms to effectively process the data, but also increases efficiency (Ducasse et al., 1999b). Following the creation of comparison data, the data is visualised and reports created based on the results of the analysis (Ducasse et al., 1999b).

2.8 Authorship Identification/Forensics

Authorship analysis of source code has become an important area of research as in areas such as legal, educational and malicious software (Kilgour et al., 1998). Kilgour et al. (1998) highlights the importance of authorship analysis through stating that it has "promising applications in both the legal sector (such as proof of ownership and software forensics) and the education sector (such as plagiarism detection and assessing style)". Gray et al. (1997) categorizes the over all field as "software forensics" which "involves the areas of author discrimination, identification, and characterisation, as well as intent analysis". Authorship identification is built on the concept that each author of software source code have differing style or preferences when writing code and analysis of the code can produce a type of author fingerprint(Gray et al., 1997).

Frantzeskou et al. (2006) wrote an article on "Effective identification of source code authors using byte-level information" motivated by issues such as "authorship disputes, proof of authorship in court, cyber attacks". Their study is based on previous work called: "IDENTIFIED"(Gray et al., 1998). IDENTIFIED or "Integrated Dictionary-based Extraction of Non-language-dependent Token Information for Forensic Identification, Examination, and Discrimination", is a system designed to extract metrics from source code for authorship identification(Gray et al., 1998). In their study Frantzeskou et al. (2006) used the "SCAP (Source Code Author Profiles) approach, which is an extension of a method that has been successfully applied to text authorship identification". The approach focuses on text and is based on It "byte level n-grams and the utilization of a similarity measure used to classify a program to an author"(Frantzeskou et al., 2006). The method and metrics used in the study made the analysis language independent and through experimentation studied factors such as multiple authors and the role of comments(Frantzeskou et al., 2006).

Ding & Samadzadeh (2004) studied the extraction of authorship fingerprints from Java programs by posing the following question: "Is it possible to extract something like human DNA finger prints from software to verify authorship of this kind of written text?". Ding & Samadzadeh (2004) focused on measuring the "likelihood of a particular author having written some piece(s) of code" through extracting a type of programming fingerprint from source code. The study identified that up to 67.2% of analysed Java programs could be accurately identified according to author. Their system and research is based on the previously mentioned "IDENTIFIED" system and similarly extracting metrics from the source code, but differs in focusing on the characteristics of the Java language syntax(Ding & Samadzadeh, 2004).

2.9 Algorithms

Add an introduction here

2.9.1 Rabin-Karp Algorithm

The Rabin-Karp algorithm "pattern-matching" algorithm popularly used for text identification, but can be used for matching other kinds of data (Caloyannides et al., 2009). The Rabin-Karp algorithm is effective for low-memory matching by hashing each string to a unique identifier before being compared Kytjoki et al. (2003). As stated by the authors of the Rabin-Karp algorithm, it "represent strings of length n by much shorter strings called fingerprints, and achieve their efficiency by manipulating fingerprints instead of longer strings" and the "algorithms require a constant number of storage locations, and essentially run in real time" (Karp & Rabin, 1987). The algorithm is easy to program and incorporate, uses less memory and as stated before "essentially" runs in "real time" (Karp & Rabin, 1987).

2.9.2 Vector Similarity Metric

Alzahrani et al. (2012) used "fingerprint and hash-based" "heuristics retrieval models for handling the large amount of string data. Fingerprinting is the selection of "subsequences of characters from a document" to be hashed and used as a unique identifier or fingerprint for the document (Heintze, 1996). Fingerprinting is in essence a hashing algorithm used more specifically on specific or selected sub-strings. Using these fingerprints or hashes Alzahrani et al. (2012) divided documents into smaller unique units or vectors, with these vectors compared and used to mark similar documents. Alzahrani et al. (2012) applied these unique vectors by "using one of the vector similarity metrics" as described in Table 2.2.

2.9.3 String Similarity Metric

Alzahrani et al. (2012) explains the various methods of string matching in their research when discussing "character-based methods" for plagiarism detection. "Exact" or "approximate" string matching can be applied to either match to exact characters or measure the distance between two characters (Alzahrani et al., 2012). Different similarity metrics can be applied as show in Table 2.3.

Vector Similarity Metric	Description & Equation	Equation	Range	Example	Ref.
Matching coefficient	-similar to Hamming distance but between vectors of equal length.	$M(x, y) = x - x \cap y $	0 to $ x $ Where $ x = y $	$x=[0.1, 0.2, 0.3, 0.4]$ $y=[0.1, 0.2, 0.3, 0.5]$ $M(x, y) = 1$	[11]
Jaccard (or Tanimoto) coefficient	-defines number of shared terms against total number of terms. This measure is computed to one if vectors are identical.	$J(x, y) = \frac{ x \cap y }{ x \cup y }$	0 to 1	$J(x, y) = 3/5 = 0.6$	[3, 7, 8, 21]
Dice's coefficient	-similar to Jaccard but reduces the effect of shared terms between vectors. This measure is computed to two if vectors are identical.	$D(x, y) = \frac{2 x \cap y }{ x \cup y }$	0 to 2	$D(x, y) = 6/5 = 1.2$	-
Overlap (or containment) coefficient	-if v_1 is subset of v_2 or the converse, then the similarity coefficient is a full match.	$O(x, y) = \frac{ x \cap y }{\min(x , y)}$	0 to 1	$O(x, y) = 3/4 = 0.75$ (or 75%)	[10]
Cosine coefficient	-finds the cosine angle between two vectors.	$Cos(x, y) = \frac{\sum_i (x_i, y_i)}{\sqrt{\sum_i (x_i)^2} \sqrt{\sum_i (y_i)^2}}$	0 to 1	$Cos(x, y) = 0.34/0.3421$ $= 0.9939 \approx 1$	[9, 21, 26, 28]
Euclidean distance	-measures the geometric distance between two vectors.	$Ec(x, y) = \sqrt{\sum_i x_i - y_i ^2}$	0 to ∞	$Ec(x, y) = 0.1$	-
Squared Euclidean Distance	-places progressively greater weight on vectors that are further apart	$SEc(x, y) = \sum_i (x_i - y_i)^2$	0 to ∞	$SEc(x, y) = 0.01$	-
Manhattan Distance	-measures the average difference across dimensions and yields results similar to the simple Euclidean distance	$Manh(x, y) = \sum_i x_i - y_i $	0 to ∞	$Manh(x, y) = 0.1$	-

An example of similarity evaluation of two vectors $x = [0.1, 0.2, 0.3, 0.4]$ and $y = [0.1, 0.2, 0.3, 0.5]$.

Table 2.2: Table: Vector Similarity Metrics (Alzahrani et al., 2012)

2.10 Conclusion

Although code duplication and plagiarism has been studied for almost 50 years, increasing research is needed to formalise the taxonomies. Additionally new and hybrid methods need to be researched for detection systems to stay effective against modern detection evasion techniques. Based on the presented research a proposed taxonomy for plagiarism and code duplication will be presented. Build on the structure of the taxonomies a hybrid method of detection will be proposed for testing with controlled and external data sets. The viability of the proposed taxonomies and hybrid model/framework will be based on the effective detection rate of the entire system.

Similarity Metric	Description	Examples	Ref.
Hamming distance (also edit distance)	defines number of characters different between two strings x and y of equal length. - allows only substitutions, cost 1	$x = "aaabbbccc"$ $y = "aaabbbccd"$ $d(x,y)=1$	-
Levenshtein distance (also edit distance)	defines minimum edit distance which transforms x into y . Edit operations include: -delete a char, cost 1 -insert a char, cost 1 -substitute one char for another, cost 1	$x = "aaabbbccc"$ $y = "aaabbbccd"$ $z = "aaabbbccdf"$ $w = "aaabbbcc"$ $d(x,y)=1$ $d(x,z)=2$ $d(x,w)=1$	[4, 5]
Longest Common Sequence (LCS) distance	measures the length of the longest pairing of chars that can be made between x and y with respect to the order of the chars. -allows insertions, cost 1 -allows deletions, cost 1	$x = "aaabbbccc"$ $y = "aaabbbccd"$ $d(x,y)=8$	[6, 28]

An example of similarity evaluation of different strings $x = "aaabbbccc"$, $y = "aaabbbccd"$, $z = "aaabbbccdf"$ $w = "aaabbbcc"$.

Table 2.3: Table: String Similarity Metrics (Alzahrani et al., 2012)

Chapter 3

Hybrid Plagiarism & Code Duplication Taxonomy

3.1 Introduction

Within the reviewed literature a lack of consensus on taxonomies was identified. The field of code duplication had significant historical research, but only by 2003 was the need for a taxonomy identified by Kapser and Godfrey(Kapser & Godfrey, 2003b). Taxonomies for plagiarism have had significantly more research than code duplication, but the focus of the research has been mostly on linguistics and natural languages(Alzahrani et al., 2012). Building on aspects and strengths from the reviewed taxonomies, this research project proposes revised and condensed taxonomies for code duplication and plagiarism. Although these taxonomies serve the purpose of detection in source code, the taxonomy for plagiarism can be applied in various other fields of study for future research.

*****Add movement to single taxonomy

3.2 Code Duplication

Although the case-based taxonomy of Kapser & Godfrey (2003b) and metamodel of Roy (2009) is effective, for the purpose of this research project a more condensed and comprehensive taxonomy was necessary (Kapser & Godfrey, 2003b). Some of the classifications or categories in the taxonomy seem similar, which leads me to propose a condensed taxonomy of the following: Within File Duplication, Across Files Duplication and Complete File Duplication. The seven classifications proposed by Kapser & Godfrey (2003b) form part of this the taxonomy for this thesis, but groups them according to the how many files were involved and whether part or the entire file was copied. The condensed taxonomy also allows the grouping of categories nearly identical in their classification. Adding the Roy (2009) Code

Duplication metamodel, to the condensed taxonomy will not only identify and classify the extent of duplication in a project, but also be able to classify the similarity types of the duplication. The similarity function is inherited and adapted from the metamodel discussed in the ‘Literature Review’, to form the basis of and demonstrate the different types of duplication. A proposed new code duplication taxonomy, could have a similarity function with Code Fragments (CF) and File Identifiers (FI):

$$f(FI1) = f(FIn)$$

$$f(FIn) = f(CF1) = f(CFn)$$

The proposed taxonomy has the following main classifications: *within file duplication (wfD)*; *across file duplication (afD)* and *complete file duplication (cfD)*. The taxonomy further classifies *wfD* and *afD* with five subcategories based on the metamodel proposed by Roy(Roy, 2009). The first four additional subcategories account for modification behaviour and the final subcategory accounts for a exact clone of a code fragment. A representation of the proposed Taxonomy in Figure 3.1.

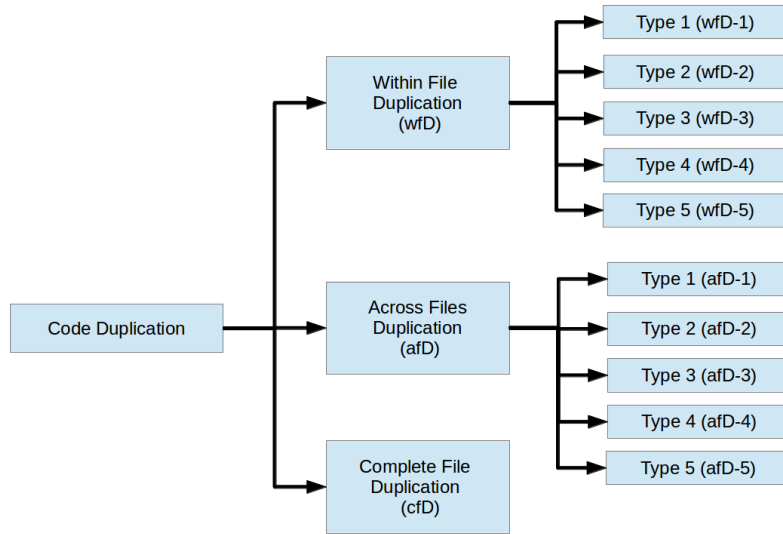


Figure 3.1: Representation of a proposed Code Duplication Taxonomy. The types are based on the RoyRoy (2009) metamodel

3.2.1 Within File Duplication (wfD)

The classification of code duplication within the same file includes all types of duplication including individual lines of code, functions, comments etc. The classification focuses on the occurrence of duplication within a single reviewed file, with additional classifications of

five types: wfD-1, wfD-2, wfD-3, wfD-4 and wfD-5. To account for the similarities between code fragments within a single file the following similarity function is proposed.

$$f(CF_1, CF_n) = 1$$

3.2.2 Across File Duplication (afD)

The classification of code duplication across multiple files includes the same types and behaviours as in wFD, but the duplication occurs across two or more different files. The same five types apply and are classified as: afD-1, afD-2, afD-3, afD-4 and afD-5. A similarity function for multiple files accounts for both different files and different code fragments is presented below.

$$f[FI_1(CF_1), FI_2(CF_1, CF_2)] = 1$$

3.2.3 Complete File Duplication (cfD)

The final classification of complete file duplication or file clone has no subcategories, as the despite manner of duplication the same result is produced as the source file. This classification applies to exact cloning of a file that resulting in the exact same file size, code length, comments etc. This is the simplest type of code duplication to classify with the following similarity function.

$$f(FI_1, FI_2) = 1$$

3.3 Plagiarism

For the purpose of this research the following plagiarism taxonomy is proposed with two main classifications: Literal Plagiarism (LitP) and Intelligent Plagiarism (IntelP). The proposed taxonomy is based on the work of Alzahrani et al. (2012) and thus focuses on the behaviour or degree of obscurification. Literal Plagiarism has no degree of obscurification, where Intelligent Plagiarism has differing degrees of obscurification. The proposed taxonomy is visually represented Figure 3.2.

3.3.1 Literal Plagiarism (LitP)

Within Literal Plagiarism two additional classifications are *Exact copy* and *Partial Copy*, in reference of the extent to which a near verbatim copy is made. This is classified as either a exact copy of the entire file or parts of the file. The cloned nature of the detected file or fragment is presented by the following two functions. The first function represents plagiarized files and the second plagiarized code fragments in a single file.

$$\begin{aligned} F_1 &= F_2 \\ CF_1 &= CF_2 \end{aligned}$$

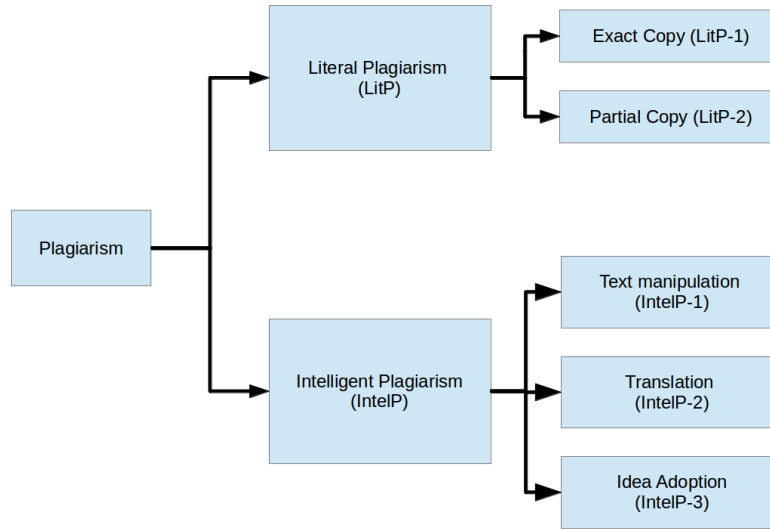


Figure 3.2: Representation of a proposed Plagiarism Taxonomy

3.3.2 Intelligent Plagiarism (IntelP)

Within Intelligent Plagiarism the following three classifications are: *Text manipulation (IntelP-1)*, *Translation (IntelP-2)* and *Idea Adoption (IntelP-3)*. All three classifications are characterized by a degree of obscurification applied by the author.

The first sub-classification of text manipulation includes all modifications of the the following nature: changing variable/functions/class names, deleting sections (like comments), re-ordering, etc. The taxonomy by Alzahrani et al. (2012) accounted for a modified copy in literal plagiarism, but modification proposes intelligent behaviour on the part of the plagiarist. Modification of information from a source to obscure its original source is classified as intelligent behaviour according to this proposed taxonomy.

The second sub-classification of translation is not only focused on the translation of information through human languages. Software source code can also be subject to translation through compiling, decompiling, code generators or adapting source code from one programming language to another. Using differing cycles of compiling and decompiling source code a programmer can obscure the original source code through the built in optimization in compilers.

Idea adoption is a difficult behaviour to classify. Essentially the programmer views existing code for a function and writes similar functionality in a personalised or modified manner. This type of plagiarism can be classified and detected through “dependence graph analysis” (Liu et al., 2006). Although the code fragment could be syntactically different, the logical flow of data and different dependencies of the fragment remain the same (Liu et al.,

2006).

Intelligent plagiarism classifications can be represented by similar functions as Within File Duplication (wFD) and Across File Duplication (aFD). These functions differ from literal plagiarism as they measure a degree of similarity. The two functions, for single or multiple files, are the following.

$$f(CF_1, CF_n) = 1$$

$$f[F_1(CF_1), F_2(CF_1, CF_2)] = 1$$

3.4 The need for Reviewing and Combining the two taxonomies

The two, above mentioned, proposed models for taxonomy and code duplication were effective in their respective fields, but did not improve the proposed hybrid model of detection. This was further supported by reviews, on a summary paper for the South African Institute for Computer Scientists and Information Technologists (SAICSIT) conference. The paper submitted for review to SAICSIT, but not accepted, was based on this research project (Matthee & Gelderblom, 2014). The reviewers for the SAICSIT conference highlighted that the proposed taxonomies improved on research in the respective fields of plagiarism and code duplication, but would have to be combined to support hybrid detection (Matthee & Gelderblom, 2014). A reviewer identified that code duplication shared similarities to literal plagiarism, providing a direction for the hybrid taxonomy (Matthee & Gelderblom, 2014). Based on the reviews and short-comings of the previous taxonomies, further research was necessary to identify the similarities and possible combination into a hybrid taxonomy.

3.4.1 Copy-and-Paste

Reviewing the literature from previous sections, code duplication or "software code cloning" by definition is the act of "copy-and-paste" by the author to clone functionality and save time or effort (Kapser, 2009). A more simple definition could be the re-use of existing code in subsequent work, which is comparable to the following statement: "Self-plagiarism occurs when authors reuse portions of their previous writings in subsequent research papers" (Collberg & Kobourov, 2005). Roy & Cordy (2007) further defend this point through the statement: "Clones are often the result of copy-paste activities...such activities are very easy and can significantly reduce programming and time as they reuse an existing fragment of code rather than rewriting similar code from scratch". The link between code duplication and self-plagiarism can be made on the previous statements and the five classifications for plagiarism provided by Ali et al. (2011):

1. "Copy & Paste Plagiarism"

2. “Word Switch Plagiarism”
3. “Style Plagiarism”
4. “Metaphor Plagiarism”
5. “Idea Plagiarism”

The focus of this section is copy and paste or literal plagiarism, so the previously discussed classifications from Ali et al. (2011)’s 2-4 will not form the basis of this argument. The copy and paste behaviour is the closest match to that between self-plagiarism and code duplication.

3.4.2 New definition for code duplication as form of plagiarism

Thus code duplication can be classified within the definition of Ali et al. (2011) and Roy & Cordy (2007) as, “copy & paste plagiarism”, but more specifically as copy and paste self-plagiarism. To further classify code duplication as plagiarism, it is necessary to review the ACM (2014)’s definitions of literal plagiarism and self-plagiarism; as it forms the basis of this research project. Further reviewing the previous ‘Literature Review’ section, self-plagiarism is defined when “a near verbatim copy is made of an author’s own work without referencing the original source” (ACM, 2014). From the ‘Literature review’ literal plagiarism is classified as “Verbatim copying” a portion or significant portion of another author’s work (ACM, 2014). Therefore in the context of code duplication as a form of literal plagiarism the following can be inferred that code duplication is the copy-paste behaviour of an authors’ own work within a new project or section of the same project.

By combining code duplication and plagiarism based on the similarity of the former and as a form of the latter, the viability of a single hybrid detection method increases. It is important to note that although similar, code duplication and literal plagiarism should be identified and classified differently based on the differing offences and disciplinary actions that each constitutes.

As this research project uses the ACM (2014) definitions for plagiarism as a basis, it is appropriate to base penalties on those imposed by the same organisation. The ACM (2014) takes action on incidents of plagiarism based on the type of plagiarism identified, categorized as that of another author or self-plagiarism.

3.4.3 Penalties for plagiarism

The ACM (2014) has strict penalties on the detection of an author plagiarizing another author. Three types of plagiarism are identified by the ACM (2014) within plagiarising other authors; namely not citing a large portion, not citing sentences or elements like images and lastly citing without applying quotation marks correctly. Although each classification

differs, they share common penalties or courses of action. The following example states the ACM (2014) penalties of the first type of plagiarism, some of which the other two types share:

1. “ACM will inform the Department Chair, Dean, or supervisor of the authors of the finding of plagiarism.”
2. “The authors will be asked to write a formal letter of apology to the authors of the plagiarized paper, including an admission of plagiarism.”
3. “If the paper has appeared in press, ACM will post a Notice of Plagiarism based on the investigation on the ACM Digital Library’s citation page of the plagiarizing paper and will remove access to the full text. The paper itself will be kept in the database for future research or legal purposes.”
4. “If the paper is under submission, the paper can be automatically rejected by the Editor-in-Chief or the Program Chair without further revisions and without any further plagiarism investigation coordinated by the ACM Director of Publications. In addition, a letter of warning will be sent by the Editor-in-Chief or the Program Chair to the authors with a copy of the ACM Policy and Procedures on Plagiarism.”

3.4.4 Penalties for self-plagiarism

The ACM (2014) classifies its final type of plagiarism as “Self plagiarism or redundant, duplicative publication (verbatim or near-verbatim reuse of significant portions of one’s own copyrighted work in subsequent papers where the authors have not disclosed in the subsequent paper the previous publication)”. Although the penalties are not as many listed in the types involved with plagiarising another author, the ACM (2014) still states the seriousness of the infringement. Self-plagiarism shares the same two penalties as the two last points in the above mentioned first type(ACM, 2014). The reasoning for the ACM being strict on self-plagiarism is clarified by Samuelson (1994a) is that they wish to “guard against multiple publications of the same material by the same author” and their reputation of publishing new and interesting ideas to the specific field.

The issue of self-plagiarism is controversial and blurred based on the intellectual property and copy right debate (Scanlon, 2007). The intellectual property and copyright ownership is based in which capacity or organization the source has been published or whether the author retains ownership(Scanlon, 2007). Scanlon (2007) discusses the lack of consensus on the topic with the following statements: “Duplicate publication of an entire article will probably always constitute copyright infringement, unless the author owns the copyright, in which case the practice remains an egregious case of intellectual fraud...However, as a legal matter, self-plagiarism as an act of copyright infringement, specifically as related to text recycling, is an area more gray than black-and-white. Case law on copyright also

provides little guidance as very few cases have been brought against self-plagiarists; and even fewer have been successful (Scanlon, 2007). Copy right infringement is a clear violation when an author republishes similar or the same publication in two separate journals, but as Scanlon (2007) states: “anyone who self-plagiarizes from work published in a professional journal when writing for another such publication, labor for which the author is not monetarily compensated in either case would likely be shielded from publishers claims of copyright infringement. Scanlon (2007) continues to explain how this is applicable under “fair use determined by: “four factors: (1) the purpose of the use, (2) the nature of the copyrighted work, (3) the amount used, and (4) the impact on potential markets for the copyrighted work. Scanlon (2007) further identifies, the need for handling self-plagiarism separately or the need for further investigation, by discussing the authors use of existing knowledge or “baseline knowledge to understand findings or research in new avenues of study. Based on the need for baseline knowledge Scanlon (2007) investigates a reasonable amount of reuse and refers to Samuelson (1994b) in the statement : “anecdotal evidence of a “30% rule,” a rule of thumb that one may reuse as much as 30% of one article in another. Although still a serious academic misconduct, the author borrows work from himself and does not steal intellectual property from another; and might only infringe on copy right based on the organization (Samuelson, 1994a). Based on this self-plagiarism and plagiarism should be dealt with and categorized as separate, while also reviewing the extent to which it is a copy or “knowledge baseline” (Scanlon, 2007). The detection and classification of plagiarism and self-plagiarism should be separate as each institution could react to the two types of misconduct differently (Scanlon, 2007). It is important to note that both plagiarism and self-plagiarism share the same consequences based on a lack of author response to accusations (ACM, 2014). Although the nature of the infraction is different, the response (or lack of response) from the author can have a similar outcome to that of the other forms of plagiarism.

What further differentiates this type of plagiarism, especially within programming, is the different behaviour leading to the infraction. It is important to differentiate on this fact, as the reasoning can be influenced by education/training, using a previously developed framework or the Development Environment’s Macros, commands or code snippets functionalities (Baral et al., 2006). Code duplication can occur from programmer error or learnt behaviour, but borrows your own effort and intellectual property, instead of plagiarism’s deliberate use of another author’s intellectual property. Programming syntax and education also leads to duplicated code fragments that does not constitute a deliberate misconduct. The language, environment and educational aspects of programming languages constitute a possible large amount of false-positives compared to academic plagiarism. The issue of self-plagiarism is more complicated than in academic papers, thus detection of such would warrant further investigation by an individual.

3.5 New proposed hybrid detection taxonomy

Based on the review of literature and the previously proposed taxonomies, the following new hybrid plagiarism and code duplication taxonomy, is proposed in Figure 3.3. The taxonomy is largely based on the previously proposed plagiarism taxonomy, with code duplication added under ‘Literal Plagiarism’; based on the above mentioned argument. The taxonomy differentiates code duplication and literal plagiarism based on the extent of the plagiarism or duplication. Literal Plagiarism is identified as the abbreviation ‘LitP’ and Code Duplication is identified as the abbreviation ‘CodeD’.

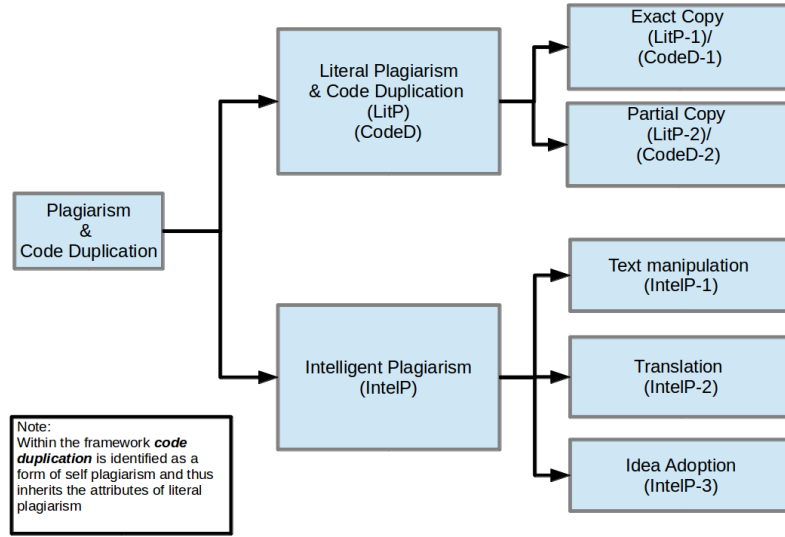


Figure 3.3: Representation of a proposed hybrid plagiarism & code duplication taxonomy

3.5.1 Similarity Function

The similarity functions for the new proposed hybrid taxonomy will be inherited from the previously proposed taxonomies, as follows:

$$f(CF_1, CF_n) = 1 \quad (3.1)$$

The similarity function for the hybrid taxonomy does not rely on the ‘Within File Duplication (wfD)’, ‘Across File Duplication (afD)’ and ‘Complete File Duplication (cFD)’ for the simplification of applying the taxonomy to a model and detection system. The taxonomy is based on the assumption that either a file is plagiarised (or copied) in its entirety or partially. Referring to Linux Kernel case study by Kapser & Godfrey (2003b), code duplication was identified either as an entire file as a code fragment or two code fragments in different files.

Within the taxonomy literal plagiarising or cloning an entire file would be classified as ‘LitP-1’ or ‘CodeD-1’; and partially would be ‘LitP-2’ or ‘CodeD-2’. For intelligent plagiarism the extent of the misconduct is asserted as partial as the author has altered or added original content to the original source. An author applying intelligent behaviour to plagiarism would alter the structure of the source to avert detection by traditional means. This is illustrated by the following Figure 3.4



Figure 3.4: Representation of a original and altered source as illustration of partial use (Watnedal, 2008)

Chapter 4

Proposed Model

4.1 Introduction

For the development of code duplication and plagiarism detection, a hybrid model is proposed. Figure 4.1 represents the full system capable of detecting plagiarism and code duplication. The model is based on five basic steps namely: pre-processing; extraction; storage; analysis, and reporting. Although these five basic steps constitute the core of the model, the model allows for a modular design. Additional steps or processes can be added to increase functionality or the effectiveness of the model.

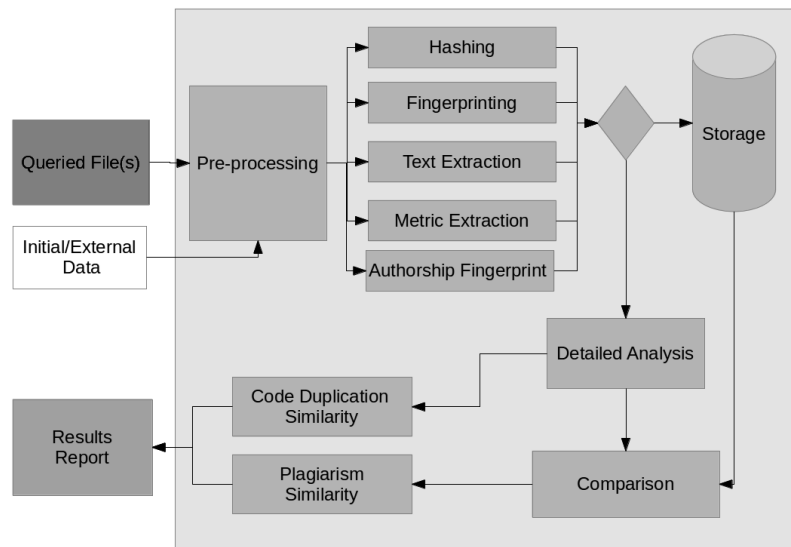


Figure 4.1: Representation of a proposed hybrid detection model

4.2 Pre-Processing

Pre-processing is an essential first step to transform the raw data from various sources for better analysis and storage. Pre-processing allows the system to identify source type (programming language, raw text, etc.), metadata and additional information from source files. During pre-processing large files are transformed into condensed strings for further processing. Processing transforms documents to a system-readable format, before continuing through further steps.

Pre-processing has an additional function except preparing suspected source documents. Pre-processing allows the processing of source documents from external documents that will provide test and similarity data.

4.3 Extraction

Extraction allows the retrieval of comments; the hashing of code fragments; fingerprinting of sections; metric extraction and a authorship profile. All of the data produced during final processing is stored in efficient and condensed formats including hashes and fingerprints. Extraction aims to retrieve all data of importance and discard unnecessary information.

Hashing allows the entire document to be hashed and compared to previously queried or mined data. Comparing hashes through the Rabin-Karp algorithm is a efficient and vast comparison that minimizes systems resource usage(Karp & Rabin, 1987). Except for hashing the entire document, hashing small sections of the source document or file can be utilized as source fingerprints. These fingerprints assist analysis and can be stored for future similarity comparison. Hashing and fingerprinting is additionally utilized to create a authorship profile of the source file. Each author's unique coding or writing style can be fingerprinted through fingerprinting certain metrics.

The complete extraction of data, metrics and fingerprints are utilized in following steps of the model. The strength of the model is the modular design of the extraction step. Removing or adding algorithms and methods, allows the model to remain current and adapt to the specific environment in which it is applied. Additionally the different types of extraction allows for multiple measurements of a suspected source file, instead of relying on a single type.

4.4 Storage

The storage step allows the model to store mined data for future similarity comparisons. Within academia the use of the model allows the storage of multiple classes and time periods, to compared beyond a given assignment. Storage additionally allows a lecturer to store data based on assignment require code shared with student. The storage of such data allows

the exclusion of false negatives based on assignment shared code.

During storage a unique code for each reviewed source file or document is stored with retrieved identifying data (name, student ID, etc.). Additional to the file identifier, a code fragment identifier will allow the pinpointing of the occurrence. This unique code identifies the various entries of data, metrics, hashes and fingerprints stored. Based on the unique code or identifier a similarity positive result can return a result based on the original source.

The information stored can be supplemented through mining sources from the internet, web or previous assignments. The supplementary data gives the model additional data with which to compare similarities of suspected or queried source files.

4.5 Analysis

The analysis step of the model will apply the taxonomies and similarity functions proposed for this research. Detection systems are developed for the purpose of analysing queried sources and this proposed model will be able to analyse both internally. The applied taxonomies and methods differ for plagiarism and code duplication; and thus are processed separately. Both use the variations of similarity functions applicable to the situation and produce a similarity analysis for each. Through reviewing literature, this proposed model is unique in its detection of both plagiarism and code duplication from the same source file, within the same system or model.

4.6 Reporting

The reports produced by the model will identify the similarity measures and the unique identifier for the queried source. Details of the author will be queried from the storage system separately, to allow a human observer to verify the results when necessary. The model or system itself cannot identify and notify a user or disciplinary system of suspected plagiarism and code duplication, directly.

4.7 Conclusion

***Tie back to taxonomy

Chapter 5

Conclusion

5.1 Introduction

5.2 Future Work

Bibliography

- ACM (2014). ACM Policy and Procedures on Plagiarism.
URL http://www.acm.org/publications/policies/plagiarism_policy
- Al-ekram, R., Kapser, C., Holt, R., & Godfrey, M. (2005). Cloning by Accident : An Empirical Study of Source Code Cloning Across Software Systems. *00(c)*, 376–385.
- Ali, A. M. E. T., Abdulla, H. M. D., & Snasel, V. (2011). Overview and Comparison of Plagiarism Detection Tools Detection Tools. (pp. 161–172).
- Alzahrani, S., Salim, N., Abraham, A., & Palade, V. (2011). iPlag: Intelligent Plagiarism Reasoner in scientific publications. *2011 World Congress on Information and Communication Technologies*, (pp. 1–6).
URL <http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=6141191>
- Alzahrani, S. M., Salim, N., & Abraham, A. (2012). Understanding Plagiarism Linguistic Patterns, Textual Features, and Detection Methods. *IEEE Transactions on Systems, Man, and Cybernetics, Part C (Applications and Reviews)*, *42*(2), 133–149.
URL <http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=5766764>
- Baral, C., Dzifcak, J., & Takahashi, H. (2006). Macros, macro calls and use of ensembles in modular answer set programming. In S. Etalle, & M. Truszczyski (Eds.) *Logic Programming*, vol. 4079 of *Lecture Notes in Computer Science*, (pp. 376–390). Springer Berlin Heidelberg.
- URL http://dx.doi.org/10.1007/11799573_28
- Boisvert, R. F., & Irwin, M. J. (2006). Plagiarism on the rise. *Communications of the ACM*, *49*(6), 23.
URL <http://portal.acm.org/citation.cfm?doid=1132469.1132487>
- Caloyannides, M., Memon, N., & Venema, W. (2009). Digital forensics. *Security Privacy, IEEE*, *7*(2), 16–17.
- Chudá, D., Chlpek, J., & Kumor, A. (2011). The impact of text pre-processing to determine the similarity in students assignments. In *Proceedings of the 12th International Conference on Computer Systems and Technologies*, CompSysTech '11, (pp. 417–422). New York, NY, USA: ACM.
URL <http://doi.acm.org/10.1145/2023607.2023677>

- Collberg, C., & Kobourov, S. (2005). Self-plagiarism in computer science. *Communications of the ACM*, 48(4).
URL <http://dl.acm.org/citation.cfm?id=1053293>
- Ding, H., & Samadzadeh, M. (2004). Extraction of Java program fingerprints for software authorship identification. *Journal of Systems and Software*, 72, 49–57.
URL <http://www.sciencedirect.com/science/article/pii/S0164121203000499>
- Ducassee, S., Rieger, M., & Demeyer, S. (1999a). A language independent approach for detecting duplicated code. ... *Maintenance, 1999.(ICSM)*
URL http://ieeexplore.ieee.org/xpls/abs/_all.jsp?arnumber=792593
- Ducassee, S., Rieger, M., & Demeyer, S. (1999b). A language independent approach for detecting duplicated code. (pp. 109–118). IEEE.
- Frantzeskou, G., Stamatatos, E., Gritzalis, S., & Katsikas, S. (2006). Effective identification of source code authors using byte-level information. *Proceeding of the 28th international conference on Software engineering - ICSE '06*, (p. 893).
URL <http://portal.acm.org/citation.cfm?doid=1134285.1134445>
- Gray, A., Sallis, P., & MacDonell, S. (1997). Software forensics: Extending authorship analysis techniques to computer programs. (97).
URL <http://otago.ourarchive.ac.nz/handle/10523/872>
- Gray, A., Sallis, P., & MacDonell, S. (1998). IDENTIFIED (Integrated Dictionary-based Extraction of Non-language-dependent Token Information for Forensic Identification, Examination, and Discrimination): a dictionary-based system for extracting source code metrics for software forensics. In *Software Engineering: Education and Practice, International Conference*, (pp. 252–259).
- Heintze, N. (1996). Scalable document fingerprinting. In *IN PROC. USENIX WORKSHOP ON ELECTRONIC COMMERCE*.
- Jackson, D., & PhishLabs (2014). Mitigating the Impact of Shellshock on Banking Operations. Tech. rep.
URL <http://www.icba.org/files/Bancard/PDFs/ShellshockImplications-PhishLabsGuidanceforFinancialSector.pdf>
- Judy Anderson (1998). *Plagiarism, Copyright Violation, and Other Thefts of Intellectual Property: An Annotated Bibliography*. London: McFarland & Company.
URL http://books.google.co.za/books?id=08EqxXsYv3EC&pg=PR3&&dq=examples+of+industry+software+plagiarism&source=gbs_selected_pages&cad=3\#v=onepage&q=examplesofindustrysoftwareplagiarism&f=false
- Kapser, C., & Godfrey, M. (2003a). A taxonomy of clones in source code: The re-engineers most wanted list. ... *on Detection of Software Clones* (... , (pp. 1–2).
URL <http://plg2.cs.uwaterloo.ca/~migod/papers/2003/wcre03-cloning-posnPaper.pdf>

- Kapser, C., & Godfrey, M. W. (2003b). Toward A Taxonomy of Clones in Source Code: A Case Study. (pp. 67–78).
- Kapser, C. J. (2009). Toward an Understanding of Software Code Cloning as a Development Practice.
- Karp, R., & Rabin, M. (1987). Efficient randomized pattern-matching algorithms. *IBM Journal of Research and ...*, 31(2), 249–260.
URL <http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=5390135>
http://ieeexplore.ieee.org/xpls/abs/_all.jsp?arnumber=5390135
- Kilgour, R. I., Gray, A. R., Sallis, P. J., & Macdonell, S. G. (1998). A Fuzzy Logic Approach to Computer Software Source Code Authorship Analysis. (pp. 865–868).
- Kytjoki, J., Salmela, L., & Tarhio, J. (2003). Tuning string matching for huge pattern sets. 2676, 211–224.
URL http://dx.doi.org/10.1007/3-540-44888-8_16
- Lesner, B., Brixtel, R., Bazin, C., & Bagan, G. (2010). A novel framework to detect source code plagiarism: now, students have to work for real! ... of the 2010 ACM Symposium on ..., (pp. 57–58).
URL <http://dl.acm.org/citation.cfm?id=1774101>
- Liu, C., Chen, C., Han, J., & Yu, P. (2006). GPLAG: detection of software plagiarism by program dependence graph analysis. *Proceedings of the 12th ACM SIGKDD ...*, (pp. 872–881).
URL <http://dl.acm.org/citation.cfm?id=1150522>
- Matthee, M., & Gelderblom, H. (2014). SAICSIT2014 notification for paper 92.
- Ramya, L., & Venkatalakshmi, R. (2013). Intelligent Plagiarism Detection. *IJREAT International Journal of Research in Engineering & Advanced Technology*, 1(1), 2–5.
- Roy, C., & Cordy, J. (2007). A survey on software clone detection research. *SCHOOL OF COMPUTING TR 2007-541, QUEENS UNIVERSITY*, (115).
URL <http://scholar.google.com/scholar?hl=en&btnG=Search&q=intitle:A+Survey+on+Software+Clone+Detection+Research+\#0http://scholar.google.com/scholar?hl=en&btnG=Search&q=intitle:A+survey+on+software+clone+detection+research\#0>
- Roy, C. K. (2009). Detection and analysis of near-miss software clones. *2009 IEEE International Conference on Software Maintenance*, (pp. 447–450).
URL <http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=5306301>
- Samuelson, P. (1994a). Self-plagiarism or fair use. *Communications of the ACM*, (pp. 21–25).
URL <http://dl.acm.org/citation.cfm?id=179731>

Samuelson, P. (1994b). Self-plagiarism or fair use. *Commun. ACM*, 37(8), 21–25.

URL <http://doi.acm.org/10.1145/179606.179731>

Scanlon, P. (2007). Song from myself: an anatomy of self-plagiarism. *Plagiary: Cross-Disciplinary Studies in Plagiarism*, ..., (pp. 57–66).

URL <http://quod.lib.umich.edu/p/plag/5240451.0002.007?rgn=main;view=fulltext>

Watnedal, T. (2008). Search and replace a line in a file in Python - Stackoverflow.

URL <http://stackoverflow.com/questions/39086/search-and-replace-a-line-in-a-file-in-python>

Zeidman, B. (2007). What, Exactly, Is Software Plagiarism? BY. *Intellectual Property Today*, (p. 17).