Style Analysis for Source Code Plagiarism Detection

by

Olfat Meraj Mirza

Thesis

Submitted to the University of Warwick

for the degree of

Doctor of Philosophy in Computer Science

Department of Computer Science

November 2018

THE UNIVERSITY OF WARWICK

Contents

List of Tables

List of Figures

Acknowledgments

Starting my own PhD project, conducting the required work, and finally producing this thesis would not have been possible without the continuous support, consistent guidance, and the assistance I have received from various generous individuals.

I would like to express my deep and sincere appreciation to all the people who have supported me and contributed to the work done in this thesis. I primarily wish to acknowledge the ongoing support, contributions, and encouragement I have received from my academic supervisor Prof. Mike Joy, who has accepted me, guided me, and enlightened my way throughout the years to produce this project. It has been a privilege to work with him.

I am deeply grateful and thankful to my second supervisor Dr. Georgina Cosma from Nottingham Trent University for all of the continuous help, the useful critiques during the planning of the project, and for the valuable, insightful and constructive suggestions toward the progress of my work.

My special thanks are extended to my advisors Dr. Jane Sinclair from the University of Warwick, and Prof. Alexandra Cristea from Durham University for their consistent advice, unlimited support, and for their generous time given during the development of this research. Additionally, I would like to thank my committee members Professor Meurig Beynon and Professor Anne James.

The supportive environment of the School of Computer Science at the University of Warwick and particularly the Human Centered Computing Research Group is greatly appreciated; the assistance I received from my colleagues and members of the research group, the staff, and from former and current students has been without any doubts invaluable and indispensable. Also, the wise and clear advices I have received from other research group members.

I would like to express my sincere gratitude to my lovely friend Ebtesam Abdulhaleem for her generous time to review the final draft and encouragement throughout the project.

Last but not least, I would like to express my deep gratitude and gratefulness for the unconditional love and support from my parents, Meraj and Sanaa and for all the confidence they had in me; the support from my sisters Israa, Ula, Iman, Alzahraa and my brother Mohammad; the logical contributions from my brothers in law Yasser and Majed; and finally for the ongoing and unconditional support of my husband Mohamed, whom I cherish his advice, time and countless contributions; and my son Suleiman, who has added the joy to my life to continue working on this thesis.

To my moon, my star and my son **Suleiman**

To the place I call home, my city **Makkah**

Declarations

This thesis is presented in accordance with the regulations for the degree of Doctor of Philosophy. It has been composed by myself and has not been submitted in any previous application for any degree. The work in this thesis has been undertaken by myself under the supervision of Prof. Mike Joy and Dr. Georgina Cosma. Some parts of this thesis are written based on previously published papers (as first author). Detail of all publications are described below.

- **Mirza, O.** and Joy, M. "Style Analysis for Source Code Plagiarism Detection" International Conference Plagiarism across Europe and Beyond 2015 Proceedings, Brno, 2015, pp. 53-61.

- **Mirza, O.**, Joy, M. and Cosma, G. "Style Analysis for Source Code Plagiarism DetectionAn Analysis of a Dataset of Student Coursework" 2017 IEEE 17th International Conference on Advanced Learning Technologies (ICALT), Timisoara, 2017, pp. 296-297.

- **Mirza, O.**, Joy, M. and Cosma, G. 'Suitability of BlackBox dataset for style analysis in source codece code plagiarism" 2017 Seventh International Conference on Innovative Computing Technology (INTECH), Luton, 2017, pp. 90-94.

Abstract

The enormous growth in the available online code resources has created new challenges for detecting plagiarism in source code of programs. Several software applications can detect source code similarity using different detection methods. However, few current detection tools detect every kind of detection plagiarism attack. The aim of this thesis is, therefore, to enhance methods for plagiarism detection in source code using a style analysis approach that has been used to detect authorship.

There are very few large source-code datasets which are suitable for research purposes, and two such datasets include the BlackBox dataset and the SOCO (Detection of SOurce COde) dataset. SOCO is a benchmark dataset that contains groups of similar source-code files that can be considered plagiarised and has been used in authorship and plagiarism detection competitions.

In the first part of the thesis, the suitability of BlackBox as source of datasets for testing plagiarism detection is explored. The files in BlackBox were analysed and visualised in order to evaluate its suitability as a dataset that can be used in this research. The analysis aimed to identify similar source code files, and therefore to detect groups of Java files within BlackBox that can be used for evaluating the performance of source-code plagiarism detection methods.

In the second part of the thesis, a plagiarism detection framework ("the Metric-File Matrix Framework (MFM)" is proposed. The MFM framework is designed to overcome some of the limitations of existing plagiarism detection methods by 1) proposing a new set of metrics which consider structural and stylistic similarities; and 2) by using Singular Value Decomposition as a technique to remove noise and to reduce the dimensionality of the data to enhance the similarity detection.

The MFM framework was implemented and its performance was evaluated using the proposed metrics. For the evaluations, the SOCO dataset was adopted and the performance of the proposed framework was compared against other state-of-the-art plagiarism detection tools including JPlag.

Chapter 1

Introduction

1.1 Introduction

Technological advances have changed our lifestyle and the way we seek information, and we have become more reliant on computers, the Internet and web search engines to find answers and seek more information about almost everything. This in turn has made us more dependent and reliant on these facilities. With this accessibility and ease of finding information online, many users can fall into the trap of plagiarism.

Plagiarism in academic work is when someone fails to acknowledge others' ideas or words, which can constitute misconduct and a breach of research integrity [39]. The computer programming community across the world is facing the increasing problem of plagiarism [64]. This widespread problem has motivated efforts to find an efficient, robust and fast detection procedure, which is difficult to achieve manually. Therefore, plagiarism detection tools have been developed. These tools focus mainly on two contexts: text [151] and source code [100]. Most text detection tools are well defined [62]. However, detection tools for source code still have problems in defining similarities between source code files [119]. Source code similarity detection entails identifying "*[a] program that has been produced from another program with a small number of routine transformations*" [118]. The main focus of this thesis is on source code plagiarism detection.

1.2 Research Background

Source code plagiarism detection in programming languages differs [61] from other types of plagiarism and the detection process is sophisticated [81]. In the context of programming files, the task of source code plagiarism detection can use tools like JPlag [125] and MOSS [18]. There are two main approaches for source code plagiarism detection: (1) attribute counting and (2) a structure-based approach. "Attribute Counting" [93] detects general document plagiarism by measuring and representing textual similarity.

The "Structure Based" approach [77] detects characteristics of the source code file [118]. For example, Song et al. [135] propose a novel method to compute how similar two program source codes are by adopting convolution kernel function as the similarity measure. Tufano et al. [141] used another technique where they demonstrated how Software Engineering (SE) would benefit from a Deep Learning (DL) based approach which can effectively replace the manual features and automatically learn code from a different representation. Ajmal et al. [2] and Barbosa et al. [9] focused on performing string matching using the Euclidean Distance formula to detect the level of similarity.

Other researchers combined the source code detection approaches. Karnalim and Sulistiani [83] combined two approaches: Attribute Based Approach (ABA) and Structure Based Approach (SBA) to identify source code plagiarism similarity, using a case study including Vector Space Model (VSM) and Cosine similarity as the ABA approaches and Running Karp-Rabin Matching and Greedy String Tiling algorithm [148] as the SBA approach.

1.3 Statement of the Problem and Motivation

Much research has studied source code plagiarism detection. However, there is a lack of research which focuses on “style analysis” source code similarity detection. The current study addresses this gap in plagiarism detection by implementing new style analysis metrics in source code plagiarism detection.

The research described in this thesis aims to complement current detection tools which have only limited functionality for detecting plagiarism. In particular, the thesis focuses on bridging the gap between coding style analysis in Java programming and plagiarism detection. The reason for using style analysis, and not the other techniques, is that most programmers leave a fingerprint in their written code. Detecting the style of the source code is very important in identifying similarity between files and suspected plagiarism without identifying the authorship.

1.4 Research Questions and Objectives

This research aims to investigate coding style analysis in source code plagiarism detection using Singular Value Decomposition (SVD) to find the similarity between files in a large dataset.

In order to meet this aim, this research answers one overarching question (RQ0), subdivided into three related ones (RQ1-3), shown below with relevant objectives (OB1-9) as follows:

RQ0: How can coding style analysis detect plagiarism effectively?

Sub-questions:

- **RQ1:** Can the BlackBox dataset be used as source of files suitable for testing source code plagiarism detection techniques?
  - **OB1**) To review the literature in the field of source code plagiarism.
  - **OB2**) To characterise the structure of the BlackBox dataset and the challenges faced in identifying good samples for further research into plagiarism detection.
  - **OB3**) To identify and analyse data using a visual approach for file similarity.
- **RQ2:** To what extent are style metrics suitable for detecting plagiarism in source code? What modifications need to be made to style metrics to enhance their potential for source code plagiarism detection?
  - **OB4**) To re-implement coding style metrics for structure based approaches to source code analysis.
  - **OB5**) To modify existing coding style metrics to be compatible with the Java programming language.
  - **OB6**) To propose new modified and extended metrics to enhance the output from the re-implementation in OB4.
- **RQ3**: How can we best capture the underlying style similarity between files using statistical approaches? Is Singular Value Decomposition a suitable technique to be embedded into a source code plagiarism detection framework?

- **OB7**) To create a source code similarity detection framework which combines style metrics with SVD.
- **OB8**) To evaluate the proposed framework using a large benchmark dataset.
- **OB9**) To compare the performance of the proposed framework against other plagiarism detection methods.

1.5 Research Methodology

In order to answer the study research questions, the methodology is presented in the following three stages. (1) selection of the dataset, (2) implementation of coding style metrics and (3) outcomes analysis and evaluation.

The focus of the first stage is the employment of large datasets to detect plagiarism in source code. In this thesis, two datasets were used: BlackBox dataset, which is a collection of BlueJ, and SOCO (source codece COde @FIRE conference 2014). The second stage is to re-implement the structure based approach to source code similarity detection with modified and extended metrics. Then SVD was used and merged with the structure-based approach. The results of the study were then compared with other studies and conclusions drawn, in the light of study limitations.

1.6 Research Rationale and Contributions

In addition to addressing the gap in the literature cited in section 1.3, this study is significant in its contribution because it has involved new empirical studies of the BlackBox and SOCO datasets. BlackBox is a collection of submission files in the online Java programming learning software called BlueJ [22]. SOCO (source codece COde) is from the Forum for Information Retrieval Evaluation (FIRE) [54].

The first contribution of this research is to demonstrate the limitations of BlackBox dataset as a source of samples suitable for testing plagiarism detection techniques.

To develop better detection methods for source code files, this study's second main contribution is to adapt coding style metrics [44], usually used to identify authorship [95], to address file similarity and identification of plagiarism in source code. Accordingly, the study re-implemented existing style metrics devised by Ding et al [44] in their structure based approach to detecting source code similarity. Due to the identified limitations in existing style metrics, two new families of coding style metrics were proposed: the Modified and the Extended families.

The third research contribution is to propose the metric-file matrix ('MFM') framework to enable coding style metrics to be analysed using Singular Value Decomposition (SVD) [13], generalising the well-established term-document matrix ('TDM') approach to document analysis. The merits of the MFM framework are demonstrated by identifying similarities between files in the SOCO dataset [53] [54].

1.7 Thesis Structure

As shown in Figure 1.1, the three main sections of the thesis reflect the three stages that were followed in order to answer the research questions.

1. The Research Background (Chapter 2): In this section, the research background and the literature are reviewed. The focus of this section is on meeting research objective OB1, to cover relevant existing detection methods, techniques and coding style metrics, and to understand the limitations of existing methods and how they need to be improved.

2. The Research Problem (Chapters 3, 4, 5 and 6): These sections discuss three research questions: RQ1, RQ2 and RQ3, as addressed in chapter 3, chapter 4

and chapters 5 and 6, respectively.

3. The Research Outcomes (Chapters 3, 4, 5 and 6): These sections present the outcomes relating to research objectives OB2-OB6. Chapter 7 presents the conclusions.

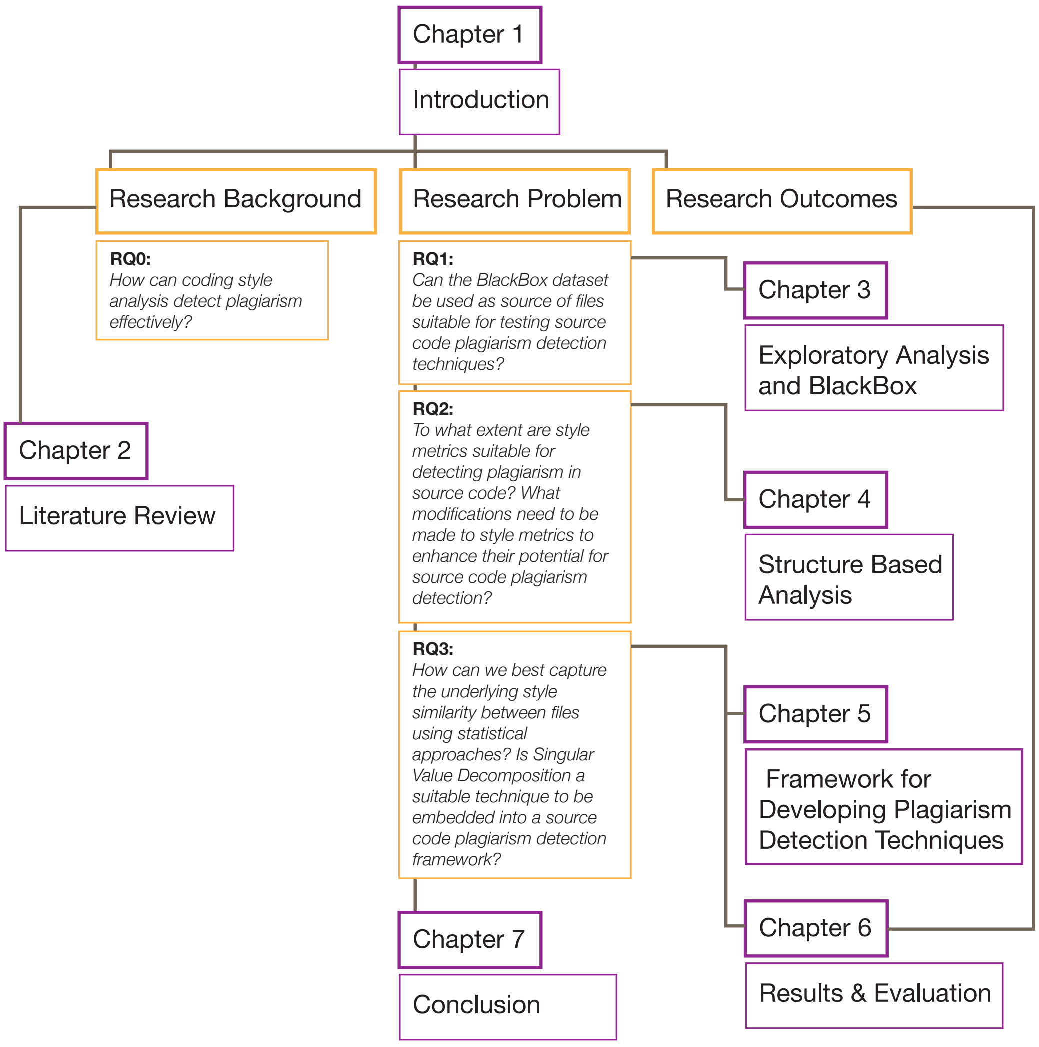

Chapter 1
Introduction
Research Background
Research Problem
Research Outcomes
RQ0: How can coding style analysis detect plagiarism effectively?
RQ1: Can the BlackBox dataset be used as source of files suitable for testing source code plagiarism detection techniques?
RQ2: To what extent are style metrics suitable for detecting plagiarism in source code? What modifications need to be made to style metrics to enhance their potential for source code plagiarism detection?
RQ3: How can we best capture the underlying style similarity between files using statistical approaches? Is Singular Value Decomposition a suitable technique to be embedded into a source code plagiarism detection framework?
Chapter 2
Literature Review
Chapter 3
Exploratory Analysis and BlackBox
Chapter 4
Structure Based Analysis
Chapter 5
Framework for Developing Plagiarism Detection Techniques
Chapter 6
Results & Evaluation
Chapter 7
Conclusion

Figure 1.1: Thesis Structure

This thesis is organised in seven chapters. After this introductory chapter the remainder of this thesis is structured as follows:

Chapter 2 provides a review of the background of plagiarism definition and deals with the literature on plagiarism in computer science. In addition, it reviews the literature in the interdisciplinary research relating to source code plagiarism. Also, it explores the detection techniques that relate to detecting similarity in source code and reviews coding style metrics. The chapter identifies the gap within the existing literature in source code plagiarism.

Chapter 3 provides background on the online software tool "BlueJ" and a brief description of the dataset "BlackBox". The BlackBox dataset is analysed to determine whether can be used as a source of datasets suitable for testing tools for plagiarism detection. Two principal techniques are used in the exploratory study of BlackBox: Grouping and Visualisation. Grouping supports statistical analysis of four random samples from BlackBox that indicates the presence of plausibly similar files. Visualisation is used to show that this similarity cannot in general be attributed to plagiarism. The chapter concludes with a discussion of RQ1 and OB1-3.

Chapter 4 builds on a re-implementation of the structure based approach to identify authorship based on coding style metrics. It introduces two new families of metrics: Modified and Extended. The Modified metrics adapt existing metrics so as to be more effective with Java code. The Extended metrics aim to detect some specific plagiarism attacks. The chapter concludes with a discussion of RQ2 and OB4-6.

Chapter 5 describes a framework for enhancing structure based approaches to source code similarity detection using alternative techniques for statistical analysis. This framework is modelled on a well established approach to document analysis which applies Singular Value Decomposition to term-document matrices. The new framework (the 'MFM Framework') is based on metric-file matrices. Its application is illustrated with reference to one of the groups studied in the SOCO dataset.

Chapter 6 presents the results of the application of the MFM framework in conjunction with the Modified and Extended families of metrics introduced in chapter 4. These results are compared with other techniques for plagiarism detection, such as JPlag, as applied to the SOCO dataset. These empirical studies address RQ3 and OB7-9.

Chapter 7 provides an overall conclusion regarding the contributions of the research conducted in this thesis. The limitations are discussed, as well as recommendations for future research.

Chapter 2

Background and Literature Review

2.1 Introduction

Plagiarism concerns reusing, copying or paraphrasing somebody else's work without making proper reference to the original author, or intentionally attempting to make the plagiarized work appear to be original (as in the case of student plagiarism) [116] [33]. Source code files can be examined for plagiarism based on their stylistic, structural and semantic similarities [130] [131].

This chapter is organised as follows: Section 2.2 explores the brief history of plagiarism detection and several definitions from the literature review. It also covers source code modification techniques in subsection 2.2.1 and source code detection techniques in subsection 2.2.2. The last section 2.3 presents the history behind coding style analysis and existing metrics, and how coding style metrics in authorship can be used to detect similarity between the files in source code.

Accordingly, this chapter serves as an essential basis for the research, given that it formulates the theoretical background upon which the research is based. Additionally, the output of this chapter fulfils the **OB1** research objective: To review the literature in the field of source code plagiarism.

2.2 Plagiarism Definition

Plagiarism is a "*theft of intellectual property*"[105] and could include plagiarism in music, painting, maps, technical drawings etc. This thesis is solely concerned with textual plagiarism. Textual aspects have little relevance for musical compositions or diagrams [98][80]. However, if information of any kind is exploited without giving appropriate acknowledgement of the owner of the information, it is called plagiarism [24]. Culwin and Lancaster defined plagiarism in the academic (student) context as "*the presentation of another person's ideas or materials as if it were one's own*"[34] [35]. Their definition flags the need for academic institutions to beware of plagiarism in terms of gaining academic credit for the student [36] [7].

Plagiarism in the student context is when a student decides to submit someone else's work as their own [67]. In a learning context, when a student expresses information (with the source properly acknowledged) in their own words, it may indicate that learning is occurring [33]. Many academic institutions have set thresholds regarding the degree of similarity between two works, aiming for students to take care when they are describing other people's work [32]. In the academic world, it is an offence to reproduce the work of another person without proper acknowledgement [80]. Originality is important in the production of new literature [116] [150] because "If everyone copied each other's work [it] would in the end produce no progress" [6] [119] and advances in literature would be limited. Joy and Luck [79] have their own definition, specifying unacknowledged copying of documents or programs, which occurs in many contexts including industry and academia. There is also an ongoing study of student perceptions and understanding of plagiarism, and a spread of awareness and motivation of how to avoid plagiarised work [79] [103] [123]. Martin [104] clarified plagiarism from an ethical point of view and identified six plagiarism forms:

1. **Word-by-word copying:** exact copying of another's work without any quotation or proper acknowledgement of the original author.
2. **Paraphrasing:** rewriting the sentences, with close paraphrasing (i.e. changing some words) without understanding of the concept of paraphrasing and without citing the original author.
3. **Plagiarism of a secondary source**: taking, referencing or quoting a work which has been taken from another work, without looking up the original work.
4. **Plagiarism of the form of a source:** is when the structure of an argument is copied from its source without providing acknowledgements.
5. **Plagiarism of ideas:** using the ideas that appear in the source text without reference to the original source.
6. **Authorship plagiarism:** taking someone's work and referencing it with one's own name.

In the rest of the chapter, the term of "style analysis" is being used to refer to "style analysis" for determining authorship in source code. Also, the definition of style analysis for this purpose is explained.

2.2.1 Source Code Modification Techniques

There have been several attempts to categorise the modifications of source code that can hide plagiarism [45]. The transformations can range from very simple changes to very difficult ones in source code, categorised as six levels of program modifications by Faidhi and Robinson [48] and Whale [147], as shown in Figure 2.1 and listed below:

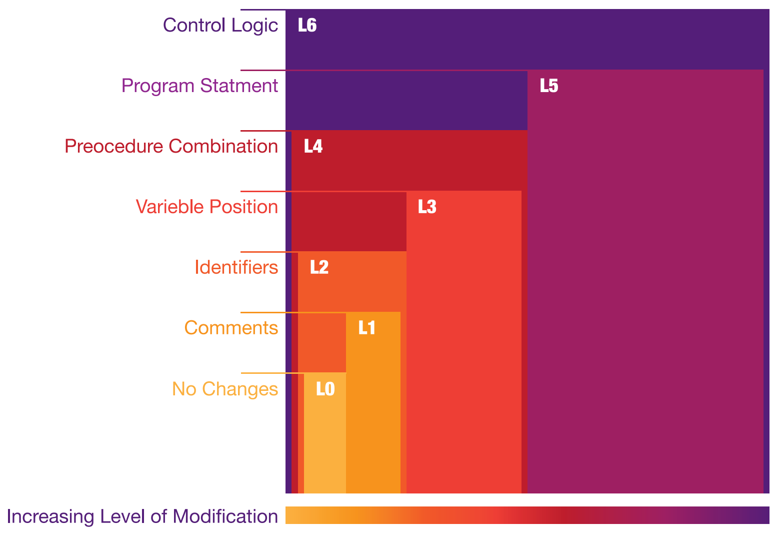


Figure 2.1: Source code plagiarism modification levels. Each level includes the modifications included in the 2 previous levels [48].

1. **Level 0 No changes:** This level contains no source code modification.

2. **Level 1 Changes in comments:** Modifying the comments in source code work, for example by providing explanations of the code or other information about the code. Comments in source code cannot be executed apart from the rest of the code and they differ between programming languages, for example

the syntax of code written in the Java language is different from that which was written in the C programming language [57] [63].

3. **Level 2 Changes in identifiers**: Making changes to identifiers, including changing names of variables, methods, classes or packages.

4. **Level 3 Changes in declarations**: These changes include modification in the way variables are declared. For example, someone could modify a statement which declares two variables at once by making it into two statements.

5. **Level 4 Changes in procedure combinations:** This includes adding redundant procedures or methods; merging procedures; and changing the order of procedures.

6. **Level 5 Changes in program statements:** This includes making changes to statements that make decisions, such as if statements, conditional expressions, comparison operators, nested if-else, Boolean operators and switch statements.

7. **Level 6 Changes in control logic**: This includes making changes to control the order of the program execution, which will affect the decision logic of the program.

The hierarchical system by Faidhi and Robinson [48] and Whale [147] and the definition of Parker and Hamblen [118] provide an overview of the different types of plagiarism attacks. Increasing the level of changes increases the complexity of plagiarism, with higher levels requiring a greater ability to understand the code and which are suitable changes in the source code. L0 to L2 are the simplest changes that can be made to source code, whereas the rest of the changes are carried out by more experienced programmers [119]. Karnalim [82] proposed a method operating on low level tokens such as those extracted from compiled code from a given source file, which detected most plagiarism attacks.

Source code changes can be classified into two types: “Lexical changes” or “Structural changes”. These classification is used by many other authors [78] [79] [97] [99] [129]. “Lexical changes” involve making simple changes to code such as comment and line spacing. In particular, there are 8 levels of lexical changes: L1: Modification of source code formatting, L2: Addition, modification or deletion of comments, L3: Language translation, L4: Program output reformatted or modified, L5: Identifiers renaming, L6: Variable declaration can be split or merged, L7: Modifiers can be added, modified and deleted and L8: Constant value modification. Conversely, “Structural changes” involve modifying functions or procedures, or changing the statement logic, and experienced programmers tend to carry out structural changes when attempting to disguise plagiarism. Structural changes comprise 11 levels: S1: Reorder variables in statement, S2: Reorder the statement within code blocks, S3: Reorder the code blocks, S4: Addition of redundant statements or variables, S5: Control structure modification, S6: Data types changes or data structure modification, S7: In-lining and re-factoring methods, S8: Redundancy, S9: Temporary variables and sub-expressions, S10: Source code structure redesign and S11: Scope modification.

Culwin and Naylor [37] identify three different kinds of plagiarisms: **Collaboration**, **Collusion** and **Copying**. Collaboration is when students share knowledge and ideas together; Collusion is when students let others have a look at their work; and Copying is when students share their work with others electronically. Whether this kind of behaviour are deemed to be plagiarism depends on the context. For instance, in this case of copying, it depends whether students are responsible for their individual work and for preventing others from copying them. On this basis, Joy and Luck indicated that the borderline between cooperation in work and plagiarism is not well defined [79]. Verco and Wise categorised people who plagiarise into two groups: **Novice programmers** and **Experienced programmers** [144] [145]. Novices are learners, and experienced programmers are more advanced and professional in writing source code. Decoo [40] assessed academic misconduct and noted that software plagiarism could be at many levels such as the user interface, content and source code.

In addition, a review of the current literature on source code plagiarism revealed that there is limited research on applying coding style analysis to source code plagiarism. There is a variety of easy ways to copy others' work because source code can be obtained from online source code banks and text books [33].

2.2.2 Source Code Detection Techniques

There are several techniques to detect source code plagiarism. Prechelt et al. [125] identify two main categories of automated plagiarism detection for program source code: **Attribute Counting** and **Structure Based** [77].

2.2.2.1 Attribute Counting

Attribute counting is used to detect general document plagiarism by measuring and representing the textual similarity [93], metrics similarity [92], feature-based similarity [146] and shared information [26], which all depend on measuring features of the text in its raw or tokenised form. The information in two files to be compared for similarity [20] [74] is based on three profiles [77]:

1. **Physical Profile**: Characterises a program based on its physical attributes, such as the number of lines, words and characters.

2. **Halstead Profile**: Characterises a program based on its token types and frequencies. These relate to the number of token occurrences (N) and the number of unique tokens (n).

3. **Composite Profile**: A combination of the physical profile and the Halstead profile.

Yamamoto et al. [149] used a number of line metrics to detect similarity between source code files, while Johnson [76] identified redundancy in source code using fingerprints. A recent work by Tahaei and Noelle [138] utilised attribute counting in order to count student resubmission patterns for programming exercises, whereas Jiang and Wang [75] proposed a source code detection method that uses a "frequent item set" metric to detect similar files. A recent study by Okutan [115] used a token based strategy which was based on two approaches: defect and non-defect files. They proposed metrics to achieve better performance compared to existing static metrics. Duracik et al. [47] focused on how to select a proper representation of source code when searching for plagiarism. They used abstract syntax trees (ASTs) in their comparison. Bandara and Wijayarathna [8] presented a new source code author identification system based on unsupervised feature learning techniques. Their system uses nine source code metrics, each of which is then tokenised, and an unsupervised neural network technique called Sparse Auto-encoder [12] is used to extract features which finally train the Logistic Regression supervised learning algorithm [17]. They evaluated their approach using 5 large datasets written in the Java programming language. The result of their evaluation failed when there is more than one author, but succeeded in identifying single authors.

Caliskan-Islam et al. [24] proposed a new method to classify authors' source code, using machine learning. Firstly, they started with parsing the source code; secondly, they define features to represent syntax and structure program code; thirdly, they explored a random forest classifier [19] for classification. They used code from Google Code Jam (an international programming competition) and achieved 95.33% accuracy. There are a number of plagiarism detection tools that use the attribute counting similarity approach [77]. However, the attribute counting approach merely compares the number of variables, loops, etc in two files. Although this is a widely adopted approach to detecting plagiarism, the same number of attributes can occur among files, but the files may not necessarily have been plagiarised [45].

2.2.2.2 Structure Based

Structure based metrics detect similarity in the structure of files. Structure based methods include character string [145], and parse tree [10] [134] methods. Structure based approaches are considered as robust to most types of source code modification [23]. Programs are compared in two stages. First, the code is parsed and the token sequences are generated. Second, the tokens are compared [77].

Kuo et al. [96] developed an automated structure based system called the Structure Plagiarism Detection System (SPDS), focusing on cosine similarity and the Winnowing algorithm, which also showed good results compared with different approaches. Delev and Gjorgjevkj [41] compared source code files in the C programming language with 13 string matching based methods: Levenshtein, normalized Levenshtein, weighted Levenshtein, Damerau Levenshtein, Optimal String Alignment (OSA), Jaro-Winkler, Longest Common Subsequence (LCS), Metric Longest Common Subsequence (M-LCS), N-gram, Q-gram, Cosine similarity, Jaccard index and Sorensen-Dice coefficient. They performed the analysis on three different datasets and they computed the similarity among potential plagiarism pairs of source code files.

A large number of existing studies in the broader literature have examined token based techniques. A recent study by Okutan [115] used a token based strategy and proposed metrics to achieve better performance than existing static metrics. Karanalim and Budi [82] employed a human predictive methodology, using three evaluation mechanisms: think aloud, aspect-oriented and empirical evaluations. They found that the structure based approach was more effective than the other detection methods. Chilowicz et al. [27] proposed a new similarity detection based on token sequence matching and factorisation of the function called Call Graphs. The process of factorisation involved merging factors of the code with inlining function, where function calls were replaced by their body up to a single level and outlining functions were moved outside their parent function. The outcomes of their method revealed that the call graph could detect similarities in the source code file.

Liu et al. [99] proposed a new tool called GPLAG that uses a program dependence graph (PDG), which is a graphical representation of data and control dependences within a procedure. Their experiments showed that GPLAG is effective and efficient. DeSOCoRe is a tool proposed by Whale [147] that shows plagiarism results in graphical representations when comparing two source code files at function levels. Shann et al. [132] proposed a method that converted the source code to the assembly language and computed the similarity between the files using the Karp and Rabin (KR) improved algorithm [148]. They analysed 27 programs and the results showed that the detection analysis between two files has great practical value.

Santanu and Atul [119] presented a framework that detected the “pattern languages” that detect code features. They implemented the SCRUPLE machine based tool which extended the source code with pattern matching symbols. The work in this research was based on the “structure pattern” that shows the proposed framework to simplify the task of locating the code fragment. By contrast, Chen et al. [26] designed a system called SID (Software Diagnosis System) which took the identified metric and used the Heuristic Compression algorithm to compute the normalized amount of shared information between two programs.

2.3 Coding Style Analysis

One kind of source code plagiarism detection attempts to identify the similarity of the code from the way the code is written - the *coding style* - which may be derived from coding conventions (sets of guidelines for a particular programming language, perhaps defined for use by a particular institution or company). Coding styles are a way of writing codes so that they are standardised enough to be consistent with the functions they are supposed to serve [6]. Coding Style is a factor which can be used to detect source code plagiarism because it relates to programmer personality but does not affect the logic of a program and can thus be used to differentiate between coders.

The "style" that is part of the program is occasionally considered as a nebulous attribute that is unmeasurable; it solely depends on the instincts of the programmer that may be "good" or "bad" and is very similar to the way in which the artist recognizes any painting as being in the categories of good and bad paintings [15] [87]. Each programming language has its own code convention, but conventions typically cover such aspects as the following [28]:

- File organization: includes the introductory comments in the main file and the packages imported into the file.
- Indentation: which refers to program's structure and control flow. The main symbol to differentiate the indent style is bracketing ({}). Line length may also be significant e.g. line length maybe limited to 80 characters.
- Comments: There are two main type of comments: implementation and documentation. Implementation comments can be block comments, single line comments, trailing comments and end-of-line comments.

- Declarations: which declare an identifier's significant characteristics. They are used in functions, variables, constants and classes.
- Use of white space: includes vertical white space such as blank lines, which improve the readability of the source code. This can be one or two blank lines, using the Enter key on the keyboard. On the other hand, horizontal white space, such as blank spaces, can be after commas, keywords or binary operators, using the Spacebar or Tab.
- Naming convention: that make the piece of code more readable and gives more information for each block of the code. The naming convention refers to identifier types such as packages, classes, interfaces, methods, variables and constants.
- Programming practices: that involve the rules which are set for developers to improve software quality. This includes prerequisites for the software like: life cycle and development structure, the requirements of the software, structure of the software system, the individual components of the software and the chosen programming language [106].

Kernighan and Plauger [84] mention that the code should not be written solely for the compiler or personal use but also for human readability. Coding style is subjective and can sometimes be difficult to define. In addition, coding style deals with visual appearance of the source code to be more readable for programmers and developers.

Oman and Cook [116] covered typographic or layout style, that is, the formatting and commenting of source code which does not affect the execution of the program [117]. Spaord and Weeber [136] explained source code features which might identify the author of the code and refer to their work as “software forensics”. They divided the analysis of the code into two dierent parts: analysis of the executable code and analysis of source files.

Ohno and Murao [112] used simple tokenised coding style rules for Java source code categorised in three token groups: (i) basic point tokens, such as opening and closing braces; (ii) identification tokens, such as single and double spaces; and (iii) other tokens, such as reserved words and identifiers. The authors proposed a new method called Coding Model (CM), based on the Hidden Markov Model (HMM), that quantifies the features based on students' coding style [113]. They conducted an experiment using Java code, which confirmed that the coding models can distinguish between source code produced by different students. Also, they proposed a combined method that measures the similarity among programs by SIM (Similarity measurement), which is a structural method that measures the similarity between two computer programs by reducing the parse trees of the code to strings, then applying a string matching algorithm to find common token sequences [60]. The authors expected the combined method to reduce the number of false positives detected [114] [111].

Arabyarmohamady et al. [5] proposed a coding style plagiarism detection framework, which performs the detection in two phases. In the first phase, a compact representation of the code is produced, and in the second phase the extracted attributes are input into three modules to detect plagiarised code and to determine authorship. The system was evaluated on 120 student assignments in C/C++. There are three main findings in their paper. First, the system is fast and can work on large datasets, since the two phase approach creates a feature file for each document to reduce the time. Second, the framework provides a method to detect the original author and the user of the code. Last, the framework is capable of detecting plagiarised documents which have been copied from the Internet or implanted by a third party.

2.3.1 Style Metrics

According to Conte et al. [31], “Software metrics are used to characterise the essential features for software quantitatively, so that classification, comparison and mathematical analysis can be applied”. Source code metrics have been researched to detect authorship [95]. In Figure 2.2, the diagram shows the history of the metrics collection in four periods: in the 70s, 80s , 90s and after new millennium (2000).

In 1978 Tassel [139] introduced the first style metrics aimed at improving style and therefore readability of C programs.

Then Conte et al. [31] in 1986 compiled a fuller list of complex style metrics. In 1987, Tauer and Ledgard [140] also used C and compiled a full list of rules for excellent programming; in the same year, Kernighan and Plauger [84], whose concern was with programming style in order to build good programming practice, analysed more than seventy rules for programming. In 1987 Oman and Cooke [116] introduced a list of programming style metrics and identified 236 different styles, based on typographic characteristics.

In 1993, Ranade and Nash [127] went even further, drawing up style rules for the programming language C comprising over three hundred pages. In 1997, to organise these varying lists and compilations, and because of the massive number of rules and metrics from all previous studies, Krsul and Spafford [95] extracted fifty style metrics for the programming language C, which they separated into three categories defined as: Programming Layout Metrics; Programming Style Metrics; and Programming Structure Metrics.

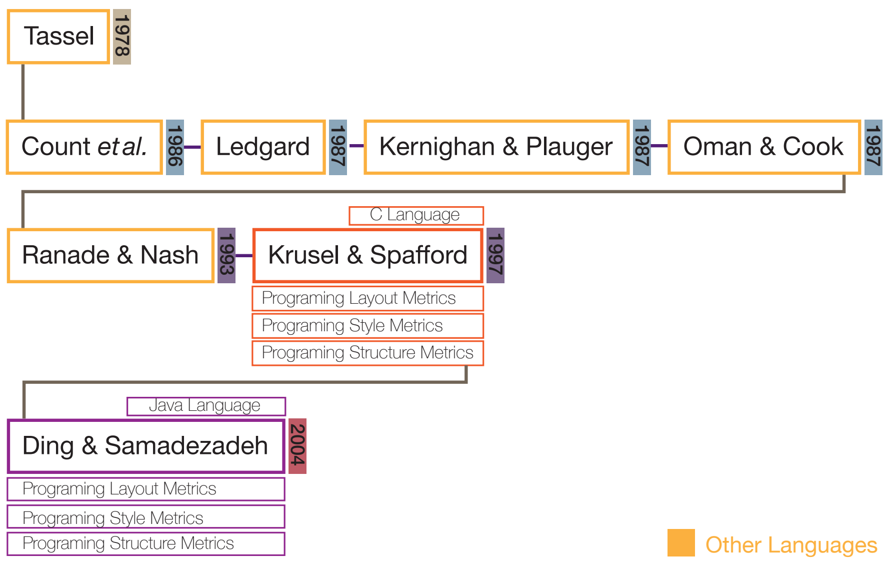


Figure 2.2: The use of Style Metrics throughout the years

Building on the previous work, in 2004, Ding and Samadezadeh [44] used the same three categories for their Fingerprint implementation which is compatible with Java code rather than C. Their study revealed significant stylistic differences between C program and Java program. The purpose of their study was to extract software metrics from Java language for authorship identification.

A code file consists of many sections that should be separated by lines and by optional comments identifying each section in the code for readability. Each Java source file contains a single public class interface and multiple private classes and interfaces in one file. Most Java files start with comments showing the author name, the date and the name of the program. Then come the packages and import statements, followed by class and interfaces declarations.

The three categories for the coding style existing metrics (Layout, Style and Structure) have their own descriptions and the metrics under each of them address different kinds of source code features. Those features are mostly for the layout category; the style category concerns the syntax of the code without consideration of the semantics of the code; and the structure category is concerned with the logic of the code in the file. The three categories for the existing metrics are described in detail below. The codes used to refer to metrics in these categories (viz. STY, PRO and PSM) are introduced by Krsul and Spafford [95].

1. **Layout Metrics (STY):**

Table 2.1: **Layout** metrics extracted from the source code of Java programs

Metric	Description
STY1	A list of metrics indicating indentation style
STY1a	Percentage of open braces ({) that are along a line
STY1b	Percentage of open braces ({) that are the first character in a line
STY1c	Percentage of open braces ({) that are the last character in a line
STY1d	Percentage of close braces (}) that are along a line
STY1e	Percentage of close braces (}) that are the first character in a line
STY1f	Percentage of close braces (}) that are the last character in a line
STY1g	Average indentation in white spaces after open braces ({)
STY1h	Average indentation in tabs after open braces ({)
STY2	A vector of metrics specifying comment style
STY2a	Percentages of pure comment lines among lines containing comments
STY2b	Percentages of "//" style comments among "//" and "/*" style comments
STY3	Percentages of condition lines where the statements are on the same line as the condition.
STY4	Average white spaces to the left side of operators
STY5	Average white spaces to the right side of operators
STY6	Ratio of blank lines to code lines (including comment lines)
STY7	Ratio of comment lines to non-comment lines
STY8	Ratio of code lines containing comment to code lines without any comments

Table 2.1 shows the layout metrics used in this category, which associate numerical measures with the following:

- Whitespace, which can be single blank lines such as between methods and local variables, or two lines such as between sections and classes. It can also be blank space which appears after commas, all binary operators except increment `(++)` and decrement `(--)`, and the expression in a `for` statement.

- Placement and indentation of brackets for blocks of code. For example, four spaces may be used as the start of indentation.
- Comments with different styles such as are featured in Figure 2.3:
  (a) Block comments give a description of files, methods and algorithms as illustrated in Figure 2.3 (a); such comments start with `(/*)`, have multiple lines and ends with `(*/)`;

  (b) Single line comments appear in a single line with indentation of the same level of the code as shown in Figure 2.3 (b); the comment starts in line 3 with `(/*)` and ends with `(*/)` as an individual single line;

  (c) Trailing comments are usually very short and give a hint of something in the code as shown in Figure 2.3 (c); the comments start in lines 2 and 4 with `(/*)` and end with `(*/)` and run along a line with the code;

  (d) End-of-line comments can be used to comment code out, or appear at the end of the code line as shown in Figure 2.3 (d); the comment in line 3 is used as a short comment and the comments in lines 9-14 as a comment about the code.

Altering these metrics is easy for formatters and pretty printers; they can also be changed by the text editor with which the program is composed; the editor's default can be used, or a preferred layout can be set.

```
/*
 * Here is a block comment.
 */
```

(a) Block Comments

```
if (condition) {

    /* Hello Olfat :) */
    ...
}
```

(b) Single line Comments

```
if (A == 10) {
    return TRUE;  /* Happy :) */
} else {
    return Happy(A); /* <3 <3 */

}
```

(c) Trailing Comments

```
if (A > 1) {

    // How are you Olfat?
    ...
}
else {
    return false;
}
//if (bar > 1) {
//    ...
//}
//else {
//    return false;
//}
```

(d) End-Of-Line Comments

Figure 2.3: Types of programming comments: (a) Block Comments, (b) Single line Comments, (c) Trailing Comments and (d) End-Of-Line Comments

2. **Style Metrics (PRO)**:

   Table 2.2 shows the style metrics that are listed in this category. They are connected with Programming Layout metrics, but are harder to change and fall within a higher level of the modification in the schema of Faidhi and Robinson [48] and Whale [147].

Table 2.2: **Style** metrics extracted from the source code of Java programs

Metric	Description
PRO1	Mean Program line length in terms of characters
PRO2	A vector of metrics of name lengths
PRO2a	Mean variable name length
PRO2b	Mean function name length
PRO3	Character preference of uppercase, lowercase, underscore, or dollar sign for name convention.
PRO3a	Percentage of uppercase characters
PRO3b	Percentage of lowercase characters
PRO3c	Percentage of underscores
PRO3d	Percentage of dollar signs
PRO4	Preference of either `while`, `for` or `do` loops
PRO4a	Percentage of `while` in total of `while`, `for` and `do`
PRO4b	Percentage of `switch` in total of `while`, `for` and `do`
PRO4c	Percentage of `do` in total of `while`, `for` and `do`
PRO5	Preference of either `if-else` or `switch-case` conditions
PRO5a	Percentage of `if` and `else` in total of `if`, `else`, `switch`, and `case`
PRO5b	Percentage of `switch` and `case` in total of `if`, `else`, `switch`, and `case`
PRO5c	Percentage of `if` in total of `if` and `else`.
PRO5d	Percentage of `switch` in total of`switch` and `case`.

The metrics in this category include:

- Length of comments that can affect the readability of the code;
- Naming preferences which give a clear understanding of the code;
- Variable lengths;
- Preferences of condition statements as shown in Figure 2.4:
  (a) How `if` statements should look, shown in Figure 2.4(a),
  (b) How `if-else` statements should look, shown in Figure 2.4 (b),

(c) How an `if else-if else` statement is used in a fragment of code, shown in Figure 2.4 (c);

(d) How nested `if` statements should look, shown in Figure 2.4 (d).

- Preference for loop statements as shown in Figure 2.5

  (a) How `for` statements should look, shown in Figure 2.5 (a);

  (b) How `while` statements should look, shown in Figure 2.5 (b);

  (c) How `do-while` statements should look, shown in Figure 2.5 (c);

  (d) How `switch` statements should look, shown in Figure 2.5 (d).

```
if (condition) {
    statements;
}
```

(a) `if` Statement

```
if (condition) {
    statements;
} else {
    statements;
}
```

(b) `if-else` Statement

```
if (condition) {
    statements;
} else if (condition) {
    statements;
} else {
    statements;
}
```

(c) `if else-if else` Statement

```
if (condition1)
{
   statements;
   if (condition2)
   {
   statements;
   }
}
```

(d) Nested `if` Statement

Figure 2.4: Types of `if` condition: (a) `if` Statement, (b) `if-else` Statement, (c) `if else-if else` Statement and (d) Nested `if` Statement

```
for (initialization; condition; update) {
    statements;
```

(a) `for` Statement

```
while (condition) {
    statements;
}
```

(b) `while` Statement

```
do {
    statements;
} while (condition);
```

(c) `do-while` Statement

```
switch (condition) {
case ABC:
    statements;
    /* falls through */
case DEF:
    statements;
    break;
case XYZ:
    statements;
    break;
default:
    statements;
    break;
}
```

(d) `switch` Statement

Figure 2.5: Types of condition statement: (a) `for` Statement, (b) `while` statement, (c) `do-while` statement and (d) `switch` statement.

3. **Structure Metrics (PSM)**: Structure metrics (see Table 2.3) are most appropriately applied to programs written by programmers with higher levels of ability and experience. A Java program consists of classes, and at least one of those classes should be "public static void main". The main aim of using programming structure metrics in this context is to improve the quality, clarity and the control flow for the program [4], e.g to make the logic of the code structure being written more efficient and easier to understand and modify. Most of the metrics are formulated as mean lines of code per function, usage of data structure, etc.

   The structure metrics in Table 2.3 analyse program structure with reference to keywords. In practice, the structure of a program may be reflected in identifiers and functions that are introduced by the programmer. It is unclear whether there is a standard generic code matching algorithm that can be applied in such situations. Plagiarism detection through the use of a word-matching algorithm [111] where words are not among a given set of keywords is more complex. For instance, some word matching will need to be case-sensitive [117].

Table 2.3: **Structure** metrics extracted from the source code of Java programs

Metric	Description
PSM1	Average non-comment lines per class/interface
PSM2	Average number of primitive variables per class/interface
PSM3	Average number of functions per class/interface
PSM4	Ratio of interfaces to classes
PSM5	Ratio of primitive variable count to lines of non-comment code
PSM6	Ratio of function count to lines of non-comment
PSM7	A list of ratios of keywords to lines of non-comment code
PSM7a	Ratio of keyword `static` to lines of non-comment code
PSM7b	Ratio of keyword `extends` to lines of non-comment code
PSM7c	Ratio of keyword `class` to lines of non-comment code
PSM7d	Ratio of keyword `abstract` to lines of non-comment code
PSM7e	Ratio of keyword `implements` to lines of non-comment code
PSM7f	Ratio of keyword `import` to lines of non-comment code
PSM7g	Ratio of keyword `instanceof` to lines of non-comment code
PSM7h	Ratio of keyword `interface` to lines of non-comment code
PSM7i	Ratio of keyword `native` to lines of non-comment code
PSM7j	Ratio of keyword `new` to lines of non-comment code
PSM7k	Ratio of keyword `package` to lines of non-comment code
PSM7l	Ratio of keyword `private`to lines of non-comment code
PSM7m	Ratio of keyword `public` to lines of non-comment code
PSM7n	Ratio of keyword `protected` to lines of non-comment code
PSM7o	Ratio of keyword `this` to lines of non-comment code
PSM7p	Ratio of keyword `super` to lines of non-comment code
PSM7q	Ratio of keyword `try` to lines of non-comment code
PSM7r	Ratio of keyword `throw` to lines of non-comment code
PSM7s	Ratio of keyword `catch` to lines of non-comment code
PSM7t	Ratio of keyword `final` to lines of non-comment code

2.4 Conclusion

This chapter provides a review of source code similarity detection in the field of plagiarism. It identifies the limitations of existing plagiarism detection methods. In conclusion, there has generally been a lack of techniques for detecting source code similarity from the style analysis perspective. There is a clear need for research to consider style analysis directed at program structure. Therefore, the findings of the literature review in this chapter confirm that further studies are required using existing approaches [44]. In addition to identify source code similarity using existing approaches, one of the main contribution of this thesis is to propose an alternative approach to the statistical analysis of coding style metrics that can deliver better results in plagiarism detection. The literature review has also revealed that not many benchmark datasets exist for evaluating source code similarity detection methods. The next chapter, chapter 3, examines the BlackBox dataset in order to determine whether it is a suitable dataset for evaluating source code similarity detection methods.

Chapter 3

Exploratory Analysis and BlackBox

3.1 Introduction

This chapter investigates the suitability of a large source code dataset, BlackBox as a resource to assist developing and evaluating plagiarism detection tools. This investigation takes the form of an exploratory study aimed at determining whether BlackBox is a suitable source of datasets for testing plagiarism detection techniques ('PD-testing datasets'). The primary motivation for this study is generating suitable source code datasets to use when building and evaluating the plagiarism detection framework to be proposed in Chapter 4, where the emphasis is on detecting source code plagiarism through style analysis.

3.1.1 An Overview of the Exploratory Study

The most direct way in which to try to extract PD-testing datasets from BlackBox is to select sample groups of files at random. The overall concerns are to determine

- whether the samples contain suspiciously similar files
- whether datasets containing family of suspiciously similar files can be reliably generated in this way.

The investigation started with choosing the online software “BlueJ” for our further analysis and then accessing the “BlackBox” dataset. A representative set of source code files (i.e. a sample) was extracted from the BlackBox dataset to show the types of source code files in BlackBox. Following the guidelines given by Cohen [30], four random samples of 250 files were downloaded from BlackBox so as to ensure that the samples are representative of the content of the BlackBox dataset.

Where identifying suspiciously similar files within each sample is concerned, certain groups of files can be discounted. These include template files in the form of skeleton programs and incomplete programs. The remaining files can then be informally classified into groups of files that might plausibly be similar. A grouping based on length of files alone was sufficient to produce three groups of files with broadly similar characteristics: ‘Short’ files (fewer than 40 lines), ‘Intermediate’ files (more than 40 and fewer than 100 lines) and ‘Long’ file (more than 100 lines).

Where reliably generating families of similar files is concerned, empirical evidence confirmed that all four samples had a similar distribution of files into Short, Intermediate and Long groups.

At this stage, the exploratory study indicated that randomly generated 250 file samples from BlackBox are likely to contain similar files (e.g. within the associated Short group of files). To complete the exploratory study, it remained to determine whether such similar files might indeed be *suspiciously* similar. To identify suspicious files, existing plagiarism detection tools were applied to Blackbox and the results were scrutinised to cluster the suspiciously similar files into groups.

Visualisation of similarity between source code files using Gephi [1] was used for this purpose.

As shown in Figure 3.1, two principal techniques were used in the exploratory study.

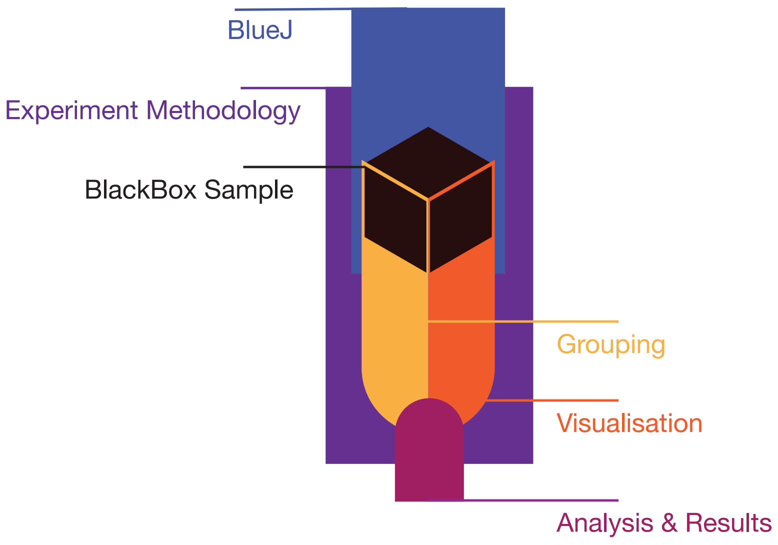


Figure 3.1: BlackBox Exploratory Study

Grouping: splitting the dataset into five groups: Template group, Short group, Intermediate group, Long group and Incomplete/Empty group. This grouping was based on the number of lines, code documentation, and number of loops.

Visualisation: the dataset sample was fed into a tool called Gephi that helps finding similarities and visualises them using the Chinese Whispers clustering method. This visualisation approach was a collaboration with a PhD student at Faculty of Organization and Informatics, University of Zagreb, with their dataset and my pre-processed dataset extracted from BlackBox.

The exploratory study is discussed in detail in the subsections that follow.

[1]https://gephi.org/

3.2 BlueJ and BlackBox

3.2.1 BlueJ

BlueJ is a Java Integrated Development Environment (IDE) online tool. Kölling developed an instructional environment called Blue for his PhD work; he then released it as **BlueJ**, which combines Blue and Java, in a collaboration with Rosenberg at Monash University [90] [88] and in 2009 BlueJ became free and open source software [91].

BlueJ has many features that aid the learning experience for the user. For instance, the main screen shows the structures of required applications that are characterised by BlueJ using Unified Modelling Language (UML) such as in Figure 3.2. UML was used instead of a standalone text editor or command line environment to make the dataset easily understandable. In UML, BlueJ gives a simple, visual, and interactive environment [89]. The primary focus for the development of BlueJ was to address the issues related to teaching programming languages that are oriented towards objects: moving towards higher level abstraction and more complex program structures. By way of illustration, Figure 3.3 shows how the hierarchy of the code is identified.

Some previous experiments have used BlueJ to identify user behaviour in Java coding, for example, novice use of compilation by students learning object-oriented programming using BlueJ [71]. The study started with automated observation of novice compilation as it naturally occurred in classroom tutorial sessions. Some error types were considered such as missing semicolons, unexpected brackets, illegal starts of expressions, and unknown classes. Their data included source code edits, compilation results, and the use of various tools within BlueJ (such as the debugger). The findings were significant in identifying user behaviour that had a

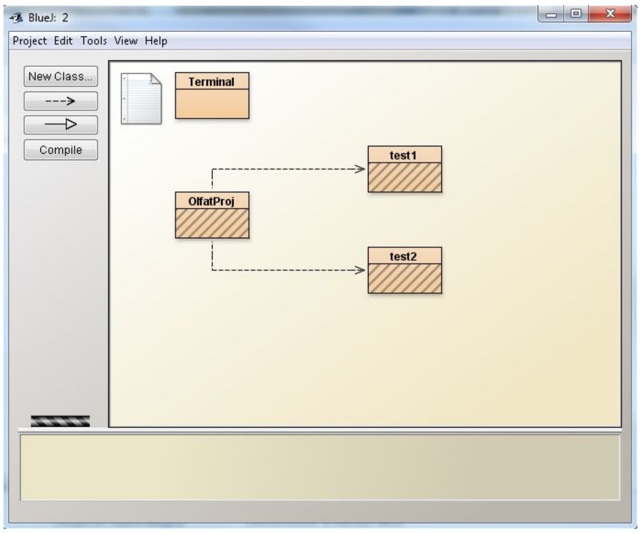
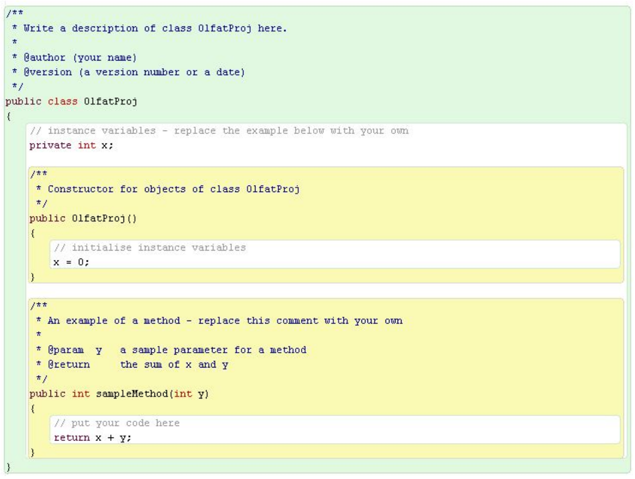


Figure 3.2: BlueJ UML Digram

practical impact on the design of the environment and the development of a visualising tool [72][50].

3.2.2 BlackBox

BlackBox is a collection project at Kent university to help other researchers to find usable data [22]. It contains the results of over 100 million compilation events from over 1 million users. The number of BlueJ users leads to a huge number of files in BlackBox, with a wide variety of user programming experience. The motivating idea behind the BlueJ project is teaching the Java programming language, especially in classrooms for beginners, as it allows students to concentrate on understanding how to solve Java homework without the distraction of executing and compiling issues [143]. The new learner will learn to build the programs using visual objects and classes. Many levels of programming can be examined in BlackBox. So most of the files in BlackBox are created by students solving problems from their programming classes. As Figure 3.3 shows, some of the programs contain common ground code



```java
/**
 * Write a description of class OlfatProj here.
 *
 * @author (your name)
 * @version (a version number or a date)
 */
public class OlfatProj
{
    // instance variables - replace the example below with your own
    private int x;

    /**
     * Constructor for objects of class OlfatProj
     */
    public OlfatProj()
    {
        // initialise instance variables
        x = 0;
    }

    /**
     * An example of a method - replace this comment with your own
     *
     * @param  y   a sample parameter for a method
     * @return     the sum of x and y
     */
    public int sampleMethod(int y)
    {
        // put your code here
        return x + y;
    }
}
```

Figure 3.3: BlueJ Java Code Hierarchy

which is already written and the student just needs to fill the gaps in the classes to understand the code hierarchy.

Several studies investigate various mechanisms to describe the BlackBox dataset properties. In the first instance, Brown et al. [22] faced some technical challenges with respect to properties such as anonymity. Strong anonymity involves removing all comments and renaming the variables; some useful information will then be lost and cannot be used in any analysis. So, they had to remove just the "header comment" where the user name is usually written and replace it with a dash "–". The BlueJ (BlackBox) dataset collection is anonymous for the purpose of supporting any research experiment or collecting any kind of information at any level [21]. In addition, another issue discussed in their study is that of data caveating, where there may be a need to hide which code belongs to which user. There are some cases that need to be considered when carrying out a dataset analysis that takes account of user identity. For example, if two users use the same account in BlueJ, it will appear as if they are a single user. On the other hand, if someone uses BlueJ on a university machine, laptop and on a home machine, it will appear that they are three users. If a user has a BlueJ project file on a USB stick and opens it on two different machines with the same pathname, it will appear that they are two projects belonging to two different users. It is possible to track the user, but there is no evidence about the user history profile [22].

The BlackBox dataset has been also used in research investigating issues encountered by students learning Java . A recent project by Keuning et al. [85] has focused on the issues which occur most frequently, and whether students are able to solve these over time and by the use of code tools. They found that students usually find it difficult to fix issues related to modularization, and code tools do not improve solutions. For assessing the behaviour of novice programmers, a study by Jadud [72] used error quotient (EQ) to score how students deal with correcting syntax error in their programs. In an investigation involving the BlackBox dataset, Jadud and Dron [73] applied the general EQ to assess the compilation behaviour of users from 10 different countries.

3.3 Evaluating BlackBox for Plagiarism Detection

The purpose of our exploratory study of the BlackBox dataset is to determine whether it can serve as a source of datasets suitable for evaluating plagiarism detection techniques ('PD-testing datasets').

There are around 17 million files in the BlackBox corpus and it is difficult to choose files randomly without any criterion[2]. To tackle this, a small program ('Fetcher') that can fetch files randomly from the BlackBox database was developed. Access to BlackBox was with permission of the administration at Kent University, and allows researchers to download the files as required. So, the program remotely accessed the dataset with an SQL connector[3].

In the exploratory study, a key issue was to determine an appropriate sample size for the main study. Sample size influences the detection of significant differences, and depends on the aim of the study, the population size and the sampling error [70], the level of confidence and the degree of variability [108]. The confidence level is selected (usually 95 per cent) so that if the population was sampled multiple times, the confidence interval would include the true population mean 95 per cent of the time [94] [110].

The dataset in BlackBox is growing daily and in October 2015 included around 17 million Java source code files. Predefined tables of sample size by Cohen et al. [30] suggest that for any population of more than a million items, a random sample of 250 items would provide a sufficient sample size, therefore, we treated the files in BlackBox as the population, and in order to meet the requirement of the sample size, we used the Fetcher implementation four times to download four random samples, each containing 250 Java files.

[2]https://bluej.org/blackbox
[3]https://bluej.org/blackbox/#Access

3.3.1 How Fetcher Handles File IDs in BlackBox

BlackBox contains a wide variety of code submissions, and BlueJ is open source and free to use by anyone. The dataset uses three different types of ID:

- User ID (uID);
- Submission ID (sID); and
- BlackBox ID (bID).

The user ID (uID) is for the user when they create an account in BlueJ and submit a file (no matter how many files), whereas the submission ID (sID) is for each file submitted by the user with that user ID. The BlackBox ID (bID) is for each file in BlackBox independent of the user ID and submission ID. For example, if one user has submitted one file from two different machines, it will appear as two files with two different sIDs. To avoid duplication for choosing the sample for our research, the proposed method for choosing the sample was to build an approach with specific requirements such as choosing one random file from a set of 10 submitted files associated with one user uID. However, if there were two users using one account, this would appear as one account with two different submission sIDs and BlackBox bIDs. To download a random sample, there is a need to design a fetcher from BlackBox in order to reach the target for the random sample.

Figure 3.4 shows an explanation of how our small Java implementation for fetching the files randomly is designed. The figure shows that BlackBox has 3 different IDs. We took an example of a user uID (uID12), and this user had two submission files (sID3 and sID9), which could represent the user submitting the same files twice within a short time if they are in the same block of 10 sID files. Those two submission sIDs are related to the user uID (as sID3-uID12 and sID9-uID12). On the other hand, each file in the BlackBox has its own unique bID with

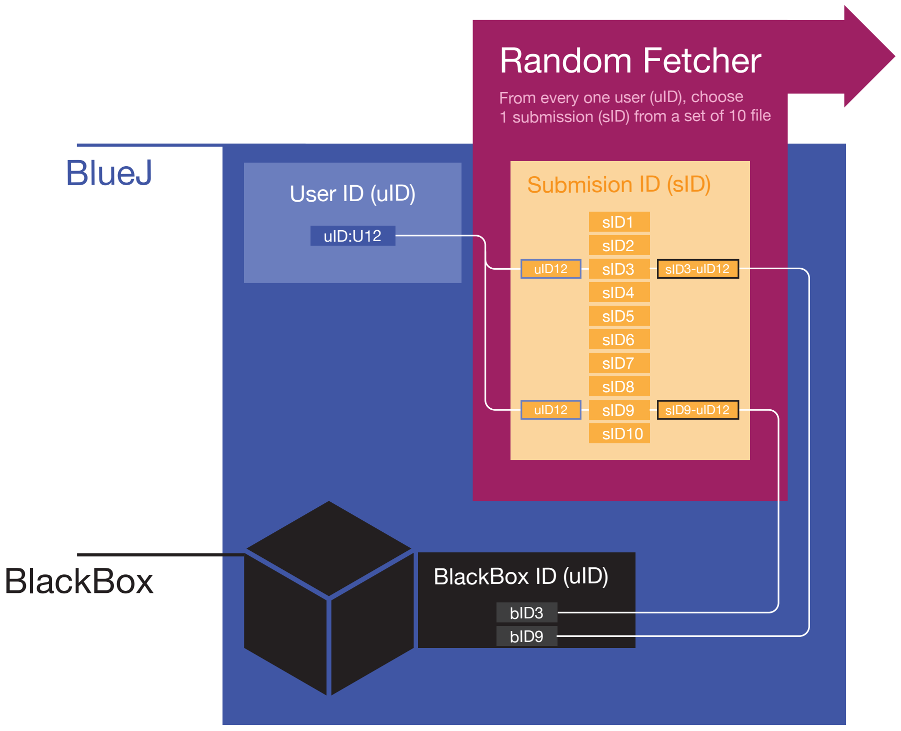


Figure 3.4: This is how Random Fetcher works with BlackBox IDs

no relation to any uID or sID. Therefore, in order to sample the BlackBox submitted files while avoiding capturing identical files, Fetcher was constructed to fetch one uID-sID from every block of 10 sID files.

3.3.2 Grouping

The primary purpose of grouping is to determine whether each 250-file sample can be considered representative of the types of source code files stored in BlackBox. A secondary goal for grouping is to determine whether BlackBox is a suitable source of PD-testing datasets.

Pre-processing is a part of any dataset analysis in research. In a data mining study, there may be files that are irrelevant or redundant in the dataset [126] and it may be important to ensure that the file name and author ID are hidden [22]. Aspects of the task of pre-processing a random sample had to be performed manually, to validate the suitability of the sample and eliminate outliers. Pre-processing involved removing blank white lines from the code files in the sample.

The first random sample was downloaded and then processed into groups in three stages:

- The first stage involved manual inspection of the sample to identify file that were unsuitable as test files for plagiarism detection. These include files that take the form of program templates where the programmer is intended to write code to complement the template. They also include incomplete files representing unfinished programs.

- The second stage subdivided the remaining files in the sample into three groups according to their number of lines using a simple Java program. The three groups of files were Short files with fewer than 40 lines, Intermediate files with between 40 and 100 lines and Long files with more that 100 lines.

- The third stage analysed the Short, Intermediate and Long files to identify the characteristics of typical files in each category and to establish that similar files can be found within each group. The analysis of files at this stage was an informal process carried out manually.

In stage three, the files were analysed manually with regard to the following specific content features:

1. Number of lines;
2. File size;
3. Number of loops:
    (a) for;
    (b) if and if else;
    (c) while.
4. Documentation:
    (a) Type one comments: //
    (b) Type two comments: */*
    (c) Type three comments: /*

The aim of the manual analysis was to identify the relevant characteristics of files in each category with reference to an indicative set of metrics. In this context, relevant characteristics relate to potential similarity between files. The choice of these metrics was informed by previous studies:

- In [95], source code files were split into groups on the basis of an indicative set of metrics devised by Ding and Samadzadeh [44]. More specifically, the metrics in the indicative set were in the three categories of layout, style and structure (as discussed in chapter 2), and the content features were the number of lines, number of loops and code documentation. Krsul's study focused on purely structural issues rather than the semantic issues more relevant to plagiarism detection research.

- A study by Fitzgerald et al. [51] dealt with content analysis of a deeper kind more closely related to the focus of this research. Fitzgerald et al. identified several strategies to test the understanding of code by students. They presented a grounded theory-based synthetic analysis which identified 19 strategies used by students. This analysis operates at syntactic and semantic levels. The syntactic level reports many strategies related to comprehension of the program components such as: S1 (reading the question), S4 (previewing the code by identifying control structure), S6 (Pattern recognition), S14 (Elimination of specific choices of answers). The semantic level reports some semantic strategies that related to the meaning of the program such as: S5 (Understanding new concepts: semantic). In addition, Dasgupta [38] used 6 strategies from the above 19 strategies to investigate the relation between program comprehension and program authorship. Dasgupta used syntactic level strategies in their studies to find the authorship of the code.

Taking account of these two studies, metrics associated with two syntactic level strategies proposed by Fitzgerald et al. were adopted in order to find appropriate source files for further analysis:

1. Previewing the code by identifying control structure (as measured by the number of conditional statements);
2. Pattern recognition: outside knowledge (as measured by the number of loops).

These two metrics were considered to be the most compatible for the grouping study.

3.3.2.1 Experimental Methodology

This section summarises and illustrates the nature of the identified groups after pre-processing of all the files in the sample. The files were split into 5 groups as discussed below:

1. The first group contains files which are **template** or **common ground files** [46]. This means that the files are the same in terms of layout, style and structure, and that this is provided by the BlueJ IDE. Files such as these are usually not going to provide help in identifying similarity or detecting plagiarism, and we propose such files be ignored by plagiarism detection algorithms.

   For example, the code in Figure 3.5 shows most of the code lines are comment lines, which identify this file as a template file because the comments ask students to fill the gaps and build the code. For instance, line 28 which is "put your code here" and line 13 asks for a "Constructor for objects of class s". We found a substantial number of files from this group (62 out of 250 files in the first random sample).

2. The second group in our pre-processed sample consisted of **Short** code files. "Short" means the length of the code is less than 40 lines. In a Short file, the level of nesting in loops is typically less than 3. Less than 40 lines was pre-specified as the cut-off, as most code files with more than 40 lines have more complex levels of loop nesting. Most files in the sample were assigned to the Short group; the majority of the files in BlackBox are Short because the target of BlueJ is to teach students how to program Java. 84 out of 250 files in the first random sample were assigned to this group.

   As an example, Figure 3.6 shows a sample of code from the second group. Line 6 shows a print statement with request to "Enter a String". The level of

```java

/**
 * ##### # ########### ## ##### # ####.
 * @###### (#### ####)
 * @####### (# ####### ###### ## # ####)
 */
public class s
{
    // instance variables - replace the example below with your own
    private int x;

    /**
     * Constructor for objects of class s
     */
    public s()
    {
        // initialise instance variables
        x = 0;
    }

    /**
     * An example of a method - replace this comment with your own
     * @param y   a sample parameter for a method
     * @return    the sum of x and y
     */
    public int sampleMethod(int y)
    {
        // put your code here
        return x + y;
    }
}
```

Figure 3.5: Example Code of Group 1

the loop in this example has only very simple nesting, as in line 14 the `for` loop statement appears with an `if-else` condition inside the loop. The total length is 28 lines.

3. The third group in the sample consisted of **Intermediate** code files. The definition of Intermediate is that the length of the code is more than 40 lines and less than 100 lines. Regarding the structure of the code, the level of loops

```java
import java.util.*;
public class Kumar
{
    public static void main()
    {
     System.out.print("Enter␣a␣String:␣");
        String str=new Scanner(System.in).nextLine();
        str.trim();
        int l=str.length();
        String str1="KUMAR";
        String fin="";
        fin.trim();
        int c=0;
         for(int i=0;i<l;i++)
         {
             char ch=str.charAt(i);

             if(ch=='␣'&&c==0)
             {
                fin=fin+"␣"+str1+"␣";
                c++;
            }
            else
                fin=fin+ch;
        }
        System.out.print(fin);
    }
}
```

Figure 3.6: Example Code of Group 2

may be more than 3 loops and include some nested loops and conditions. This includes all the types of conditions and loops in Java: the conditions are `if`, `else`, `if-else` and `switch-case`, and the loops are `for` and `while-do`. In the first random sample, 50 files out of 250 were assigned to this group.

Figure 3.7, shows a fragment of code (starting at line 33) from group three. The do-loop beginning at line 37 and ending at line 50 uses `while`. It also features `try` and `catch` from lines 39-41, and `if` at line 46.

```

    public int leaEntero(String txt, int x, int y){
        boolean valorErroneo;
        int valor=0;
        do{
            valorErroneo=false; //asumo valor que dara el usuario es
                correcto
            try{
                valor= Integer.parseInt(this.showInputDialog(txt));
            }catch (NumberFormatException e){
                valorErroneo=true;
                this.showMessageDialog(null,"Se␣espera␣un␣valor␣entero",
                "VALOR␣INVALIDO", this.ERROR_MESSAGE);
            }
            if ((valor<x)||(valor>y))
            this.showMessageDialog(null,"Se␣espera␣un␣valor␣entero␣"+
            "entre␣"+x+"␣y␣"+y,
                "VALOR␣INVALIDO", this.ERROR_MESSAGE);
        }while ((valorErroneo)||(valor<x)||(valor>y));
        return valor;
    }
```

Figure 3.7: Example Code of Group 3

4. The fourth group consisted of code files which were **Long**. “Long” means that the length of the code is more than 100 lines . In a Long file, the looping and the level of the nesting in the loops is typically more complex than the third group. Such files contain rich data, and coding style based detection algorithms are likely to be successful when applied to them. In the first random sample, 34 files out of 250 were assigned to this group.

5. The fifth group contained files which were **incomplete** or **empty** files, and could thus be safely excluded by detection algorithms. For example, some file submissions are just test files with one line of code, or blank files without any line of code. In the first random sample, 20 files out of 250 were assigned to this group.

Since the first and the fifth group contain files which a detection algorithm can safely ignore, they have not been tested statistically in the next section.

3.3.3 The Results of the Evaluation of BlackBox

As discussed in the previous section, grouping of files gives us insight into the nature of the BlackBox dataset. This section is concerned with testing the hypothesis that: generating samples from BlackBox is a way of creating datasets with similar files that are useful as PD-testing datasets. Such similar files would be found within the Short, Intermediate and Long groups.

Statistical analysis of the grouping techniques applied to the BlackBox samples is a first step towards testing the above hypothesis Each random sample consisted of a total of 250 files divided into five groups, however the statistical analysis has been done only for groups two, three and four for the reasons mentioned in the previous section. Each sample and the main statistical calculations are presented in Tables 3.1, 3.2, 3.3 and 3.4 below. Each table shows the number of files falling into each group, and the mean, median, mode, standard deviation, maximum and minimum values for the numbers of lines. By definition, the Group 3 range (minimum to maximum) should be 40 to 100 lines, with Group 2 less and Group 4 more than these values.

The statistical analysis shows that the distribution of Short, Intermediate and Long files is consistent across all four samples, taking account of Mean, Median, Mode and Standard Deviation measures.

This analysis of the BlackBox dataset gives clear indications that some portions of the source code would be usable for coding style analysis studies aimed at plagiarism detection. Similar files in each random sample in the experimental study might be found in groups two, three and four; the distribution of files in each

Table 3.1: Random Sample (1) Results

	Group 2	**Group 3**	**Group 4**
Num. Files	84	50	34
Mean	24.64	67.50	195.79
Median	24.00	63.00	158.00
Mode	18	57	143
Std. Deviation	10.361	16.180	123.396
Maximum	40	99	673
Minimum	8	45	100

Table 3.2: Random Sample (2) Results

	Group 2	**Group 3**	**Group 4**
Num. Files	123	60	38
Mean	22.37	66.15	227.92
Median	22.00	62.00	164.50
Mode	13	51	260 accorss
Std. Deviation	8.968	19.741	147.035
Maximum	40	99	587
Minimum	6	41	102

Table 3.3: Random Sample (3) Results

	Group 2	**Group 3**	**Group 4**
Num. Files	140	41	38
Mean	22.89	65.54	249.45
Median	21.00	61.00	167.50
Mode	15	47	180
Std. Deviation	10.396	14.517	260.824
Maximum	40	99	1323
Minimum	5	46	100

Table 3.4: Random Sample (4) Results

	Group 2	**Group 3**	**Group 4**
accorss Num. Files	124	61	40
Mean	22.00	63.90	192.65
Median	19.00	61.00	170.50
Mode	19	55	142
Std. Deviation	9.608	14.830	102.966
Maximum	43	100	649
Minimum	7	41	106

group in each random sample were similar. Furthermore, the mean, the median and the mode in random samples 1-4 were similar to each other. The analysis also suggests that the random samples are each representative of the files contained in the BlackBox dataset.

The maximum and the minimum values in the tables 3.1, 3.2, 3.3 and 3.4 refer to the number of lines in each group. Group 4 contained (by definition) the highest number of lines, giving more scope for style analysis in this group. The results show that in the original numbering, groups 2-4 (excluding groups 1 and 5) might offer a rich source to which coding style analysis can be applied for the purpose of detecting plagiarism.

3.3.4 Visualisation

To complete the testing of the hypothesis framed in section 3.3.2, it remains to determine whether BlackBox can serve not only as a source of datasets containing files with similar characteristics (such as the grouping technique constructs in Groups 2, 3 and 4) but of datasets that contain files that exhibit similarities characteristic of plagiarism. To this end, the datasets studied in the previous section will be analysed using alternative methods for source code plagiarism detection.

All of the methods operate by calculating the degree to which two pieces of work are similar, and end with producing a report. The report will usually show two sections of code sufficiently similar that plagiarism is a possibility. It is then the teacher's task to examine the coding "pairs" to decide whether plagiarism really is evident. Depending on the system, the report may be a table, a graph, or merely a simple list. In this research Gephi is the measuring tool that is used to identify plagiarism by tracking visual clustering.

Gephi is an interactive tool for visualising and exploring complex networks and systems, providing a visual analysis. Gephi is usable for a variety of applications including: link analysis; exploration of data analysis; social network analysis; and biological network analysis. Gephi's unique feature is its ability to analyse data using different machine learning algorithms. In this case, data are presented as graphs depicting collaboration networks based on similarity between source code sets. Visualisation is today the normal approach, as can be seen in Makuc [102] where the force-directed graph and co-occurrence matrix provide a visual view of similarities. For the most part, visualisation is an intuitive process, though some visualisation requires practice. Freire [56] observes:

> "An over-simplification of the interface, however, would be dangerous, since graders may decide to 'let the system do the work' instead of mak-

ing informed decisions based on the actual submissions."

As mentioned, this research focuses on visualisation through the use of collaboration networks and employs visualization methods that operate on all submissions. Examples include: Sherlock [79]; SSID [120]; CoMoTo [107]; and Misic et al.[109]. Luquini and Omar [101] were the first to create social networks of students in a project. They then carried out plagiarism detection on those students' assignments (undertaken at home or in the classroom). What they established was that there was a greater correlation with a student's preferred working partner in assignments carried out at home. It may be that this is informative about relationships on the basis of pair similarity; it may also be a sign that collusion is more likely between students who are friends than between those who are merely classmates. The latter possibility would add force to the arguments for improving plagiarism detection by analysing networks.

3.3.4.1 Experimental Methodology

To complete the testing of the hypothesis under consideration, the following experimental methodology was adopted. In the exploratory study of the BlackBox dataset, the files in groups 2, 3 and 4 of the sample were extracted from the dataset as a CSV file and the nodes and edges imported to Gephi which produces a graphical visualisation of the results. In this context, Gephi identifies a collaboration network, from here on referred to only as the network, on which clustering algorithms are performed. The principle at work here is the one described by Misic et al. [109]. The Sherlock plagiarism detection tool was used to calculate similarity, and produced some results in the form of paired files, which were then checked manually for similarity characteristic of plagiarism. Further indications of similarities were inspected after importing the data into Gephi.

Gephi's visualisation graphs did not indicate any cluster or similarity. In this case, Gephi has some algorithms which can be applied, such as the Chinese Whispers (CW) clustering approach. Most of the previous studies [16] [43] [124] [66] use CW in Natural Language Processing (NLP). Chinese Whispers is a clustering method used in network science. This method is capable of very quickly identifying communities in any network. Therefore, Chinese Whispers is a good method in analysing community structure in graphs with a very high number of nodes. Random sample 1 was chosen from the BlackBox for this experiment to find files that had similarities.

3.3.4.2 The Results

Figure 3.8 shows the result of applying CW to the BlackBox dataset; two main clusters are observed to display significant similarities. As the two clusters contained a number of files (14 in the first and 23 in the second), anything with a similarity below 30% was excluded (Figure 3.9). It then became clear that three clusters existed, one with four files (blue in the Figures), one with three (red in the Figures) and one with two (purple in the Figures).

In the first cluster, three files showed a similarity greater than 65%. These files contained almost identical code, but each named the variables differently. This difference reduced the percentage similarity. Figure 3.8 shows a relationship between these files and some others (those found in the original cluster); however, similarity with those files was 30% and a manual check revealed no plagiarism. There were four similar files in the second cluster, but the code was different from that in first clusters files. Similarity for each pair was 85%. This second cluster is illustrated in Figure 3.9; all of the files were plagiarised. Two files in the third cluster had 80% similarity; these, too, were plagiarised. Figure 3.9 shows them clearly.

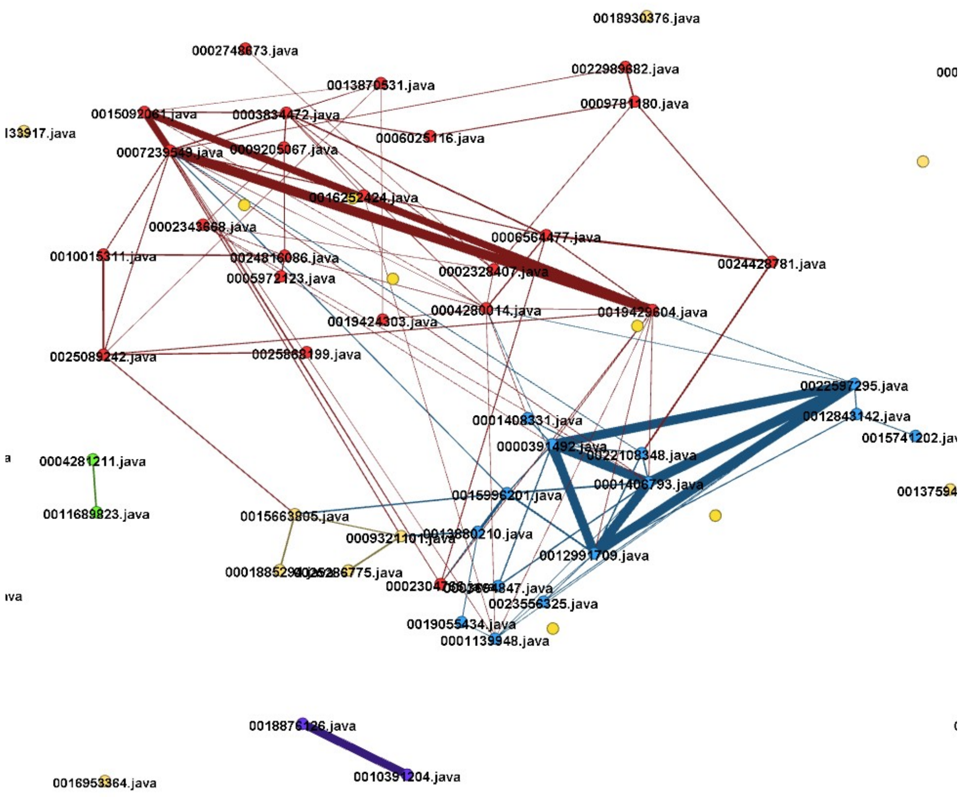


Figure 3.8: Initial network display for BlackBox dataset with the Chinese Whispers algorithm

A number of different similarity combinations were explored, but no plagiarised cases were found apart from those already mentioned, which also have the highest similarity percentage. Figure 3.10 shows the same three clusters as visualised in Sherlock. The results given by Sherlock were confirmed using Gephi and no new areas of interest emerged. The advantage of Gephi is that its representation is simpler and it has better options for manipulating similarities and combinations of clustering algorithms.

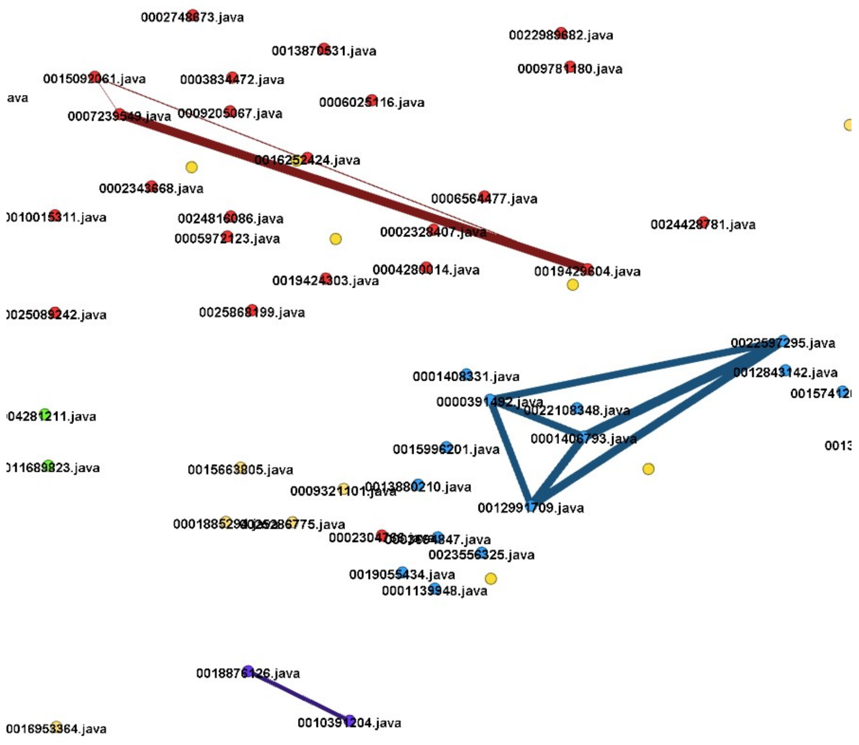


Figure 3.9: BlackBox network with performed Chinese whispers algorithm and 30%-97% similarity

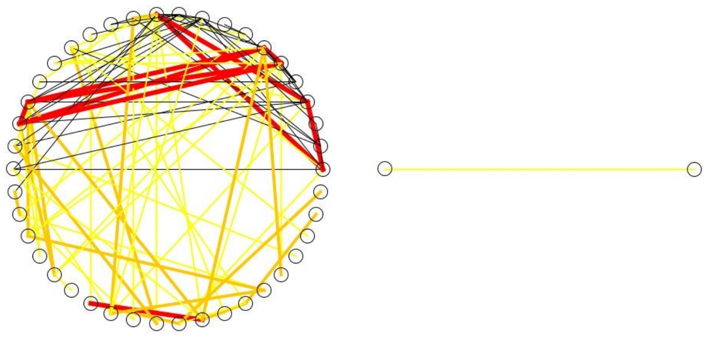


Figure 3.10: The Sherlock network display for BlackBox dataset

3.4 Conclusion

The main research question was “Can the BlackBox dataset be used as a source of datasets suitable for testing methods of plagiarism detection?”. The exploratory study went through different stages, starting from requesting access from Kent University and discovering the nature of the source code that had been submitted. BlackBox is a large dataset and the process of sample selection was a difficult task. Sample size determination was the initial step before proceeding to download the random samples. The size of the random sample from huge number of files was determined as 250 files. Specific criteria for fetching the files were identified, taking account of the file IDs, and a small program was implemented for this purpose. The sample had to be pre-processed to eliminate irrelevant files unsuitable for similarity detection. A grouping technique was applied to the sample to allocate the remaining files to groups according to their length. In this way, each of the samples was split into 5 groups: group 1 is called Template or common ground, group 2 called Short, group 3 called Intermediate, group 4 called Long and group 5 called Incomplete. As the result of this process of classification, the files most appropriate for testing for plagiarism were those in groups 2, 3 and 4, and those in groups 1 and 5 could be discarded.

At this stage, the exploratory study indicated that the BlackBox dataset contains many similar files. To complete the study, it remained to determine whether BlackBox contains files with similarities characteristic of plagiarism. To this end, the study exploited the visualisation approach to detecting plagiarism using the Gephi visualisation tool. Gephi was applied to one of the four random samples of BlackBox to find similar files that might have been plagiarised. Gephi has some embedded clustering algorithms. The Chinese Whispers (CW) method was used to detect relevant similarities between files using clustering.

The results of the study revealed that although some similar files were detected in BlackBox, their similarity was not in general so significant that the files could be deemed to have been plagiarised. Furthermore, the number of plagiarised file pairs/groups were not enough to thoroughly evaluate the performance of similarity detection methods with a view to detecting plagiarism.

Chapter 4

Structure Based Metrics

4.1 Introduction

Coding Style metrics can be used as an effective approach for detecting authorship. Ding and Samadzadeh [44] integrated different metrics found in the literature and proposed a consolidated set of coding style metrics for detecting authorship in source code files in Java. Detecting similarity in coding style can also be considered as a technique for detecting plagiarism. This chapter proposes extended variants of the approach of Ding and Samadzadeh [44], which focused on authorship detection, in order to make it suitable for plagiarism detection. The chapter addresses the research objectives OB4-6, including re-implementation of coding style metrics, modifying metrics to be compatible with Java, and extending the approach with new metrics to enhance plagiarism detection.

- **OB4**) To re-implement existing coding style metrics.
- **OB5**) To modify existing coding style metrics to be compatible with the Java programming language.

- **OB6**) To propose and extend new metrics to enhance the output from the implementation.

This chapter is outlined as follows. Section 4.2 introduces Ding et al existing work [44] and explains how their metrics could be suitable for the task of source-code plagiarism detection. Section 4.3 introduces the first variant family of metrics, i.e. Modified Style Metrics, which contains the existing metrics and minor modification on them in order to be more reliable for detecting plagiarism in Java programming. Section 4.4 discusses the second proposed family of metrics, i.e. Extended Style Metrics, which builds upon the Modified Metrics, and adds new metrics based on the author's programming experience.

4.2 Existing Style Metrics

The main coding style metrics used in this chapter were derived from Krsul and Spafford [95]. Krsul and Spafford's coding style metrics were designed to be used in the analysis of C programs; Ding and Samadezadeh [44] subsequently transformed the coding style metrics from C to Java compatibility. The metrics are divided into three categories [95], each category has main metrics and sub-metrics. In summary:

- Layout metrics: these are utilised to measure the similarity of the layout of the files. Layout metrics measure similarities in such things as indentation, type of comments, and white space before and after the brackets.

- Style metrics: these are utilised to measure the similarity of the style of the files. Style similarities include mean variable length, mean comment length and preference of condition stems.

- Structure metrics: these are utilised to measure the similarity of the structure of the files. Structure similarities include keywords within the class and the interfaces and the usage of data structure.

The codes used for the 3 families of metrics (STY for Layout, PRO for Style and PSM for Structure) are as taken from Krsul and Spafford's original paper [95]. Those codes refer to the different classes of metric and submetric:

1. Layout Metrics: **STY**(nm);
2. Style Metrics: **PRO**(nm);
3. Structure Metrics: **PSM**(nm).

The number of metrics n and submetrics m in each category are as shown in Table 4.1 below. There are 8 main metrics in the Layout category, 5 main metrics in the Style category and 7 main metrics in the Structure category. The Layout metrics are represented as STY1, STY2, .., STY8; the Style metrics as PRO1, PRO2,.., PRO5 and the Structure metrics as PSM1, PSM2, .., PSM7.

Table 4.1: Number of Metrics and submetrics: STY, PRO and PSM as in [44]

	Metric (n)	Submetric(m)
STY	8	10
PRO	5	13
PSM	7	22

In Table 4.1, n represents the number of types of metric in each category and m represents the number of sub-metrics of each type, to be identified by small letters $a \rightarrow z$. There are 10 submetrics in the Layout category, 13 in the Style category and 22 in the Structure category. For example, in the Layout metrics, the metrics of the first type STY1 consist of 8 sub-metrics represented by small letters: STY1a, STY1b, STY1c, STY1d, STY1e, STY1f, STY1g and STY1h. In addition, each sub-metric is described by features that relate to the main metric. For example, in the Style category, PRO2 refer to metrics associated with name length, and the 2 sub-metrics under this are PRO2a, which calculates mean variable name length, and PRO2b, which calculates mean function name length.

The study of coding style metrics for plagiarism detection carried out in this thesis began with the re-implementation of existing style metrics using the Python programming language. In the process of re-implementation, there was a need for a modification for some metrics in each category to reflect the analysis of coding style that is most appropriate for detecting plagiarism rather than identifying authorship [44]. The next subsections 4.3.1, 4.3.2 and 4.3.3 explain how the existing approach by Ding and Samadzadeh [44] was modified to make it suitable for detecting plagiarism rather than identifying authorship.

4.3 Modified Style Metrics

The Modified metrics extend the metrics proposed by Ding and Samadzadeh [44]. Modifications were made to the Layout, Style and Structure metrics as described in the subsections that follow.

4.3.1 Modified Layout Metrics (STY)

Table 4.2: **Modified Layout** metrics extracted from the source code of Java programs

Metric	Description
STY1	A list of metrics indicating indentation style
STY1a	Percentage of open braces ({) that are along a line
STY1b	Percentage of open braces ({) that are the first character in a line
STY1c	Percentage of open braces ({) that are the last character in a line
STY1d	Percentage of close braces (}) that are along a line
STY1e	Percentage of close braces (}) that are the first character in a line
STY1f	Percentage of close braces (}) that are the last character in a line
STY1g	Average indentation in white spaces after open braces ({)
STY1h	Average indentation in tabs after open braces ({)
STY2	A list of metrics specifying comment style
STY2a	Percentages of pure comment lines among lines containing comments
STY2b	Percentages of "//" style comments among "//" and "/*" style comments
STY3	Percentages of condition lines where the statements are on the same line as the condition. searching for `if`, `if-else` and `switch-case`
STY4	Average white spaces to the left side of operators
STY5	Average white spaces to the right side of operators
STY6	Ratio of blank lines to code lines (including comment lines)
STY7	Ratio of comment lines to non-comment lines
STY8	Ratio of code lines containing comment to code lines without any comments

These metrics consider the layout of the code. The features described in this category are important as they can be used as a basis for building the underlying infrastructure. Table 4.2 shows the layout metrics, denoted STY1-STY8. The STY metrics category was the first one to be re-implemented as it is the basis of the other metrics in the second and third categories. There are 8 main metrics and 10 sub

```
 1 import java.util.Scanner;
 2
 3 public class Main{
 4  public static void main(String[] args) {
 5   Scanner in= new Scanner(System.in);
 6   int T= in.nextInt(); int n,t,s,answer=0;
 7   for (int i = 0; i < T; i++) {
 8    n=in.nextInt(); t=in.nextInt(); s=in.nextInt();
 9    for (int j = 0; j < n; j++) {
10     int x=in.nextInt(); int nashti=x%3;
11     if(x==0 && s!=0) continue;
12     if(x/3>=s){ answer++; continue;}
13     if(x/3==(s-1)){
14      if(nashti>0){answer++; continue;}
15      if(t>0){t--; answer++; continue;}
16     }
17     if(x/3==(s-2)){
18      if(nashti>=2 && t>0){t--; answer++; continue;}
19     }
20    }
21    System.out.println("Case␣#"+(i+1)+":␣"+answer);
22    answer=0;
23   }
24  }
25 }
```

Figure 4.1: Java example: showing the STY3 metric in Modified Layout metrics

metrics in this category, focusing on the layout of the code, such as: indentation style, including open and close braces in different cases, and indentation using white spaces and tabs in code lines. The red colour in the table 4.2 shows our modified layout metrics compared to the existing layout metrics which are shown in black.

STY3 calculates the percentage of condition lines with the statement on the same line as the condition. This involves calculating and checking only condition lines where the `if`, `if-else` and `case-switch` keywords are on the same line as the statement. An example is shown in Figure 4.1; there are 7 condition lines which contain `if` in (11, 12, 13, 14, 15, 17 and 18); the re-implementation takes this file and tracks the condition lines (7 lines; expressed as a percentage of the total file lines).

Looking at another example of layout metrics: **STY7** calculates the ratio of comment lines to non-comment lines. This involves calculating and checking the comment lines based on the metrics STY2 which take a vector of metrics specifying comment style and the sub metrics STY2a (percentage of pure comment line among line containing comments) and STY2b (percentage of "//" style comments among "//" and "/*" style comments). The calculation for STY7 takes the number of lines containing any kind of comment as a ratio to the non-comment lines. The rest of the Layout metrics are described in Table 4.2.

4.3.2 Modified Style Metrics (PRO)

The PRO metrics in this category concern the style of the code in terms of many aspects such as lengths of lines (number of characters), variables and functions. These metrics are presented in Table 4.3. For example, the metrics look at whether a character is lowercase or uppercase; whether underscores or dollar signs have been used; and the conditions (`if`, `else` and `switch-case`) and the loops (`while`, `for` and `do`) used.

PRO2a is shown in Table 4.3, the variables considered in this category are only nine common data types in both the existing approach and the modified approach. However, these types of data are upper and lower case sensitive, the change here applied to `string` and `boolean` data types; the upper case and lower case character both are checked in this category (see Table 4.8).

PRO5 is about calculating the preference of either `if-else` or `switch-case` conditions, so there are four sub-metrics under PRO5: PRO5a, PRO5b, PRO5c and PRO5d. There have been small changes in PRO5a and PRO5b.

1. **PRO5a** calculates the percentage of the `if` and `else` out of the total of `if`,

Table 4.3: **Modified Style** metrics extracted from the source code of Java programs

Metric	Description
PRO1	Mean Program line length in terms of characters
PRO2	A vector of metrics of name lengths
PRO2a	Mean variable name length Changed string to String and boolean to Boolean
PRO2b	Mean function name length
PRO3	Character preference of uppercase, lowercase, underscore, or dollar sign for name convention.
PRO3a	Percentage of uppercase characters
PRO3b	Percentage of lowercase characters
PRO3c	Percentage of underscores
PRO3d	Percentage of dollar signs
PRO4	Preference of either `while`, `for` or `do` loops
PRO4a	Percentage of `while` in total of `while`, `for` and `do`
PRO4b	Percentage of `switch` in total of `while`, `for` and `do`
PRO4c	Percentage of `do` in total of `while`, `for` and `do`
PRO5	Preference of either `if-else` or `switch-case` conditions
PRO5a	Percentage of `if` and `else` in total of `if`, `else`, `switch`, and `case` Percentage of `if`, `if-else` in total of `if`, `if-else` and `switch-case`
PRO5b	Percentage of `switch` and `case` in total of `if`, `else`, `switch`, and `case` Percentage of `switch-case` in total of `if`, `if-else` and `switch-case`
PRO5c	~~Percentage of `if` in total of `if` and `else`~~. Discarded.
PRO5d	~~Percentage of `switch` in total of`switch` and `case`~~. Discarded.

`if-else` and `switch-case` in the existing approach as shown in the table. The change here in the modified metrics was to calculate instead the percentage of `if` and `if-else` out of the total of `if`, `if-else` and `switch-case`. This is a very simple modification but it does affect the percentage extracted.

2. In **PRO5b**, the total used for the percentage of `switch-case` was adjusted to be the total of `if`, `if-else` and `switch-case`.

3. **PRO5c** and **PRO5d** have been discarded because they have already been calculated in the two previous metrics **PRO5a** and **PRO5b** (see Table 4.3).

4.3.3 Modified Structure Metrics (PSM)

Table 4.4: **Modified Structure** metrics extracted from the source code of Java programs

Metric	Description
PSM1	Average non-comment lines per class/interface
PSM2	Average number of primitive variables per class/interface Primitive variables are the ones used in **PRO2a**
PSM3	Average number of functions per class/interface
PSM4	Ratio of interfaces to classes This metrics divided into 2
PSM4a	Percentage of interfaces in total of interfaces and classes
PSM4b	Percentage of classes in total of interfaces and classes
PSM5	Ratio of primitive variable count to lines of non-comment code
PSM6	Ratio of function count to lines of non-comment
PSM7	A list of ratios of keywords to lines of non-comment code
PSM7a	Ratio of keyword `static` to lines of non-comment code
PSM7b	Ratio of keyword `extends` to lines of non-comment code
PSM7c	Ratio of keyword `class` to lines of non-comment code
PSM7d	Ratio of keyword `abstract` to lines of non-comment code
PSM7e	Ratio of keyword `implements` to lines of non-comment code
PSM7f	Ratio of keyword `import` to lines of non-comment code
PSM7g	Ratio of keyword `instanceof` to lines of non-comment code
PSM7h	Ratio of keyword `interface` to lines of non-comment code
PSM7i	Ratio of keyword `native` to lines of non-comment code
PSM7j	Ratio of keyword `new` to lines of non-comment code
PSM7k	Ratio of keyword `package` to lines of non-comment code
PSM7l	Ratio of keyword `private`to lines of non-comment code
PSM7m	Ratio of keyword `public` to lines of non-comment code
PSM7n	Ratio of keyword `protected` to lines of non-comment code
PSM7o	Ratio of keyword `this` to lines of non-comment code
PSM7p	Ratio of keyword `super` to lines of non-comment code
PSM7q	Ratio of keyword `try` to lines of non-comment code
PSM7r	Ratio of keyword `throw` to lines of non-comment code
PSM7s	Ratio of keyword `catch` to lines of non-comment code
PSM7t	Ratio of keyword `final` to lines of non-comment code

The PSM metrics in this category, as presented in Table 4.4, are concerned with the logic of the code. Modifications in this category were to **PSM2** and **PSM4**. The metric **PSM2** calculates the primitive variables without any specification and the addition is to tag them as the variables used in **PRO2a**.

Moreover, **PSM4** calculates the ratio of interfaces to classes without any specification in the fingerprint approach, whereas in the modified approach, this metric has now been divided into 2 new metrics: **PSM4a**, which calculates the percentage of interfaces in the total of interfaces and classes, and **PSM4b**, which calculate the percentage of classes in the total of interfaces and classes [1].

To extract coding style features from the Java code, additional criteria need to be considered:

- **Comments:**

  Most of the files have some kind of programming language comments to describe the code or add some information about the code. In this research, Java files have two kinds of comments and experts usually write comments to add more clarification or explain how the code works. In this case, some words of the code such as any condition, operation, or even white space, may be repeated in the comments, which would definitely affect the final result of the metrics. To address this issue, the re-implementation processes the file as if there were no comments in the code by stripping the comments from the code when required for some analyses.

  For example, when it comes to detecting **STY2**: A list of metrics specifying comment style, and STY2a: calculate the percentage of pure comment lines among lines containing comments and the STY2b: calculate percentage of "//" style comments among "//" and "/*" style comments: the code remains as it is without any stripping. However, the comments are stripped from the code when calculating the other metrics. In contrast, for the purpose of computing certain metrics, such as **STY6-8**, removing all comments from the

[1]The value of the metric PSM4b can be inferred from that of the metric PSM4a, rendering it redundant as a metric. However, the use of both metrics has a material impact on the matrix using in the MFM technique to be introduced in Chapter 5. There are other instances of such redundancy in the variants of metrics introduced in this chapter, cf. the metric STY3a

code is not deemed to affect the number of lines in the file. Calculating each metric uses the same input, apart from removing the comments from the code.

- **Operations:**

Table 4.5: The differences between the operations in the Modified metrics and the Extended metrics

	Existing		Modified-Extended		
Operations	+	+=	+	++	+=
	-	-=	-	-=	||
	*	*=	*	*=	|=
	/	/=	/	/=	|
	%	%=	%	%=	&&
	=	==	=	==	instanceof
			>>	<<	>>>
			>	>=	>>=
			<	<=	<<=
			!	!=	^=
			a&b	a|b	a^b
			∼a		

Metrics **STY4** and **STY5** calculate the average amounts of white space to the left side and right side of the operators. The Existing approach considers just 12 operators based on what Ding and Samadezadeh [44] used in their research, whereas there are a total of 34 operators available in the Java language ,as shown in Table 4.5.

- **Variables:**

Defining the variables in the re-implementation is an essential stage; the variables were limited in the Existing approach to: short, int, long, float, double, byte, char, boolean and string. The list was expanded to cover all variables covered in the Java language. The new list included all the previous variables and the new variables as follows: Short, Integer, Long, Float, Double, Byte, Character, Boolean and String.

The next section focuses on addressing the research objective OB 6 as set out in the introduction in this chapter, viz. to propose and extend new metrics to enhance the output from the implementation. As will be seen the Modified and the Extended metrics play key role in the formulation of enhanced methods of plagiarism detection.

4.4 Extended Style Metrics

One of the functional requirements for the re-implementation is to have better accuracy in terms of plagiarism detection and find similar files in a big dataset. The Modified Metrics presents the existing metrics with some modest modifications in addition to Ding's existing metrics [44]. In this section some metrics in each category have been extended if they were not covered in the existing or modified metrics. Based on my Java programming experience, I proposed and examined several extensions in the three existing metrics categories: Layout, Style and Structure that have been used in Ding et. al [44]. The extended metrics in each table are written in blue. The extended metrics proposed for more enhanced results in detection, are presented in the next three subsections.

4.4.1 Extended Layout Metrics (STY)

For STY category, most of the metrics remained the same. However, **STY3**, which calculates the percentage of the condition line when the statements are on the same line as the condition, was split into 2 new metrics. The new metric STY3a calculates the percentage of condition lines of (`if`, `else`, `if-else` and `case-switch`) where the statements are on the same line as the condition. On the other hand, STY3b calculates the percentage of condition lines of (`if`, `else`, `if-else` and `case-switch`) where the statements are on a different line from the condition. The value of the metric STY3b can be inferred from that of the metric STY3a, rendering it redundant as a metric. However, the use of both metrics has a material impact on the matrix using in the MFM technique to be introduced in Chapter 5. There are other instances of such redundancy in the variants of metrics introduced in this chapter, cf. the metric PSM4a.

Table 4.6: **Extended Layout** metrics extracted from the source code of Java programs

Metric	Description
STY1	A list of metrics indicating indentation style
STY1a	Percentage of open braces ({) that are along a line
STY1b	Percentage of open braces ({) that are the first character in a line
STY1c	Percentage of open braces ({) that are the last character in a line
STY1d	Percentage of close braces (}) that are along a line
STY1e	Percentage of close braces (}) that are the first character in a line
STY1f	Percentage of close braces (}) that are the last character in a line
STY1g	Average indentation in white spaces after open braces ({)
STY1h	Average indentation in tabs after open braces ({)
STY2	A vector of metrics specifying comment style
STY2a	Percentages of pure comment lines among lines containing comments
STY2b	Percentages of "//" style comments among "//" and "/*" style comments
STY3	Percentages of condition lines where the statements are on the same line as the condition. ~~searching for `if`, `else`, `if-else` and `case-switch`~~. This metric divided into 2
STY3a	Percentage of condition lines where the statement is on the same as the condition
STY3b	Percentage of condition lines where the statement is on a different as the condition
STY4	Average white spaces to the left side of all operators
STY5	Average white spaces to the right side of all operators
STY6	Ratio of blank lines to code lines (including comment lines)
STY7	Ratio of comment lines to non-comment lines
STY8	Ratio of code lines containing comment to code lines without any comments

4.4.2 Extended Style Metrics (PRO)

The PRO category covers the aspect of style of code as discussed in section 4.3.2. Most of the metrics in the extended style metrics remain the same, however, there is a slight change in this category in the **PRO2** main metric.

The submetric PRO2a now includes datatype names with uppercase and lowercase characters; they include datatype names with capitalised initial letter (e.g. `byte` / `Byte`) and unabbreviated names (e.g. `int` / `Integer`).

The other metric affected in the proposed extended set was **PRO3**, which calculates character preferences for uppercase, lowercase, underscore, or dollar sign for

Table 4.7: **Extended Style** metrics extracted from the source code of Java programs

Metric	Description
PRO1	Mean Program line length in terms of characters
PRO2	A vector of metrics of name lengths
PRO2a	Mean variable name length Changed string to String and boolean to Boolean Add new data types: `byte`, `short`, `char`, `int`, `long`, `float`, `boolean`, `double` `Byte`, `Short`, `Character`, `Integer`, `long`, `Float`, `Boolean`, `Double`, `String`
PRO2b	Mean function name length
PRO3	Character preference of uppercase, lowercase, underscore, or dollar sign for name convention.
PRO3a	Percentage of uppercase characters
PRO3b	Percentage of lowercase characters
PRO3c	Percentage of underscores
PRO3d	Percentage of dollar signs
PRO3e	Percentage of symbols other than those considered in PROa, PROb, PROc and PROd
PRO4	Preference of either `while`, `for` or `do` loops
PRO4a	Percentage of `while` in total of `while`, `for` and `do`
PRO4b	Percentage of `for` in total of `while`, `for` and `do`
PRO4c	Percentage of `do` in total of `while`, `for` and `do`
PRO5	Preference of either `if-else` or `switch-case` conditions
PRO5a	Percentage of `if` and `else` in total of `if`, `else`, `switch-case` Percentage of `if` and `if-else` in total of `if`, `if-else` and `switch-case`
PRO5b	Percentage of `switch-case` in total of `if`, `else`, `switch-case` Percentage of `switch-case` in total of `if`, `if-else` and `switch-case`
PRO5c	~~Percentage of `if` in total of `if` and `else`~~. Discarded.
PRO5d	~~Percentage of `switch` in total of `switch` and `case`~~. Discarded.

name convention. **PRO3e** calculate the percentage of symbols other than those considered in PRO3a, which calculates the percentage of uppercase characters; PRO3b, which calculate the percentage of lowercase characters); PRO3c, which calculates the percentage of underscores or PRO3d, which calculates the percentage of dollar signs.

Table 4.8: **Extended Structure** metrics extracted from the source code of Java programs

Metric	Description
PSM1	Average non-comment lines per class/interface
PSM2	Average number of primitive variables per class/interface Primitive variables are the ones used in **PRO2a**
PSM3	Average number of functions per class/interface
PSM4	Ratio of interfaces to classes This metrics divided into 2
PSM4a	Percentage of interfaces in total of interfaces and classes
PSM4b	Percentage of classes in total of interfaces and classes
PSM5	Ratio of primitive variable count to lines of non-comment code
PSM6	Ratio of function count to lines of non-comment
PSM7	A list of ratios of keywords to lines of non-comment code
PSM7a	Ratio of keyword `static` to lines of non-comment code
PSM7b	Ratio of keyword `extends` to lines of non-comment code
PSM7c	Ratio of keyword `class` to lines of non-comment code
PSM7d	Ratio of keyword `abstract` to lines of non-comment code
PSM7e	Ratio of keyword `implements` to lines of non-comment code
PSM7f	Ratio of keyword `import` to lines of non-comment code
PSM7g	Ratio of keyword `instance of` to lines of non-comment code
PSM7h	Ratio of keyword `interface` to lines of non-comment code
PSM7i	Ratio of keyword `native` to lines of non-comment code
PSM7j	Ratio of keyword `new` to lines of non-comment code
PSM7k	Ratio of keyword `package` to lines of non-comment code
PSM7l	Ratio of keyword `private` to lines of non-comment code
PSM7m	Ratio of keyword `public` to lines of non-comment code
PSM7n	Ratio of keyword `protected` to lines of non-comment code
PSM7o	Ratio of keyword `this` to lines of non-comment code
PSM7p	Ratio of keyword `super` to lines of non-comment code
PSM7q	Ratio of keyword `try` to lines of non-comment code
PSM7r	Ratio of keyword `throw` to lines of non-comment code
PSM7s	Ratio of keyword `catch` to lines of non-comment code
PSM8	Character preference of uppercase, lowercase, underscore, or dollar sign for name convention
PSM8a	Ratio of uppercase characters for classes and interfaces only
PSM8b	Ratio of lowercase characters for classes and interfaces only
PSM8c	Ratio of underscores for classes and interfaces only
PSM8d	Ratio of dollar signs for classes and interfaces only
PSM8e	Anything not covered in PSM8a, PSM8b, PSM8c and PSM8d
PSM9	Preference of either `while`, `for` or `do` loops
PSM9a	Ratio of `while` in total of `while`, `for` and `do` for classes and interfaces only
PSM9b	Ratio of `for` in total of `while`, `for` and `do` for classes and interfaces only
PSM9c	Ratio of `do` in total of `while`, `for` and `do` for classes and interfaces only
PSM10	Preference of either `if-else` or `switch-case` conditions
PSM10a	Ratio of `if` and `if-else` in total of `if`, `if-else` and `switch-case` for classes and interfaces only
PSM10b	Ratio of `switch-case` in total of `if`, `if-else` and `switch-case` for classes and interfaces only

4.4.3 Extended Structure Metrics (PSM)

The PSM category plays a very important role at this stage, as there were some new additions which significantly enhance the power of the plagiarism techniques to be derived from theses metrics in chapter 5 . The extensions are in 3 main metrics: PSM8, PSM9 and PSM10; each of them is divided into several sub-metrics. All the extensions are calculated for specific classes and interfaces only, whereas all the rest of the metrics are calculated in terms of the overall file code.

Firstly, **PSM8** is about character preference of uppercase, lowercase, underscore or dollar sign for name convention, and it is divided into 5 sub-metrics: PSM8a, PSM8b, PSM8c,PSM8d and PSM8e. First, **PSM8a** calculates the ratio of uppercase characters for class and interfaces only. Second, **PSM8b** calculates the ratio of lowercase characters for classes and interfaces only. Third, **PSM8c** calculates the ratio of underscores for classes and interfaces only. Fourth, **PSM8d** calculates the ratio of dollar signs for classes and interfaces only. Last, **PSM8e** calculates everything not covered in PSM9a, PSM9b, PSM9c or PSM9d.

Secondly, **PSM9** is about preferences of either `while`, `for` or `do` loops and it is divided into 3 sub-metrics: PSM9a, PSM9b and PSM9c. First, **PSM9a** calculates the percentage of `while` loops in the total of `while`, `for` and `do` loops for classes and interfaces only. Second, **PSM9b** calculates the percentage of `for` loops in the total of `while`, `for` and `do` loops for classes and interfaces only. Third, **PSM9c** calculates the percentage of `do` loops in the total of `while`, `for` and `do` loops for classes and interfaces only.

Thirdly, **PSM10** is about preference of either `if-else` or `switch-case` conditions and it is divided into 2 sub-metrics: PSM10a and PSM10b. First, **PSM10a** calculates the percentage of `if`, `if-else` conditions in the total of `if`, `if-else` and `switch-case` conditions for classes and interfaces only. Second, **PSM10b** calculates the percentage of `switch-case` conditions in the total of `if`, `if-else` and `switch-case` statements for classes and interfaces only.

4.5 Example

The Java example in the Appendix to this chapter shows a sample of Java code from SOCO dataset from group A1. The example is used to illustrate the results of calculation simply presented in three categories (Layout, Style and Structure) as required for each metric. This section shows the output calculation for both the modified and extended approaches. The differences in the output will be discussed for each metric where the extended approach picks up something different from the modified approach.

4.5.1 Metrics Calculation

In each metric in each category, the main calculations are: the percentage, the average or the ratio of items in the source code file. How these are calculated is shown below: Table 4.2 shows which metric is used for each subcategory of the Layout Metrics; Table 4.3 for the Style Metrics and Table 4.4 for the Structure Metrics.

1. **The Average:** The most common type of average is the arithmetic mean (AM). If n numbers are given, each number denoted by a_i (where i = 1,2,..., n), the arithmetic mean is the sum of the a's divided by n or:

$$AM = \frac{1}{n}\sum_{i=1}^{n} a_i = \frac{1}{n}(a_1 + a_2 + ... + a_n) \tag{4.1}$$

2. **The Percentage:** The value of x as a percentage of y is computed as:

$$P = (x/y) * 100 \tag{4.2}$$

3. **The Ratio:** A ratio compares the number of instances of one entity with that of another.

The next section presents the Modified metrics and the Extended metrics with their submetrics's calculation. The calculation of metrics has been implemented using the Python programming language.

Layout Metrics Calculation

The total number of open braces in the Java example code in the Appendix to this chapter is 22 open braces (in lines 11, 16, 21, 25, 27, 29, 31, 39, 62, 65, 68, 70, 73, 75, 83, 86, 91, 94, 96, 116, 117 and 122).

Number of open braces (OB)= 22

- STY1a calculates the percentage of open braces ({) which are within a line. There are 4 such braces (lines 25, 27, 29 and 31), so the result is 4/OB = 18.1818%.

- STY1b calculates the percentage of open braces ({) that are the first character in a line. There are 0 such braces, so the result is 0/OB = 0% .

- STY1c calculates the percentage of open braces ({) that are the last character in a line. There are 18 such braces, so the result is 18/OB =81.8181%.

A similar calculation can be performed for close braces to calculate STY1d, STY1e and STY1f.

- STY1g calculates the average indentation in white space after open braces ({), the average is 0.
- STY1h calculates the average indentation tabs after open braces ({), the average is 0.

STY2 : A vector of metrics specifying comment style.

- STY2a calculates the percentage of pure comment lines among lines containing comments. There are 7 pure comment lines (23, 87, 110, 111, 112, 113, 114) and a further 3 in line comments (44, 45, 46), giving a percentage of 70%.
- STY2b calculates the percentage of "//" style comments among "//" and "/*" style comments. The total number of "//" is 5 and "/*" is 1, so the total number of both comments is 6, and the calculated percentage is 5/6=83.3333%.

STY3 calculates the percentage of condition lines where the statements are on the same line as the condition.

- STY3a The number of conditions in the code (CC) = 15, of which 14 are in the same line, giving 14/CC =93.3333%.
- STY3b calculates the percentage of condition lines(CC) where the statements are on a different line from the condition, i.e. 1/CC=6.6666%.

STY4 calculates the average of white space to the left side of operators (WSL)= 84 . The total number of operators (95) in the code gives an average of 0.8842 white spaces per operator.

STY5 calculates the average of white space to the right side of operators (WSR), and gives the same average figure of 0.8842 white spaces per operator.

STY6 calculates the ratio of blank lines to code lines (including comments). The total number of blank lines in this code (BL)= 19, divided by the total number of code lines (CL)= 109, the result is BL/CL = 0.1743.

STY7 calculates the ratio of comments lines to non-comments lines. The total number of pure comments lines (7) divided by the total number of non-comments lines (121) =0.0578 pure comments lines per non-comments line.

STY8 calculates the ratio of code lines containing comment to code lines without any comments as 0.1010 code lines containing comment per code line without any comments.

Table 4.9: Layout Metrics Calculation: Modified and Extended for Java Example

Modified		**Extended**	
STY1a	18.1818%	STY1a	18.1818%
STY1b	0%	STY1b	0%
STY1c	81.8181%	STY1c	81.8181%
STY1d	18.1818%	STY1d	18.1818%
STY1e	81.8181%	STY1e	81.8181%
STY1f	0%	STY1f	0%
STY1g	0	STY1g	0
STY1h	0	STY1h	0
STY2a	70%	STY2a	70%
STY2b	83.3333%	STY2b	83.3333%
STY3	93.3333%	STY3a	93.3333%
		STY3b	6.6666%
STY4	0.8842	STY4	0.7851
STY5	0.8842	STY5	0.7851
STY6	0.1743	STY6	0.1743
STY7	0.0578	STY7	0.0578
STY8	0.1010	STY8	0.1010

Style Metrics Calculation

his section will explain the calculation of the style metric in the Java example in the Appendix to this chapter.

PRO1 calculates the mean of line length in terms of characters. The total number of characters in the code (3178) divided by the total number of lines (128) = 24.8281 characters per line.

PRO2: a vector of metrics of name length

- PRO2a calculates the mean variable name length. The variables length is (87) / the total number of variables (26) = 3.4166 characters per variable name.
- PRO2b calculates the mean function name length. The function length is (4) / the total number of variables (1) = 4 function length per variable.

PRO3: Character preferences of uppercase, lowercase, underscore, or dollar sign for name convention.

- PRO3a calculates the percentage of uppercase characters,
- PRO3b calculates the percentage of lowercase characters,
- PRO3c calculates the percentage of underscore and PRO3d calculates the percentage of dollar signs, among the sum of the lengths of both variables and functions in the entire code, taken from PRO2a and PRO2b.

PRO4: Preference of either `while`, `for` or `do` loops.

- PRO4a calculates the percentage of `while` in the total of `while`, `for` and `do`; there is not any `while` in this code, so the final value is 0.
- PRO4b calculates the percentage of `for` in the total of `while`, `for` and `do`. The number of `for` is 3, divided by the total number of `while`, `for` and `do` (3) gives 100%.
- PRO4c calculates the percentage of `do` in the total of `while`, `for` and `do`; there is not any `do` in this code, so the final value is 0.

PRO5: Preference of either `if-else` or `switch-case` condition.

- PRO5a calculates the percentage of `if` and `if-else` in the total of `if`, `if-else` and `switch-case`, giving 11/15=73.3333%.
- PRO5b calculates the percentage of `switch-case` in the total of `if`, `if-else` and `switch-case`, giving 4/15=26.6666%.

Table 4.10: Style Metrics Calculation: Modified and Extended for Java Example

Modified		**Extended**	
PRO1	24.8281	PRO1	24.8281
PRO2a	3.4166	PRO2a	3.4166
PRO2b	4	PRO2b	4
PRO3a	1.0989%	PRO3a	1.0989%
PRO3b	58.2417%	PRO3b	58.2417%
PRO3c	0%	PRO3c	0%
PRO3d	0%	PRO3d	0%
		PRO3e	40.6593%
PRO4a	0%	PRO4a	0%
PRO4b	100%	PRO4b	100%
PRO4c	0%	PRO4c	0%
PRO5a	73.3333%	PRO5a	73.3333%
PRO5b	26.6666%	PRO5b	26.6666%

Structure Metrics Calculation

This section will explain the calculation of the structure metric in the Java example in the Appendix to this chapter.

-The total number of class /interface (CI) = 1

PSM1 calculates the average non-comment lines per class/interface. The total number of lines per class and interface is 111 divided by CI = 111 non-comment lines per class/interface.

Table 4.11: Structure Metrics Calculation: Modified and Extended for Java Example

Modified		**Extended**	
PSM1	111	PSM1	111
PSM2	26	PSM2	26
PSM3	1	PSM3	1
PSM4a	0%	PSM4a	0%
PSM4b	100%	PSM4b	100%
PSM5	0.25490	PSM5	0.25490
PSM6	0.0098	PSM6	0.0098
PSM7a	0.0098	PSM7a	0.0098
PSM7b	0	PSM7b	0
PSM7c	0.0098	PSM7c	0.0098
PSM7d	0.0098	PSM7d	0.0098
PSM7e	0	PSM7e	0
PSM7f	0.0686	PSM7f	0.0686
PSM7g	0	PSM7g	0
PSM7h	0	PSM7h	0
PSM7i	0	PSM7i	0
PSM7j	0.0686	PSM7j	0.0686
PSM7k	0.0098	PSM7k	0.0098
PSM7l	0.0294	PSM7l	0.0294
PSM7m	0.0392	PSM7m	0.0392
PSM7n	0	PSM7n	0
PSM7o	0	PSM7o	0
PSM7p	0	PSM7p	0
PSM7q	0.0098	PSM7q	0.0098
PSM7r	0	PSM7r	0
PSM7s	0.0098	PSM7s	0.0098
PSM7t	0	PSM7t	0
		PSM8a	1.0989
		PSM8b	58.2417
		PSM8c	0
		PSM8d	0
		PSM8e	40.6593
		PSM9a	0
		PSM9b	100
		PSM9c	0
		PSM10a	73.3333
		PSM10b	26.6666

PSM2 calculates the average number of variables per class/interface. The total number of variables is 26 divided CI = 26 variables per class/interface.

PSM3 calculates the average number of functions per class/interface. The total number of functions in the code is 1, divided by CI = 1 function per class/interface.

PSM4 calculates the ratio of interfaces to classes. This metric is divided into two components:

- PSM4a: the percentage of interfaces in the total of interfaces and classes is 0%.
- PSM4b: the percentage of classes in the total of interfaces and classes is 100%.

- Number of lines of non-comment code (LNCC) = 102

- Total number of lines of code (LC) = 109

PSM5 calculates the ratio of variable count / LNCC= 26/102= 0.2549 primitive variables per line of non-comment code.

PSM6 calculates the ratio of function count / LNCC= 1/102= 0.0098 functions per line of non-comment code.

The calculation of the PSM7* metrics is here shown explicitly:

- PSM7a calculates ratio of the number of instances of keyword `static` / LNCC=The total number of keyword `static` is 1/LNCC= 0.0098 keywords `static` per line of non-comment code.
- PSM7b calculates the ratio of keyword `extends` / LNCC= The total number of keyword `extends` is 0/ LNCC = 0 keywords `extends` per line of non-comment code.
- PSM7c calculates the ratio of keyword `class` / LNCC= The total number of

keyword `class` is 1/LNCC = 0.0098 keywords `class` per line of non-comment code.

- PSM7d calculates the ratio of keyword `abstract` / LNCC= the total number of keyword `abstract` is 0/ LNCC= 0 keywords `abstract` per line of non-comment code.
- PSM7e calculates the ratio of keyword `implements` / LNCC. The total number of keyword `implements` is 0/ LNCC = 0 keywords `implements` per line of non-comment code.
- PSM7f calculates the ratio of keyword `import` / LNCC= The total number of keyword `import` is 7/ LNCC = 0.0686 keywords `import` per line of non-comment code.
- PSM7g calculates the ratio of keyword `instance of` / LNCC = the total number of keyword `instance of` is 0/ LNCC = 0 keywords `instance of` per line of non-comment code.
- PSM7h calculates the ratio of keyword `interface` / LNCC = the total number of keyword `interface` is 0/ LNCC = 0 keywords `interface` per line of non-comment code.
- PSM7i calculates the ratio of keyword `native` LNCC= the total number of keyword `native` is 0/ LNCC = 0 keywords `native` per line of non-comment code.
- PSM7j calculates the ratio of keyword `new` LNCC= the total number of keyword `new` is 7/ LNCC = 0.0686 keywords `new` per line of non-comment code.
- PSM7k calculates the ratio of keyword `package` LNCC= the total number of

keyword `package` is 0/ LNCC = 0 keywords `package` per line of non-comment code.

- PSM7l calculates the ratio of keyword `private` LNCC= the total number of keyword `private` is 0/ LNCC = 0 keywords `private` per line of non-comment code.
- PSM7m calculates the ratio of keyword `public` LNCC= the total number of keyword `public` is 3/ LNCC = 0.0294 keywords `public` per line of non-comment code.
- PSM7n calculates the ratio of keyword `protected` LNCC= the total number of keyword `protected` is 4/LNCC = 0.0392 keywords `protected` per line of non-comment code.
- PSM7o calculates the ratio of keyword `this` LNCC= the total number of keyword `this` is 0/ LNCC = 0 keywords `this` per line of non-comment code.
- PSM7p calculates the ratio of keyword `super` LNCC= the total number of keyword `super` is 0/ LNCC = 0 keywords `super` per line of non-comment code.
- PSM7q calculates the ratio of keyword `try` LNCC= the total number of keyword `try` is 0/ LNCC = 0 keywords `try` per line of non-comment code.
- PSM7r calculates the ratio of keyword `throw` LNCC= the total number of keyword `throw` is 1/ LNCC = 0.0098 keywords `throw` per line of non-comment code.
- PSM7s calculates the ratio of keyword `catch` LNCC= the total number of keyword `catch` is 1/ LNCC = 0.0098 keywords `catch` per line of non-comment code.

PSM8 Character preference of uppercase, lowercase, underscore, or dollar sign for name convention.

- Total number of upper-case letters used in class and interface names (UC) = 1

- Total number of lower-case letters used in class and interface names (LC) = 53.

The calculation of the metrics PSM8, PSM9 and PSM10 follows a similar pattern. All three metrics take account of the number of classes and interfaces.

4.6 Conclusion

In this chapter, the study used existing style analysis metrics as structure-based metrics for plagiarism detection purposes. This research is one of few studies that have investigated coding style analysis in order to identify plagiarism. Additionally, the metrics have been enhanced by introducing two new families of metrics: the Modified metrics and the Extended metrics. The two metrics cover these categories: Layout, Style and Structure. This chapter proposed the Modified and the Extended metrics to identify possible instances of plagiarism attacks. The Modified metrics introduced the Modified programming metric STY3 that calculates the percentage of condition lines where the statements are in the same line as the condition with the aim of predicting an attack called "modification of control structure" [125]. The Extended metrics introduced the Extended metrics PSM8, PSM9 and PSM10 [125] with the aim of detecting an attack called "modification of control structure and structural redesign of code". This chapter has addressed research two objectives: "To re-implement coding style metrics of the fingerprint to identify the plagiarism cases" and "To modify and extend new style metrics to enrich the ability to detect new cases of source code plagiarisms". The calculation of the Modified and Extended metrics is illustrated with reference to the Java program in the Appendix to this chapter. The outputs of the three categories are summarised and the differences between the Modified and the Extended metrics are presented.

The next chapter will examine the way in which the Modified and the Extended metrics can be exploited in plagiarism detection. The Modified and the Extended metrics will be embedded into a framework within which Singular Value Decomposition analysis can be applied to plagiarism detection.

Appendix: Java Code Example for Illustrating Metrics Calculation

Code 4.1: Java Example

```java
package Aufgabe2;

import java.io.FileInputStream;
import java.io.FileNotFoundException;
import java.io.FileOutputStream;
import java.io.InputStream;
import java.io.OutputStream;
import java.io.PrintWriter;
import java.util.Scanner;

public class Main {

 private Scanner scanner;
 private PrintWriter writer;

 public Main(InputStream is, OutputStream os) {
  scanner = new Scanner(is);
  writer = new PrintWriter(os);
 }

 public void solve() {

  // Unit Tests:
  int tmp;
  int tmp1[] = {15,13,11};
  if (calculate(1, 5, tmp1)!=3) return;
  int tmp2[] = {23,22,21};
  if (calculate(0, 8, tmp2)!=2) return;
  int tmp3[] = {8,0};
  if (calculate(1, 1, tmp3)!=1) return;
  int tmp4[] = {29,20,8,18,18,21};
  if (calculate(2, 8, tmp4)!=3) return;

  for (int i=0; i<200; i++)
   System.out.println("");

  int cases = scanner.nextInt();

```

Code 4.1 (Cont.): Java Example

```
 for (int i = 1; i <= cases; i++) {
  writer.print("Case␣#");
  writer.print(i + ":␣");

  scanner.nextLine();
  int n = scanner.nextInt(); // number of Googlers
  int s = scanner.nextInt(); // number of surprising triplets
  int p = scanner.nextInt(); // minimum score
  int t[] = new int[n];
  for (int j = 0; j < n; j++)
   t[j] = scanner.nextInt();

  System.out.println(i + ":␣MinScore=" + p + ",␣Surprising␣Triplets="
      + s);
  int result = calculate(s, p, t);

  writer.println(result);
  System.out.println("␣->␣" + result);
  System.out.println("-------------");
 }
 writer.flush();
}

private int calculate(int noSurprisingScores,
  int minimumScore, int[] scores) {
 int result = 0;

 for (int score : scores) {
  int triplet[] = new int[3];
  System.out.print(score + ":␣");
  switch (score % 3) {
  case 0:
   if (score / 3 >= minimumScore) {
    result++;
    System.out.println((score / 3) + "," + (score / 3) + "," + (score
        / 3));
   } else {
    if (noSurprisingScores > 0)
     if (score / 3 + 1 >= minimumScore && score / 3 - 1 >= 0) {
      result++;
      noSurprisingScores--;
      System.out.println((score / 3 - 1) + "," + (score / 3) + "," +
          (score / 3 + 1) + "␣*");
     }
```

Code 4.1 (Cont.): Java Example

```java
   }
   break;
  case 1:
   if ((score-1)/3 + 1 >= minimumScore) {
    result++;
    System.out.println((score-1)/3 + "," + ((score-1)/3) + "," + ((score-1)/3 + 1));
   } else {
    // Surprising Triplet fhrt nicht zur Erhhung des max Scores
   }
   break;
  case 2:
   if ((score-2)/3 + 1 >= minimumScore) {
    result++;
    System.out.println((score-2)/3 + "," + ((score-2)/3 + 1) + "," + ((score-2)/3 + 1));
   } else {
    if (noSurprisingScores > 0)
     if ((score - 2) / 3 + 2 >= minimumScore) {
      result++;
      noSurprisingScores--;
      System.out.println((score-2)/3 + "," + (score-2)/3 + "," + ((score-2)/3 + 2) + "␣*");
     }
   }
   break;
  }

 }

 return result;
}

/*
 * private boolean exceedsMinumumScore(int[] triplet, int minScore) { for
 * (int i=0; i<3; i++) if (triplet[i] >= minScore) return true; return
 * false; }
 */

public static void main(String[] args) {
 try {
  InputStream is = new FileInputStream("B-small-attempt.in");
  OutputStream os = new FileOutputStream("B-small-attempt.out");
```

Code 4.1 (Cont.): Java Example

```
    Main problem = new Main(is, os);
    problem.solve();
   } catch (FileNotFoundException e) {
    e.printStackTrace();
   }

   System.out.println("finished");
  }
}
```

Chapter 5

A Framework for Developing Plagiarism Detection Techniques

5.1 Introduction

The potential use of coding style metrics for plagiarism detection has been discussed in the previous chapter. This chapter describes a framework for enhancing such metrics based approaches using alternative techniques for statistical analysis.

Section 5.2 introduces a new framework for plagiarism detection modelled on standard techniques for document analysis which are based on applying Singular Value Decomposition (SVD) to 'term-document' matrices (TDMs). These techniques are generalised to apply to 'metric-file' matrices (MFMs). The section describes background principles and concepts behind SVD. It also introduces the cosine similarity measure for detecting similar files in a dataset. Section 5.3 describes the MFM framework for plagiarism detection in more detail and is followed by an illustration of application of the MFM framework in section 5.4.

5.2 Adapting TDM Analysis for Plagiarism

One of the basic and most important tools for numerical analysis is Singular Value Decomposition (SVD) which was established by five mathematicians: Eugenio Beltrami (1835–1899) and Camille Jordan (1838-1921) [137]; James Joseph Sylvester (1814–1897), Erhard Schmidt (1876–1959) and Hermann Weyl (1885–1955) [137]. SVD is a technique for expressing a real or complex matrix A as a product

$$A_{[m\times n]} = U_{[m\times r]} \sum\nolimits_{[r\times r]} (V_{[n\times r]})^T \tag{5.1}$$

where $\sum$ is a diagonal matrix of singular values.

SVD has been applied in various scenarios such as for transforming genome-wide expression data from genes [3] and finding the gene structure pattern and the underlying “characteristic modes” [65]. Also, SVD has appeared in textual database searching [14], robotics [11], financial mathematics [49] and data computation and secure encryption [86] [133] [142]. In a medical context, SVD was introduced by Ikeda et al. [69] for analysing ultrasound signals, to separate tissue flow and cavitation signals. Also, SVD has been used for detecting voting similarities between politicians [122] [121].

In this thesis, we apply SVD to detect similarities between source code files. For this purpose we adapt a technique for document analysis based on term-document matrices. A term-document matrix has rows corresponding to documents and columns corresponding to terms. The entry in row r and column c is the number of times the term in column c occurs in the document in row r.

In its raw form, a term-document matrix is typically too large for mathematical analysis. SVD makes it possible to derive a low-rank matrix approximation which can be applied in document analysis (e.g for author identification and similarity detection). We shall refer to such an application of SVD in conjunction with a term-document matrix as a *TDM approach* to document analysis.

To adapt the TDM approach for detecting source code plagiarism, we substitute a metric-file matrix for a term-document matrix. In a metric-file matrix, the rows represent the source code files in the dataset and the columns represent the source code programming metrics being applied to the files. The entry in row r and column c is the value of the metric in column c when applied to the source file in row r. The substitution of metric-file matrices for term-document matrices motivates us to generalise the TDM approach to a new metric-file matrix (MFM) approach to plagiarism detection. This approach will now be explained in detail.

Let $A = [a_{ij}]$ be an $m \times n$ metric-file matrix, where the m rows represent files, the n columns are coding style metrics, and each cell of matrix A contains the value of coding style metric i when applied to file j. SVD will be used to reduce the dimensionality of the input matrix (number of input files by number of extracted metrics).

SVD supplies a decomposition of the $m \times n$ matrix as a product of three matrices:

$$A_{[m\times n]} = U_{[m\times r]} \sum\nolimits_{[r\times r]} (V_{[n\times r]})^T \tag{5.2}$$

where:

- A is the $m \times n$ input data matrix.

- U is the $m \times r$ matrix of left singular vectors.
- $\sum$ is $r \times r$ diagonal matrix of singular values.
- V is $n \times r$ matrix of right singular vectors.

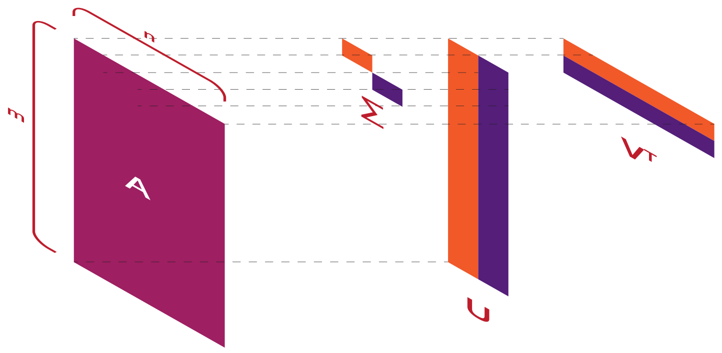


Figure 5.1: Explanatory Diagrams of SVD

Conceptually, the rows of the matrix A correspond to files and the columns to coding style metrics. The rows of the matrices U and V also correspond to files and their columns to ‘concepts’ that characterise files in abstract mathematical terms. The singular values in $\sum$ reflect the strength of each concept.

The rank r of matrix A is the number of non-zero diagonal elements of matrix $\sum$. As in the TDM approach, we shall use SVD to give a low-rank matrix approximation to matrix A. To derive an approximation of rank k, we can select the k largest singular values from the matrix $\sum$, replacing the other singular values by zero. This renders all but k columns from the matrices U and V redundant, and the other columns in these matrices can be deleted. This process, described as dimensionality reduction, means that the three matrices U, $\sum$ and V can be reduced to dimension $m \times k$, $k \times k$ and $n \times k$ respectively.

The reduced matrices are denoted by U_k, $\sum_k$ and V_k where U_k is a $m \times k$ matrix, $\sum_k$ is a $k \times k$ matrix and V_k is a $n \times k$ matrix. The rank k approximation to matrix A can be constructed as $A_k = U_k \sum_k V_k^T$.

In deriving a low-rank matrix approximation, it is very important that k is significantly smaller than r. Having a low value for the rank k reduces the noise in the data and makes it easier to identify the relation between the coding style metrics and the files. It is also helps to reduce the computation time [14] [13].

After SVD is applied to the metric-file matrix and the data is represented in a reduced dimensional space, a new file (i.e. a query file) can be projected into the space. The similarity of the query and the rest of the files can then be calculated using the cosine similarity measure, as described in subsection 5.2.1.

The MFM approach as set out above can be based on many different sets of coding style metrics. These include for instance the Modified and Extended families of metrics introduced in chapter 4. On this basis, the MFM approach serves as a framework for deriving plagiarism detection techniques from families of metrics.

In order to evaluate the performance of plagiarism techniques derived from families of metrics within the MFM framework, the source code dataset was processed in the following way. A number of groups of similar files were identified. One or more files from each group were selected to be query files. The SVD approximation technique was applied to the files which were not selected to be the queries. Once SVD was applied, the queries were projected into the space and their similarity with the rest of the files was computed using the cosine similarity measure.

The metric-file matrix and queries record the coding-style metric values as applied to the source code files. As explained in [14], a query vector q can derived from a query file by applying each of the coding style metrics; the query vector can then be projected to k-dimensional space. In [14] the roles of terms and documents are interchanged so that terms correspond to rows and documents to columns.

Taking this into account the projection to k-dimensional space is as follows:

$$Q = qV_k \sum\nolimits_k^{-1} \tag{5.3}$$

5.2.1 Cosine Similarity

The query vector is projected by term-file space, and can be compared to all other files using the cosine similarity measurement. Cosine similarity computes the angles between a query and each file vector and returns the file IDs most similar to the query, in ranked order. Hence, the result shows the file IDs in a rank list sorted in descending order, which means the most similar values to the query have highest rank.

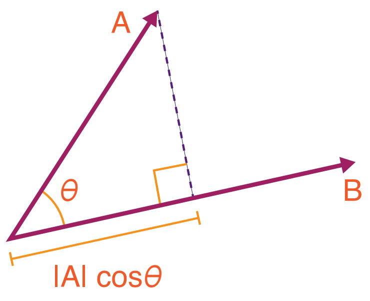


Figure 5.2: Explanatory Diagrams of *Cosine*

The *cosine* similarity measurement is applied to two vectors, in our case two k-dimensional vectors representing Java source code files. One of these vectors is vector Q which is derived by projection as in equation 5.2. The other vector is selected from the set $\{t_1, t_2, ..., t_m\}$ of vectors representing source code files in the original dataset. For the j-th source code file the cosine similarity is then computed as follows:

$$\theta_j = \frac{t_j^T q}{||t_j||||q||} = \frac{\sum_{i=1}^{k} t_{ji} q_i}{\sqrt{\sum_{i=1}^{k} t_{ji}^2}\sqrt{\sum_{i=1}^{k} q_i^2}} \tag{5.4}$$

The performance of the proposed model is then evaluated using the Precision, Recall and F-measure [25] [68] evaluation measures.

5.2.2 Performance Evaluation Measures

How well a similarity detection system performs is normally measured by two factors: Precision and Rcall.

- **Precision** is the proportion of the retrieved files that are relevant. Precision is computed as in formula 5.5, where T is the number of files similar to the query in the dataset that are retrieved and R is the total number of files retrieved. Precision is 1.00 when all the retrieved files are similar to the query.

$$P = \frac{T}{R} \tag{5.5}$$

- **Recall** is the proportion of files similar to the query in the dataset that are retrieved. Recall is computed as in formula 5.5, where T is the number of files similar to the query in the dataset that are retrieved and S is the total number of similar files in the dataset. Recall is 1.00 when every similar file is retrieved.

$$R = \frac{T}{S} \tag{5.6}$$

- **F-measure** or F_1 score, is a measure that combines precision and recall and it is the harmonic mean of precision and recall. The traditional F-measure function is shown in (5.7)

$$F_1 = 2 \times \frac{Precision \times Recall}{Precision + Recall} \tag{5.7}$$

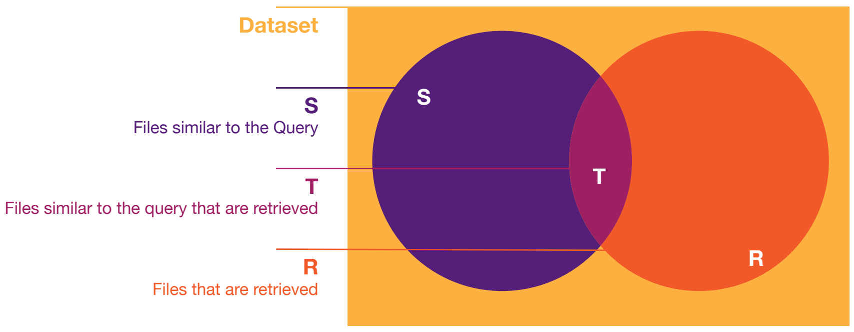

Dataset
S
Files similar to the Query
T
Files similar to the query that are retrieved
R
Files that are retrieved
S
T
R

Figure 5.3: Explanatory Diagram for Precision and Recall

Figure 5.3 depicts the relationship between the values R, S and T introduced above. R is the total number of files retrieved, S is the total number of files similar to the query in the dataset and T is the number of files similar to the query in the dataset that are retrieved.

The application of the MFM framework is based on theoretical principles and concepts described in this section. The following section in this chapter discuss the practical application of the MFM framework in more detail. The empirical results of this application, as set out on the rest of the thesis, have been computed using the MATLAB software package. The implementation of the key techniques described above makes use of standard procedures for Singular Value Decomposition analysis built into the MATLAB environment.

5.3 The MFM Framework for Plagiarism Detection

This section introduces a new framework which takes advantage of analysis of source code style metrics based on SVD to enhance source code similarity detection.

The proposed framework can exploit the Modified and the Extended families of metrics introduced in sections 4.3 and 4.4. These respectively provide the basis for the Modified MFM and Extended MFM approaches to plagiarism detection.

The components of the proposed framework are presented in Figure 5.4, where each component has a specific computational function to accomplish in the analysis process.

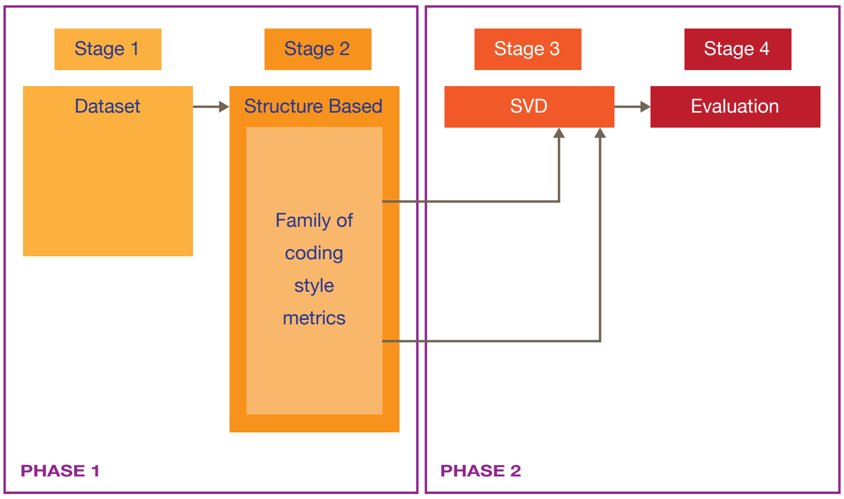


Figure 5.4: Outline of the main components of MFM Framework

5.3.1 Phase One in the Application of the MFM Framework

As shown in 5.4 there are two stages in the first phase of the application of the MFM framework:

1. **Stage 1:** the source code dataset has been downloaded, and the files in the dataset are not pre-processed.

2. **Stage 2:** A family of coding style metrics is used to represent source code files by metric-file matrices. In this chapter, the Modified and Extended families of metrics introduced in chapter 4 are chosen for further study.

5.3.2 Phase Two in the Application of the MFM Framework

This phase includes the SVD calculation and analysis. It consists of two main stages:

1. **Stage 3:** the output from the previous phase - a family of metric-file matrices - is subjected to SVD, which reduces the noise in the data and produces relationships between the files.

2. **Stage 4:** present the results of the SVD analysis, measuring the similarity between files using *cosine* similarity calculation and evaluating the results with respect to the Precision, Recall and F-measure.

5.4 Illustrating the Application of the MFM Framework

In order to illustrate the application of the MFM framework, several components are proposed to fulfill the task of capturing the source code style and identifying similar files. The components and their detailed implementation are presented in the subsections that follow.

5.4.1 The Dataset

Experiments were performed using the SOCO dataset. The dataset contains source code files created by the PAN@FIRE event which was derived from Google Jam competition 2012. So this research obtained the dataset from PAN@FIRE to be described in more detail chapter 6 in section 6.2. The dataset contains 6 sub-folders A1, A2, B1, B2, C1 and C2. The results obtained through application of the MFM framework are compared with those obtained from the standard JPlag plagiarism detection tool [125].

5.4.2 Coding Style Metrics

Stage two requires a family of coding style metrics. The application of the MFM framework is illustrated by using the Modified metrics introduced in chapter 4. Table 5.1 shows the results from just category one, the modified layout metrics, for 4 files denoted as F1, F2, F3 and F4 which were extracted from a group of files (i.e. Group A1) found in the SOCO dataset. The coding style metrics are calculated as discussed in the chapter 4 example. So, for example, STY1c is calculating the percentage of open braces ({) that are the last character in the line. In the table the results of F1, F2, F3 and F4 for STY1c are all 100%. The three tables 5.1, 5.2

and 5.3 show calculations for each programming metric with the four files. F1 is labelled as a query and F2 is the similar file to F1; similarly, F3 is a query and F4 is the similar file to F3.

Table 5.1: Modified Layout Metrics: F1 and F3 are JPlag queries, F2 and F4 are similar files

File ID	**F1**	**F2**	**F3**	**F4**
STY1a	0%	0%	0%	0%
STY1b	0%	0%	0%	0%
STY1c	100%	100%	100%	100%
STY1d	0%	0%	0%	0%
STY1e	100%	100%	100%	100%
STY1f	0%	0%	0%	0%
STY1g	0	0	0	0
STY1h	0	0	0	0
STY2a	50%	50%	100%	100%
STY2b	100%	100%	100%	100%
STY3	100%	100%	100%	100%
STY4	0.596	0.596	0.069	0.069
STY5	0.557	0.557	0.093	0.093
STY6	0.189	0.189	0.212	0.212
STY7	0.031	0.031	0.021	0.021
STY8	0.077	0.077	0.025	0.025

Table 5.2: Modified Style Metrics : F1 and F3 are JPlag queries, F2 and F4 are similar files

File ID	F1	F2	F3	F4
PRO1	25.083	25.083	15.082	15.082
PRO2a	4.7619	4.7619	4.057	4.057
PRO2b	12.75	12.75	17	17
PRO3a	2.649%	2.649%	4.166%	4.166%
PRO3b	90.728%	90.728%	87.5%	87.5%
PRO3c	0%	0%	0%	0%
PRO3d	0%	0%	0%	0%
PRO4a	12.5%	12.5%	100%	100%
PRO4b	87.5%	87.5%	0%	0%
PRO4c	0%	0%	0%	0%
PRO5a	100%	100%	100%	100%
PRO5b	0%	0%	0%	0%

In order to evaluate the similarity detection performance of the proposed framework, a set of queries and their relevant files were required. The process for creating the sets of queries and their similar files was performed using the JPlag tool.

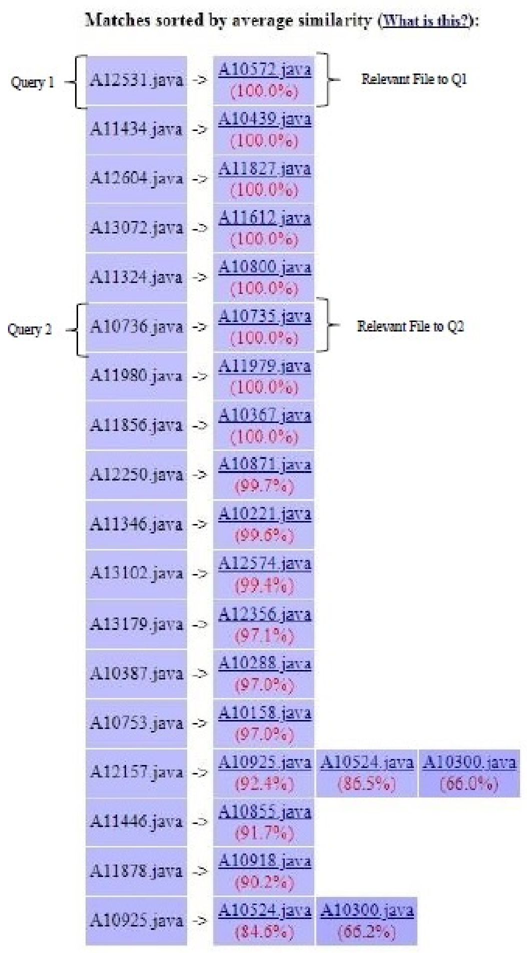
The dataset was initially passed into the external tool JPlag for identifying the queries and the similar files. The output consisted of groups of similar source code files that were detected by JPlag. Each group has a query file, along with its relevant files that are suspected to be similar to the main file. The queries were transformed into a vector which means the values of the metrics in the query would be non-zero elements in the query. The results from JPlag are shown in Figure 5.5: the main files are on the left hand side, whereas the similar files are on the right hand side. One main file can have many similar files.

Table 5.3: Modified Structure Metrics : F1 and F3 are JPlag queries, F2 and F4 are similar files

File ID	F1	F2	F3	F4
PSM1	188	188	92	92
PSM2	21	21	19	19
PSM3	4	4	1	1
PSM4a	0%	0%	0%	0%
PSM4b	100%	100%	100%	100%
PSM5	0.196	0.196	0.243	0.243
PSM6	0.037	0.037	0.012	0.012
PSM7a	0.009	0.009	0.025	0.025
PSM7b	0	0	0	0
PSM7c	0.009	0.009	0.012	0.012
PSM7d	0	0	0	0
PSM7e	0	0	0	0
PSM7f	0.065	0.065	0.012	0.012
PSM7g	0	0	0	0
PSM7h	0	0	0	0
PSM7i	0	0	0	0
PSM7j	0.102	0.102	0.051	0.051
PSM7k	0.009	0.009	0.012	0.012
PSM7l	0	0	0	0
PSM7m	0.065	0.065	0.038	0.038
PSM7n	0	0	0	0
PSM7o	0	0	0	0
PSM7p	0	0	0	0
PSM7q	0.018	0.018	0.012	0.012
PSM7r	0	0	0	0
PSM7s	0.018	0.018	0.012	0.012
PSM7t	0.009	0.009	0	0

5.4.3 SVD Analysis and Cosine Similarity

SVD was applied to the matrices created after applying the Modified metrics. For the styling metrics in each category (Layout, Style and Structure), the value of each metric was calculated and metric-file matrices were created. The metric-file matrices were subjected to SVD analysis with reference to a list of queries and their relevant



Matches sorted by average similarity (What is this?):

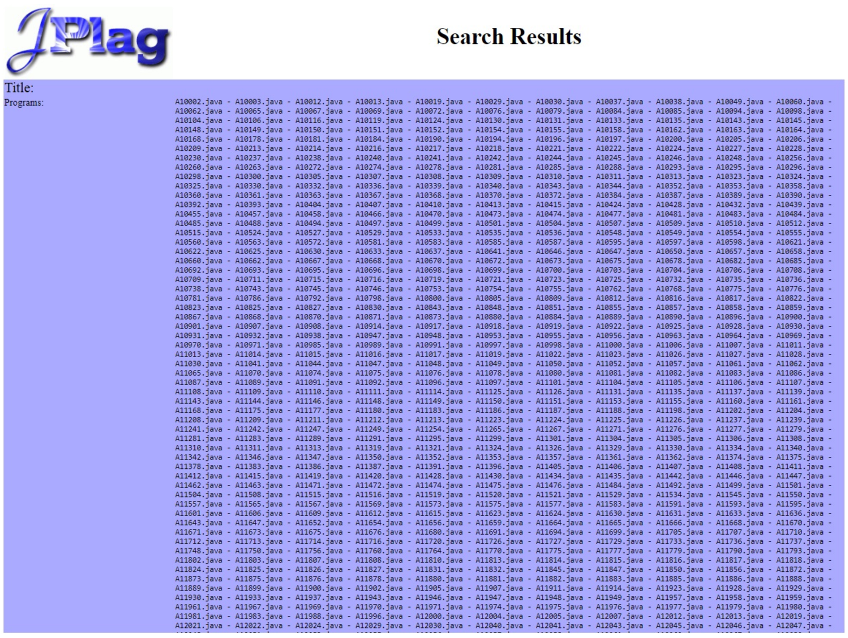
A12531.java	->	A10572.java (100.0%)		
A11434.java	->	A10439.java (100.0%)		
A12604.java	->	A11827.java (100.0%)		
A13072.java	->	A11612.java (100.0%)		
A11324.java	->	A10800.java (100.0%)		
A10736.java	->	A10735.java (100.0%)		
A11980.java	->	A11979.java (100.0%)		
A11856.java	->	A10367.java (100.0%)		
A12250.java	->	A10871.java (99.7%)		
A11346.java	->	A10221.java (99.6%)		
A13102.java	->	A12574.java (99.4%)		
A13179.java	->	A12356.java (97.1%)		
A10387.java	->	A10288.java (97.0%)		
A10753.java	->	A10158.java (97.0%)		
A12157.java	->	A10925.java (92.4%)	A10524.java (86.5%)	A10300.java (66.0%)
A11446.java	->	A10855.java (91.7%)		
A11878.java	->	A10918.java (90.2%)		
A10925.java	->	A10524.java (84.6%)	A10300.java (66.2%)	

Figure 5.5: The output results from JPlag for A1 files and relevant files

files.

Table 5.4 and 5.5 presents the results of the two queries: Q1(A12531) and Q2(A12250) for the A1 group. The two tables consist of three columns: the first column shows how the relevant files are ranked based on the Cosine similarity after applying SVD analysis, the second column shows the names of relevant file in group A1, renamed as F1, F2, ... and the third column represents the similarity as assessed by a human expert.

Table 5.4 presents the results of the query (1: A12531). The results show



```
Title:
Programs:  A10002.java - A10003.java - A10012.java - A10013.java - A10019.java - A10029.java - A10030.java - A10037.java - A10038.java - A10049.java - A10060.java -
           A10062.java - A10065.java - A10067.java - A10069.java - A10072.java - A10076.java - A10079.java - A10084.java - A10085.java - A10094.java - A10098.java -
           A10104.java - A10106.java - A10116.java - A10119.java - A10124.java - A10130.java - A10131.java - A10133.java - A10135.java - A10143.java - A10145.java -
           A10148.java - A10149.java - A10150.java - A10151.java - A10152.java - A10154.java - A10155.java - A10158.java - A10162.java - A10163.java - A10164.java -
           A10168.java - A10178.java - A10181.java - A10184.java - A10190.java - A10194.java - A10196.java - A10197.java - A10200.java - A10205.java - A10206.java -
           A10209.java - A10213.java - A10214.java - A10216.java - A10217.java - A10218.java - A10221.java - A10222.java - A10224.java - A10227.java - A10228.java -
           A10230.java - A10237.java - A10238.java - A10240.java - A10241.java - A10242.java - A10244.java - A10245.java - A10246.java - A10248.java - A10256.java -
           A10260.java - A10263.java - A10272.java - A10274.java - A10278.java - A10281.java - A10285.java - A10288.java - A10293.java - A10295.java - A10296.java -
           A10298.java - A10300.java - A10305.java - A10307.java - A10308.java - A10309.java - A10310.java - A10311.java - A10313.java - A10323.java - A10324.java -
           A10325.java - A10330.java - A10332.java - A10336.java - A10339.java - A10340.java - A10343.java - A10344.java - A10352.java - A10353.java - A10358.java -
           A10360.java - A10361.java - A10363.java - A10367.java - A10368.java - A10370.java - A10372.java - A10384.java - A10387.java - A10389.java - A10390.java -
           A10392.java - A10393.java - A10404.java - A10407.java - A10410.java - A10413.java - A10415.java - A10424.java - A10428.java - A10432.java - A10439.java -
           A10455.java - A10457.java - A10458.java - A10466.java - A10470.java - A10473.java - A10474.java - A10477.java - A10481.java - A10483.java - A10484.java -
           A10485.java - A10488.java - A10494.java - A10497.java - A10499.java - A10501.java - A10504.java - A10507.java - A10509.java - A10510.java - A10512.java -
           A10515.java - A10524.java - A10527.java - A10529.java - A10533.java - A10535.java - A10536.java - A10548.java - A10549.java - A10554.java - A10555.java -
           A10560.java - A10563.java - A10572.java - A10581.java - A10583.java - A10585.java - A10587.java - A10595.java - A10597.java - A10598.java - A10621.java -
           A10622.java - A10625.java - A10630.java - A10633.java - A10637.java - A10641.java - A10646.java - A10647.java - A10650.java - A10657.java - A10658.java -
           A10660.java - A10662.java - A10667.java - A10668.java - A10670.java - A10672.java - A10673.java - A10675.java - A10678.java - A10682.java - A10685.java -
           A10692.java - A10693.java - A10695.java - A10696.java - A10698.java - A10699.java - A10700.java - A10703.java - A10704.java - A10706.java - A10708.java -
           A10709.java - A10711.java - A10715.java - A10716.java - A10719.java - A10721.java - A10723.java - A10725.java - A10732.java - A10735.java - A10736.java -
           A10738.java - A10743.java - A10745.java - A10746.java - A10753.java - A10754.java - A10755.java - A10762.java - A10768.java - A10775.java - A10776.java -
           A10781.java - A10786.java - A10792.java - A10798.java - A10800.java - A10805.java - A10809.java - A10812.java - A10816.java - A10817.java - A10822.java -
           A10823.java - A10825.java - A10827.java - A10830.java - A10843.java - A10848.java - A10851.java - A10855.java - A10857.java - A10858.java - A10859.java -
           A10867.java - A10868.java - A10870.java - A10871.java - A10873.java - A10880.java - A10884.java - A10889.java - A10890.java - A10896.java - A10900.java -
           A10901.java - A10907.java - A10908.java - A10914.java - A10917.java - A10918.java - A10919.java - A10922.java - A10925.java - A10928.java - A10930.java -
           A10931.java - A10932.java - A10938.java - A10947.java - A10948.java - A10953.java - A10955.java - A10956.java - A10963.java - A10964.java - A10969.java -
           A10970.java - A10971.java - A10985.java - A10989.java - A10991.java - A10997.java - A10998.java - A11000.java - A11006.java - A11007.java - A11011.java -
           A11013.java - A11014.java - A11015.java - A11016.java - A11017.java - A11019.java - A11022.java - A11023.java - A11026.java - A11027.java - A11028.java -
           A11030.java - A11041.java - A11044.java - A11047.java - A11048.java - A11049.java - A11050.java - A11052.java - A11057.java - A11061.java - A11062.java -
           A11065.java - A11070.java - A11074.java - A11075.java - A11076.java - A11078.java - A11080.java - A11081.java - A11082.java - A11083.java - A11086.java -
           A11087.java - A11089.java - A11091.java - A11092.java - A11096.java - A11097.java - A11101.java - A11104.java - A11105.java - A11106.java - A11107.java -
           A11108.java - A11109.java - A11110.java - A11111.java - A11114.java - A11125.java - A11126.java - A11131.java - A11135.java - A11137.java - A11139.java -
           A11143.java - A11144.java - A11146.java - A11148.java - A11149.java - A11150.java - A11151.java - A11153.java - A11155.java - A11160.java - A11161.java -
           A11168.java - A11175.java - A11177.java - A11180.java - A11183.java - A11186.java - A11187.java - A11188.java - A11198.java - A11202.java - A11204.java -
           A11208.java - A11209.java - A11211.java - A11212.java - A11213.java - A11223.java - A11224.java - A11225.java - A11226.java - A11237.java - A11239.java -
           A11241.java - A11242.java - A11247.java - A11249.java - A11254.java - A11265.java - A11267.java - A11271.java - A11276.java - A11277.java - A11279.java -
           A11281.java - A11283.java - A11289.java - A11291.java - A11295.java - A11299.java - A11301.java - A11304.java - A11305.java - A11306.java - A11308.java -
           A11310.java - A11311.java - A11313.java - A11319.java - A11321.java - A11324.java - A11326.java - A11329.java - A11330.java - A11334.java - A11340.java -
           A11342.java - A11346.java - A11347.java - A11350.java - A11352.java - A11353.java - A11357.java - A11361.java - A11362.java - A11374.java - A11375.java -
           A11378.java - A11383.java - A11386.java - A11387.java - A11391.java - A11396.java - A11405.java - A11406.java - A11407.java - A11408.java - A11411.java -
           A11412.java - A11415.java - A11419.java - A11420.java - A11428.java - A11430.java - A11434.java - A11435.java - A11442.java - A11446.java - A11447.java -
           A11462.java - A11463.java - A11471.java - A11472.java - A11474.java - A11475.java - A11476.java - A11484.java - A11492.java - A11499.java - A11501.java -
           A11504.java - A11508.java - A11515.java - A11516.java - A11519.java - A11520.java - A11521.java - A11529.java - A11534.java - A11545.java - A11550.java -
           A11557.java - A11565.java - A11567.java - A11569.java - A11573.java - A11575.java - A11577.java - A11583.java - A11591.java - A11593.java - A11595.java -
           A11601.java - A11606.java - A11609.java - A11612.java - A11615.java - A11623.java - A11624.java - A11630.java - A11631.java - A11633.java - A11636.java -
           A11643.java - A11647.java - A11652.java - A11654.java - A11656.java - A11659.java - A11664.java - A11665.java - A11666.java - A11668.java - A11670.java -
           A11671.java - A11673.java - A11675.java - A11676.java - A11680.java - A11691.java - A11694.java - A11699.java - A11705.java - A11707.java - A11710.java -
           A11712.java - A11713.java - A11714.java - A11716.java - A11720.java - A11726.java - A11727.java - A11729.java - A11733.java - A11736.java - A11737.java -
           A11748.java - A11750.java - A11756.java - A11760.java - A11764.java - A11770.java - A11775.java - A11777.java - A11779.java - A11790.java - A11793.java -
           A11802.java - A11803.java - A11807.java - A11808.java - A11810.java - A11813.java - A11814.java - A11815.java - A11816.java - A11817.java - A11818.java -
           A11824.java - A11825.java - A11826.java - A11827.java - A11831.java - A11832.java - A11845.java - A11847.java - A11850.java - A11856.java - A11872.java -
           A11873.java - A11875.java - A11876.java - A11878.java - A11880.java - A11881.java - A11882.java - A11883.java - A11885.java - A11886.java - A11888.java -
           A11889.java - A11899.java - A11900.java - A11902.java - A11905.java - A11907.java - A11911.java - A11914.java - A11923.java - A11928.java - A11929.java -
           A11930.java - A11933.java - A11937.java - A11943.java - A11946.java - A11947.java - A11948.java - A11949.java - A11957.java - A11958.java - A11959.java -
           A11961.java - A11967.java - A11969.java - A11970.java - A11971.java - A11974.java - A11975.java - A11976.java - A11977.java - A11979.java - A11980.java -
           A11981.java - A11983.java - A11988.java - A11996.java - A12000.java - A12004.java - A12005.java - A12007.java - A12012.java - A12013.java - A12019.java -
           A12021.java - A12022.java - A12024.java - A12029.java - A12030.java - A12040.java - A12041.java - A12043.java - A12045.java - A12046.java - A12047.java -
```

Figure 5.6: The A1 files imported to JPlag

Query ID	Relevant File	Total Relevant	Similarity Value
F1	F2	1	100.0%
F3	F4	1	100.0%

the top 10 files returned for Q1. The files are ranked based on the cosine similarity value. Each file in the ranked list was scrutinised by a human judge, such that all the returned files were manually checked for similarity with the query in order to identify the file pairs which contained similarity that could be regarded as suspicious. This revealed that relevant file F1 is similar to the query 1, whereas file F5 contained parts similar to the query 1 but the file was not detected by JPlag.

Table 5.5 presents the results of the query (2: A12250). The table shows the top 10 files returned as potentially most similar to the query as gauged by Cosine similarity. There are two similar files to the query, and these files were returned with ranking 1 and ranking 7. The file that is ranked 1 is more similar to the query file than the file in position 7.

Table 5.4: Query 1 (F1): Number of relevant files and similarity file rank

Rank	Relevant Files	Similarity
1	F1	1
2	F2	0
3	F3	0
4	F4	0
5	F5	1
6	F6	0
7	F7	0
8	F8	0
9	F9	0
10	F10	0

Table 5.5: Query 2 (F3): Number of relevant files and similarity file rank

Rank	Relevant Files	Similarity
1	F1	1
2	F2	0
3	F3	0
4	F4	0
5	F5	0
6	F6	0
7	F7	1
8	F8	0
9	F9	0
10	F10	0

5.4.4 Evaluation Stage

In Phase two, stage 4, the application of the MFM framework to plagiarism detection was assessed using the evaluation measures *Precision*, *Recall* and the $F\!-\!measure$. The similarity values given to files in the ranked list differentiate between potentially relevant and non-relevant files for a given query, supplemented by human inspection as shown in Table 5.6.

Table 5.6: Evaluation measures for Q1 and Q2

Query ID	Precision	Recall	f-measure
Query 1	0.33	1.00	0.50
Query 2	0.20	1.00	0.33

Some files were not detected by JPlag but were detected by our approach, which improved detection accuracy.

5.5 Conclusion

This chapter has introduced a novel framework for detecting similarity in source code files. The framework combines style metrics with analysis based on Singular Value Decomposition. It has been designed to extract informative features, apply deep text analysis principles and enhanced style analysis for plagiarism detection. The framework is based on re-implementation and refinement of robust and well established techniques for code similarity detection based on existing, Modified and Extended metrics.

The application of proposed framework has been illustrated (in section 5.4) with an example using a small corpus of source code. The MFM framework provides a feature extraction technique to shrink the high-dimensional vector space that results from fingerprint re-implementation. The evaluation of the proposed framework is based on $Recall$, $Precision$ and $F{-}measure$ calculation.

Chapter 6 presents thorough evaluations using the MFM framework described in this chapter. These evaluations were performed using SOCO, a large benchmark dataset which contains groups of plagiarised files.

Chapter 6

Results and Evaluation

6.1 Introduction

This chapter presents the main results of the proposed similarity detection framework presented in Chapter 5. The SOCO dataset is described, along with the subsets of similar files which were used for evaluating the performance of the proposed method compared to alternative methods.

In Chapter 4, two families of metrics were proposed which can be embedded in the MFM framework. These were named the Modified and Extended families. The Modified metrics are explained in section 4.3 and the Extended metrics are described in 4.4.

The chapter is organised as follows: section 6.2 describes the SOCO dataset, and 6.3 presents the results of evaluating plagiarism detection techniques derived using the MFM framework on the SOCO dataset. The performance of the MFM approach is evaluated using the *Precision*, *Recall* and $F-measures$. Section 6.3.2 compares the results of the evaluation to methods presented in the PAN@FIRE competition using the SOCO dataset [53]. The last section 6.4 explains the types of attacks that the MFM approach can detect and which other approaches it fails to detect.

6.2 Dataset Description: SOCO

The Internet has provided easy access to information; there are repositories, forums and blogs, all making source code available so that it can be read, copied, and modified. Programmers are encouraged to avoid reinventing the wheel and to reuse available source code. There is such a large volume of resources on the Web, that manually analysing suspicious code is not feasible, and thus automatic re-use detection systems are needed. While software companies can be seen to have a special interest in protecting their intellectual property, more than three quarters of 3,970 developers surveyed admitted to re-using source code they had found elsewhere without giving appropriate credit to the author of the code [105]. Therefore, it is important to consider academic environments as well as commercial programming environments to look for similarities in re-used source code which does not include appropriate citations (i.e. plagiarism). Since there can be a degree of similarity between student programming solutions - especially those that have been designed to solve a specific problem – detecting plagiarism in source code can be a difficult task.

6.2.1 Forum for Information Retrieval Evaluation

The "Forum for Information Retrieval Evaluation" (FIRE), was established in India in 2008, under the umbrella of "The Information Retrieval Society of India" (IRSI), 2004[1]. The aim of the society is to provide a common ground for learning, and exchange of knowledge and ideas between researchers. FIRE was needed as elec-

[1]http://www.irsi.res.in/

tronic documents increased 700% between 2000 and 2007. Information Retrieval (IR) systems were needed to access these data for research. FIRE aims to: provide a platform for developing and evaluating information access technologies by providing large scale test collections; provide infrastructure for comparing IR systems; explore new IR techniques; and explore mono- or cross-lingual document retrieval methods. The FIRE forum runs every year with the same aims and different tasks [55]. The tasks concerning detection in recent years were:

1. CL!TR @2011(Cross Language !ndian Text Reuse)
2. CL!NSS @2012-2013 (Cross Language !ndian News Story Search)
3. SOCO @2014 (source codece COde) in two programming languages: C/C++ and Java [54]
4. CL-SOCO @2015 (Cross-Language source codece COde Re-use) [52]
5. PR-SOCO @2016 (Personality Recognition in SOurce COde) [128] Regarding programming style and personality, Bishop-Clark [17] and Zhara [81] investigated the relation between the cognitive style and personality of the author and their computer programming style.

6.2.2 SOCO

SOCO is a task which is part of FIRE that involved identifying and distinguishing the most similar source code pairs in a large source code collection [42]. SOCO focuses on source code re-use detection. It contains monolingual source code developed in an academic environment. SOCO is a large source code dataset and contains code pairs that share great similarity, and hence may be instances of plagiarism. The files are taken from Google Code Jam 2012.

	A1	A2	B1	B2	C1	C2	Total
Java	3,241	3,093	3,268	2,266	124	88	12,080

Table 6.1: Number of source codes files in SOCO @2014 in groups: A1, A2, B1, B2, C1 and C2

6.2.3 Google Code Jam

Google hosts and administers Google Code Jam as an international programming competition[2], which comprises algorithmic problems to which the solution (in any programming language and development environment) must be given within a set time. It started in 2003 to identify the most talented engineers for Google to employ [29]. Several rounds are conducted each year, and these are usually called: the Qualification Round, Rounds 1A, 1B and 1C, Round 2, Round 3, and a Final Round [29]. The competition requires that participants develop similarity detection tools for detecting re-used source code pairs from the SOCO dataset. SOCO participants annotated several blocks of source code to identify cases where source code had been re-used. SOCO contains two sets of data, Training and Testing. The training data is a collection of programs written in the C and Java programming languages, and which were obtained from a previous study [6]. The testing data comprised source code taken from the 2012 edition of the Google Jam Contest [64] [46]. In the experiments described in this chapter, the testing data comprised a Java test collection. Six monolingual scenarios designated A1, A2, B1, B2, C1 and C2 were used, as detailed in Table 6.1. File names are made up of the name of their scenario together with an identifier. For example, file B10021 denotes scenario B1 and has the identifier number 0021. The total number of submitted files in C/C++ group was 19,895 and the number of Java files was 12,080 [54] [29].

[2]https://code.google.com/codejam/

6.3 Experimental Results

This section presents the results of the proposed framework (which merges the structure based approach with the SVD approach) using the SOCO dataset in chapter 5. The performance of the proposed framework is evaluated using Precision, Recall and F-measure, as described in section 5.2.2. The experimental methodology is presented in Chapter 5 - section 5.3. After SVD, dimensionality reduction was applied on each of the structure based metrics using 2 dimensions $k = 2$ [3]. Comparisons were performed after the plagiarism methods were applied to SOCO dataset.

- **JPlag Queries:** JPlag is a well-defined tool for detecting plagiarism and finding similarities in a set of source code in Java. JPlag works as a tool that takes different source code files and compares them on a pairwise basis, and computes the total similarity value. JPlag converts each source code file into a list of tokens and then applies a Greedy String Styling method to detect similar substrings of code [148]. JPlag provides the result as an HTML output and provides the similar source code fragments in similar file pairs. JPlag returns a main file along with its similar files. Each main file, as shown in (6.1), is treated as a query file for the experiments which are described in Section 6.3.1 and onward.

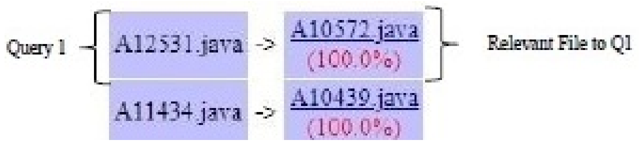


Figure 6.1: JPlag Queries

[3]The value $k = 2$ was chosen after an empirical study in which the similarity rankings of the top 10 files with values of k in the range 2 to 60 were compared

- **SOCO Queries**: As described in section 6.2, the SOCO dataset is split into two datasets: a training and testing dataset, and for each dataset there are relevance judgement files (i.e. known similar files)[4]. As can be seen in table 6.1, the number of source code files in the test set are significantly higher that the number of files in the training set. The files considered in the experiment as queries are from group A1, A2, B1, B2 and C2 in the test set. The reason for using the test dataset is because it contained groups of similar files which enabled the comparison between the proposed and other methods.

6.3.1 JPlag Queries: Results of the MFM-Modified and MFM-Extended Approaches

Table 6.2, 6.3, 6.4, 6.5, 6.6 and 6.7 present the experimental results using the six groups of similar source code files found in the SOCO dataset (A1, A2, B1, B2, C1 and C2). The first column shows the Query number. The second and the fifth column, Precision@mR shows the maximum value of *Precision* in the top R retrieved files for each query, where R is the total number of similar files for a query. The third and sixth column, R@10, show the value of *Recall* at the 10th ranked file. The fourth and the seventh column show the $F{-}measure$ values. The last row of each table shows the average of each column. In the discussion that follows, X.M indicates that data X was prepared using the MFM approach with the Modified metrics. X.E indicates that the data X was prepared using the MFM approach with the Extended metrics. These two approaches will be refer to as the 'MFM-Modified' and 'MFM-Extended' approaches respectively. For example, A1.M indicates applying the MFM-Modified approach to the A1 dataset, and A1.E indicates applying the MFM-Extended approach to the A1 dataset.

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

Table 6.2, 6.3, 6.4, 6.5, 6.6, and 6.7 show the results when performing evaluations using the queries from the A1, A2, B1, C1, C2 datasets, respectively using the MFM-Modified and the MFM-Extended approaches.

Table 6.2: Group A1, with JPlag Queries from the MFM-Modified and MFM-Extended approaches

	A1.M			A1.E		
	P@mR	R@10	F-measure	P@mR	R@10	F-measure
Q1	0.50	1.00	0.67	0.50	1.00	0.67
Q2	1.00	1.00	1.00	1.00	1.00	1.00
Q3	1.00	0.33	0.50	1.00	0.33	0.50
Q4	1.00	1.00	1.00	1.00	1.00	1.00
Q5	1.00	1.00	1.00	1.00	1.00	1.00
Q6	0.50	1.00	0.67	0.50	1.00	0.67
Q7	0.50	1.00	0.67	0.50	1.00	0.67
Q8	0.50	1.00	0.67	0.50	1.00	0.67
Q9	0.50	1.00	0.67	0.50	1.00	0.67
Q10	0.33	1.00	0.50	1.00	1.00	1.00
Q11	0.33	1.00	0.50	1.00	1.00	1.00
Q12	0.50	1.00	0.67	1.00	1.00	1.00
Q13	0.50	1.00	0.67	1.00	1.00	1.00
Average	**0.63**	**0.95**	**0.71**	**0.81**	**0.95**	**0.83**

Table 6.3: Group A2, with JPlag Queries when applying the MFM-Modified and MFM-Extended Approaches

	A2.M			A2.E		
Q1	P@mR	R@10	F-measure	P@mR	R@10	F-measure
Q2	0.50	1.00	0.67	0.50	1.00	0.67
Q3	0.50	0.50	0.50	0.50	0.50	0.50
Q4	0.50	1.00	0.67	0.50	1.00	0.67
Q5	0.50	1.00	0.67	0.50	1.00	0.67
Q6	0.50	1.00	0.67	0.50	1.00	0.67
Q7	0.33	1.00	0.50	0.50	1.00	0.67
Q8	1.00	1.00	1.00	1.00	1.00	0.67
Q9	0.50	1.00	0.67	0.50	1.00	0.67
Q10	0.50	1.00	0.67	1.00	1.00	1.00
Q11	0.50	1.00	0.67	1.00	1.00	1.00
Q12	0.50	0.50	0.50	0.50	0.50	0.50
Q13	0.33	1.00	0.50	1.00	1.00	1.00
Q14	0.33	1.00	0.50	1.00	1.00	1.00
Q15	0.50	0.50	0.50	0.50	0.50	0.50
Q16	0.50	1.00	0.67	1.00	1.00	1.00
Q17	0.50	1.00	0.67	1.00	1.00	1.00
Q18	0.50	1.00	0.67	1.00	1.00	1.00
Average	**0.50**	**0.91**	**0.63**	**0.74**	**0.91**	**0.77**

Table 6.4: Group B1, with JPlag Queries when applying the MFM-Modified and MFM-Extended Approaches

	B1.M			B1.E		
	P@mR	R@10	F-measure	P@mR	R@10	F-measure
Q1	0.20	0.50	0.29	0.50	0.50	0.50
Q2	1.00	1.00	1.00	1.00	1.00	1.00
Q3	0.50	1.00	0.67	1.00	1.00	1.00
Q4	0.33	1.00	0.50	0.50	1.00	0.67
Q5	0.50	1.00	0.67	0.50	1.00	0.67
Q6	0.50	1.00	0.67	0.50	1.00	0.67
Q7	1.00	1.00	1.00	1.00	1.00	1.00
Q8	1.00	1.00	1.00	1.00	1.00	1.00
Q9	0.50	1.00	0.67	1.00	1.00	1.00
Q10	1.00	1.00	1.00	1.00	1.00	1.00
Q11	1.00	1.00	1.00	1.00	1.00	1.00
Q12	0.50	1.00	0.67	1.00	1.00	1.00
Q13	0.50	1.00	0.67	1.00	1.00	1.00
Q14	1.00	1.00	1.00	1.00	1.00	1.00
Average	**0.68**	**0.96**	**0.77**	**0.86**	**0.96**	**0.89**

Table 6.5: Group B2, with JPlag Queries when applying the MFM-Modified and MFM-Extended Approaches

	B2.M			B2.E		
	P@mR	R@10	F-measure	P@mR	R@10	F-measure
Q1	1.00	1.00	1.00	1.00	1.00	1.00
Q2	0.50	1.00	0.67	0.50	1.00	0.67
Q3	0.50	0.50	0.50	1.00	0.50	0.67
Q4	0.33	1.00	0.50	0.33	1.00	0.50
Q5	0.33	1.00	0.50	1.00	1.00	1.00
Q6	0.50	1.00	1.00	1.00	1.00	1.00
Average	**0.53**	**0.92**	**0.69**	**0.81**	**0.92**	**0.81**

Table 6.6: Group C1, with JPlag Queries when applying the MFM-Modified and MFM-Extended Approaches

	C1.M			C1.E		
	P@mR	R@10	F-measure	P@mR	R@10	F-measure
Q1	1.00	1.00	1.00	1.00	1.00	1.00
Q2	0.50	1.00	0.67	1.00	1.00	1.00
Q3	0.50	1.00	0.67	0.50	1.00	0.67
Q4	1.00	1.00	1.00	1.00	1.00	1.00
Q5	0.50	1.00	0.67	1.00	1.00	1.00
Average	**0.70**	**1.00**	**0.80**	**0.90**	**1.00**	**0.93**

Table 6.7: Group C2, with JPlag Queries when applying the MFM-Modified and MFM-Extended Approaches

	C2.M			C2.E		
	P@mR	R	F-measure	P@mR	R	F-measure
Q1	1.00	1.00	1.00	1.00	1.00	1.00
Q2	0.50	1.00	0.67	1.00	1.00	1.00
Q3	1.00	0.50	0.67	1.00	0.50	0.67
Q4	1.00	1.00	1.00	1.00	1.00	1.00
Average	**0.83**	**0.83**	**0.83**	**1.00**	**0.83**	**0.92**

Table 6.8: The overall results are drawn from previous tables when applying the MFM-Modified and MFM-Extended Approaches showing Precision, Recall and F-measure averages

	X.Modified			X.Extended					
	P	R	F-measure	P	R	F-measure	P (E vs M)	R (E vs M)	F-measure (E vs M)
A1	0.630	0.950	0.710	0.810	0.950	0.830	+0.180	0.000	+0.120
A2	0.500	0.910	0.630	0.740	0.910	0.770	+0.240	0.000	+0.140
B1	0.680	0.960	0.770	0.860	0.960	0.890	+0.180	0.000	+0.120
B2	0.530	0.920	0.690	0.810	0.920	0.810	+0.280	0.000	+0.120
C1	0.700	1.000	0.800	0.900	1.000	0.930	+0.200	0.000	+0.130
C2	0.830	0.830	0.830	0.830	0.830	0.920	+0.000	0.000	+0.090
Average	0.645	0.928	0.738	0.825	0.928	0.858	+0.180	0.000	+0.120

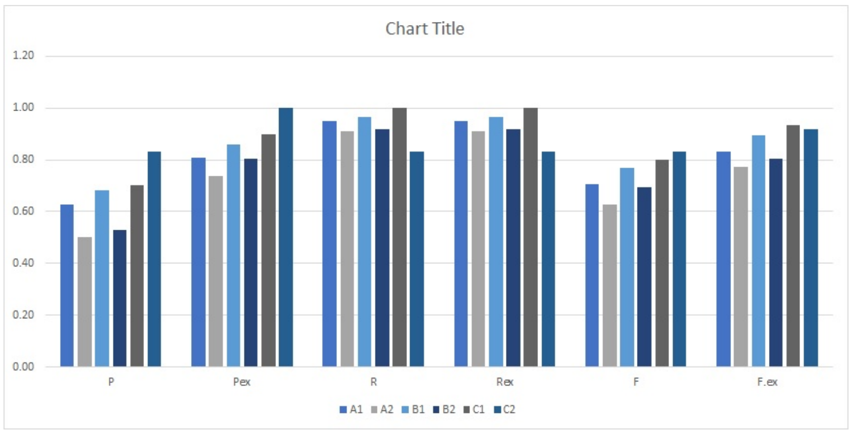


Figure 6.2: The output results from JPlag for the A1 dataset queries and their similar files.

Table 6.8 shows the average results for each dataset, and the final row of the table shows the Average performance of the MFM-Modified and MFM-Extended approaches across all queries. The last column of Table 6.8 shows the difference in performance when using the MFM-Modified and MFM-Extended approaches. Figure 6.2 illustrates the results shown in Table 6.8.

The results shown in Table 6.8 reveal that overall, when adopting the MFM-Extended approach, similarity detection performance was consistently higher in respect of Precision and F-measure. Recall was not affected. In particular, there was an increase of 0.18 points in average Precision, and 0.12 points in the F-measure when using the MFM-Extended approach as opposed to the MFM-Modified approach. The results thus indicate that using the MFM-Extended approach, more similar files are returned when a user is trying to detect similar source code files.

6.3.2 SOCO Queries: Results of the MFM-Modified and MFM-Extended Approaches

This section provides a description of seven methods which were compared with plagiarism detection techniques derived from the MFM framework. All seven methods were applied to SOCO dataset. Five of the methods are described by Flores at al. [54] and the other two methods are benchmark source code similarity detection methods. Note that the next section summarises the approaches which are described by Flores et al. [54]; details of those approaches were limited and the paper did not publish further background or analysis.

6.3.2.1 Participation Overview

The SOCO competition (see section 6.2) resulted in five solutions each proposed by a different team. The results of the MFM approach are compared with the results of each team, since the same datasets were used for the experiments carried out to evaluate the MFM approach. The details of each team and their solutions are provided below:

- **UAEM** [59]: Autonomous University of the State of Mexico. This team developed a model for source code detection which contained 4 phases: in the first phase, lexical items and whitespaces are removed. The second phase uses a similarity measurement to compute the similarity value for each source code compared to other source codes. The third phase ranks the distances between the files based on their similarity values to other files. The fourth phase is the decision phase which returns the files whose distance measure is less than 0.45.
- **UAM-C** [1]: Universidad Autonoma Metropolitana - Unidad Cuajimalpa

analysed the source code with respect to 3 different types of similarity: lexical, structural and stylistic. For lexical analysis, they considered a bag of 3-grams [5], excluding the programming language's reserved words. The authors proposed techniques for identifying structural similarities. For detecting stylistic changes, the counts of the style attributes such as number of spaces and number of upper case characters were considered. For each category (lexical, structural and stylistic) they calculated a similarity value. The authors also trained a supervised approach for detecting similar source code files in the corpus.

- **DCU** [58]: Dublin City University used an information retrieval approach for detecting plagiarised source code files. They used an Abstract Syntax Tree (AST) of source code and created a model for source code file indexing.
- **Rajat** [54]: proposed a method based on a string comparison technique. Their approach calculates the number of common lines and the total number of lines. The results obtained are based on the similarity value.
- **Apoorv** [54]: proposed a method following a string matching technique and similarity measurement. The selected files that had a high similarity value were considered to be a similar.

The MFM-Modified and MFM-Extended approaches were also compared with the following two benchmark methods:

- **Baseline 1** [125]: which is a greedy string tiling algorithm to compare the string in the detected source code file.
- **Baseline 2** [53]: which consists of a character 3-gram based model, and computes the cosine similarity value between file pairs.

[5]An n-gram is a contiguous sequence of n items from a given sample of text or speech.

Table 6.9, 6.10, 6.11, 6.12 and 6.13 present the results of the performance of the MFM detection methods and other algorithms when applied to the SOCO (A1, A2, B1, B2, C1 and C2) dataset.

Table 6.9: Group A1, with SOCO Queries from MFM-Modified and MFM-Extended Approaches

	A1.M			A1.E		
	P@mR	R@10	F-measure	P@mR	R@10	F-measure
Q1	0.50	1.00	0.67	1.00	1.00	1.00
Q2	0.50	1.00	0.67	1.00	1.00	1.00
Q3	1.00	1.00	1.00	1.00	1.00	1.00
Q4	0.50	1.00	0.67	0.50	1.00	0.67
Q5	0.50	1.00	0.67	0.50	1.00	0.67
Q6	0.50	1.00	0.67	0.50	1.00	0.67
Q7	1.00	1.00	1.00	0.50	1.00	0.67
Q8	0.50	1.00	0.67	0.50	1.00	0.67
Q9	0.50	1.00	0.67	0.50	1.00	0.67
Q10	1.00	1.00	1.00	0.50	1.00	0.67
Q11	0.50	1.00	0.67	0.50	1.00	0.67
Q12	1.00	1.00	1.00	0.50	1.00	1.00
Q13	0.50	1.00	0.67	1.00	1.00	1.00
Q14	0.50	1.00	0.67	1.00	1.00	1.00
Q15	0.50	1.00	0.67	0.50	1.00	0.67
Q16	0.50	1.00	0.67	0.50	1.00	0.67
Q17	0.50	1.00	0.67	0.50	1.00	0.67
Average	**0.62**	**1.00**	**0.75**	**0.65**	**1.00**	**0.78**

Table 6.10: Group A2, with SOCO Queries from MFM-Modified and MFM-Extended Approaches

	A2.M			A2.E		
	P	R	F-measure	P	R	F-measure
Q1	0.50	1.00	0.67	0.50	1.00	0.67
Q2	0.50	1.00	0.67	0.50	1.00	0.67
Q3	1.00	1.00	1.00	1.00	1.00	1.00
Q4	0.50	1.00	0.67	0.50	1.00	0.67
Q5	1.00	1.00	1.00	1.00	1.00	1.00
Q6	0.50	1.00	0.67	0.50	1.00	0.67
Q7	0.50	1.00	0.67	0.50	1.00	0.67
Q8	0.50	1.00	0.67	0.50	1.00	0.67
Q9	0.50	1.00	0.67	0.50	1.00	0.67
Q10	0.50	1.00	0.67	0.50	1.00	0.67
Q11	0.33	1.00	0.50	1.00	1.00	1.00
Q12	0.50	1.00	0.67	0.50	1.00	0.67
Q13	0.50	1.00	0.67	0.50	1.00	0.67
Average	**0.56**	**1.00**	**0.71**	**0.62**	**1.00**	**0.74**

Table 6.11: Group B1, with SOCO Queries from MFM-Modified and MFM-Extended Approaches

	B1.M			B1.E		
	P@mR	R@10	F-measure	P@mR	R@10	F-measure
Q1	0.20	0.50	0.29	0.33	1.00	0.50
Q2	0.50	1.00	0.67	0.50	1.00	0.67
Q3	0.50	1.00	0.67	0.50	1.00	0.67
Q4	0.50	1.00	0.67	0.50	1.00	0.67
Q5	0.50	1.00	0.67	0.50	1.00	0.67
Q6	1.00	1.00	1.00	0.50	1.00	0.67
Q7	0.33	1.00	0.50	0.50	1.00	0.67
Q8	0.50	1.00	0.67	0.50	1.00	0.67
Q9	0.50	1.00	0.67	0.50	1.00	0.67
Q10	0.50	1.00	0.67	1.00	1.00	1.00
Q11	1.00	1.00	1.00	1.00	1.00	1.00
Q12	0.50	1.00	0.67	0.50	1.00	0.67
Q13	0.50	1.00	0.67	0.50	1.00	0.67
Q14	0.50	1.00	0.67	0.50	1.00	0.67
Q15	0.50	1.00	0.67	0.50	1.00	0.67
Q16	0.50	1.00	0.67	0.50	1.00	0.67
Q17	0.50	1.00	0.67	0.50	1.00	0.67
Q18	0.50	1.00	0.67	1.00	1.00	1.00
Q19	1.00	1.00	1.00	0.50	1.00	0.67
Q20	0.50	1.00	0.67	1.00	1.00	1.00
Average	**0.55**	**0.98**	**0.69**	**0.59**	**1.00**	**0.73**

Table 6.12: Group B2, with SOCO Queries from MFM-Modified and MFM-Extended Approaches

	B2.M			B2.E		
	P@mR	R@10	F-measure	P@mR	R@10	F-measure
Q1	1.00	1.00	1.00	1.00	1.00	1.00
Q2	0.50	1.00	0.67	0.50	1.00	0.67
Q3	0.50	1.00	0.67	0.50	1.00	0.67
Q4	0.33	1.00	0.50	0.50	1.00	0.67
Q5	0.50	1.00	0.67	0.50	1.00	0.67
Q6	0.50	1.00	1.00	0.50	1.00	1.00
Q8	0.50	1.00	0.67	0.50	1.00	0.67
Average	**0.55**	**1.00**	**0.74**	**0.57**	**1.00**	**0.76**

Table 6.13: Group C2, with SOCO Queries from MFM-Modified and MFM-Extended Approaches

	C2.M			C2.E		
	P@mR	R@10	F-measure	P@mR	R@10	F-measure
Q1	1.00	1.00	1.00	1.00	1.00	1.00
Q2	0.50	1.00	0.67	1.00	1.00	1.00
Q3	1.00	1.00	1.00	0.50	1.00	0.67
Average	**0.83**	**1.00**	**0.89**	**0.83**	**1.00**	**0.89**

Table 6.14: Results of the MFM-Modified, MFM-Extended Approaches, and of methods proposed by participants in the SOCO competition using the SOCO dataset and Queries

	A1			**A2**			**B1**			**B2**			**C2**		
	F1	P	R	F1	P	R	F1	P	R	F1	P	R	F1	P	R
MFM-M	0.750	0.620	1.00	0.705	0.564	1.00	0.690	0.550	0.980	0.740	0.550	1.00	0.890	0.830	1.00
MFM-E	0.780	0.650	1.00	0.744	0.615	1.00	0.730	0.590	1.00	0.760	0.570	1.00	0.890	0.830	1.00
UAEM	0.25	0.143	1.00	0.234	0.133	1.00	0.234	0.193	1.00	0.248	0.142	1.00	0.5	0.333	1.00
UAM-C	0.755	0.607	1.00	0.058	0.03	0.957	0.021	0.01	0.973	0.027	0.014	1.00	0.111	0.019	0.143
DCU	0.667	0.5	1.00	0.676	0.511	1.00	0.702	0.541	1.00	0.687	0.523	1.00	0.667	0.5	1.00
Baseline 1	0.324	0.6	0.222	0.388	0.65	0.277	0.237	0.55	0.151	0.556	0.75	0.441	0.824	0.7	1.00
Baseline 2	0.529	0.439	0.667	0.737	0.434	0.702	0.559	0.472	0.685	0.568	0.463	0.735	0.667	0.5	1.00
Apoorv	0.025	0.013	0.759	0.021	0.011	0.702	0.038	0.019	0.863	0.25	0.013	0.853	0.275	0.159	1.00
Rajat	0.352	0.241	0.648	0.337	0.232	0.617	0.54	0.425	0.74	0.369	0.275	0.559	0.718	0.56	1.00

Table 6.15: The average F-measure of all queries across the dataset groups

Team Name	A1	A2	B1	B2	C2
MFM-Modified	**0.750**	**0.705**	**0.690**	**0.738**	**0.889**
MFM-Extended	**0.780**	**0.741**	**0.730**	**0.762**	**0.889**
UAEM	0.250	0.234	0.234	0.248	0.500
UAM-C	0.755	0.058	0.021	0.027	0.111
DCU	0.667	0.676	0.702	0.687	0.667
Baseline 1	0.324	0.388	0.237	0.556	0.824
Baseline 2	0.529	0.737	0.559	0.568	0.667
Apoorv	0.025	0.021	0.038	0.250	0.275
Rajat	0.352	0.337	0.540	0.369	0.718

Table 6.15 presents the overall results in F-measure across all datasets. Note that UAM-C team gives relatively poor results For groups A2, B1, B2 and C2; this can be attributed to their use of machine learning techniques which are effective only on large datasets. The MFM-Extended approach achieved the highest F-measure value compared to the other methods. In particular, when applying the MFM-Extended approach to detect similar files in the A1 dataset, the F-measure value reached 0.780, which was the highest compared to all other methods. Second place was the method proposed by the UAM-C group, which achieved an F-measure of 0.755.

Similarly, the F-measures for the MFM-Extended approach were higher than the competitors for the datasets: A2 (0.741 vs. next highest was Baseline 2 with 0.737); B1 (0.730 vs. next highest was DCU 0.702); B2 (0.762 vs. next highest was the MFM-Modified approach with 0.738); and C2 (MFM-Extended and MFM-Modified tied with 0.889, vs. the next highest was Baseline 1 with 0.824).

Table 6.16: The overall average of evaluation calculation across the dataset groups

Team Name	F-measure Average	Precision Average	Recall Average
MFM-Modified	0.7534	0.6228	1.00
MFM-Extended	0.7807	0.6517	1.00
UAEM	0.2932	0.1888	1.00
UAM-C	0.1944	0.136	0.8146
DCU	0.6798	0.515	1.00
Baseline 1	0.4658	0.65	0.4182
Baseline 2	0.612	0.4616	0.7578
Apoorv	0.1218	0.043	0.8354
Rajat	0.4632	0.3466	0.7128

6.4 The Failure Analysis

There are some drawbacks of JPlag when it comes to detecting similarities in Java source code. JPlag document [125] analysed some successful and non-successful plagiarising attacks. JPlag is token based and the authors focus on techniques for detecting 'local confusion' in a fragment of code. Local confusion is a strategy for plagiarism whereby short fragments of the original code are changed in such a way that the overall functionality of the program is unaffected [125].

There are 3 types of plagiarism attack as discussed in [125], and they are listed below:

1. **Futile Attack:** this is the kind of attack (attempt at plagiarism) that is very unlikely to work (i.e. go undetected) because it does not modify the tokens that are considered by JPlag. Table 6.17 presents 9 types of futile attacks which do not affect JPlag's ability to detect plagiarism. For example, modifying comments does not disguise plagiarism since JPlag removes the comments when parsing. Table 6.17 shows the number of programs that had attempted each type of attack in one study, followed by (after the slash), the number that were initially undetected; most of the attacks were successfully

detected.

2. **Granularity-Sensitive Attacks:** this is the kind of attack that relates to Java classes which consist of declarations of methods and variables. How successful this type of attack is will depend on the size of the section that has been reordered (i.e. its granularity) - reordering large chunks of text is more likely to be detected than multiple small changes. Table 6.18 presents two types of attacks and shows that a minority did avoid detection by JPlag.

3. **Locally Confusing Attacks:** this is the kind of attack that relates to the logic and the control structure of the code, and it is the type of attack most likely to be successful (avoid detection). Table 6.19 shows ten types of attacks which JPlag struggled to detect, especially the last three.

Table 6.17: Futile Attack [125]

Futile Attack		
	Attack Name	**# Program**
1	Modification of code formatting	48/0
2	Insertion, modification or deletion of comments	30/0
3	Translation from English to German or vice verse	19/0
4	Modification of program output or of its formatting	14/2
5	Change names of variables, methods or class	44/0
6	Split/merge of variables deceleration	6/0
7	Insertion, modification or deletion of modifiers	6/0
8	Modification of constant values	3/0
9	No change at all	4/0

Table 6.18: Granularity-Sensitive Attacks [125]

Granularity-Sensitive Attacks		
	Attack Name	**# Program**
1	Reordering within blocks of variable declaration	25/6
2	Global reordering of variables and method declarations	30/3

Table 6.19: Locally Confusing Attacks [125]

Locally Confusing Attacks		
	Attack Name	**# Program**
1	Modification of control structure	35/35
2	Temporary variables and sub-expressions	28/28
3	Inlining and refactoring	20/16
4	Modification of scope	9/9
5	Statement reordering in absence of data dependencies	8/6
6	Mathematical identities	5/2
7	Introduction of bugs on purpose	5/3
8	Modification of data structure	6/5
9	Redundancy	15/15
10	Structural redesign of code	3/3

Table 6.20: Attacks identified by applying the MFM-Extended approach to the SOCO dataset which were not identified by JPlag or the Proposed Modified approach

Group	Name of Attack	Metrics
A1.M		
A1.E	Mathematical Identity	E.STY4 and E.STY6
A2.M		
A2.E		
B1.M		
B1.E	Modification of control structure	STY3a, STY3b, PRO5a, PRO5b, PSM9 and PSM10
B2.M		
B2.E		
C2.M		
C2.E		

Tables 6.17, 6.18 and 6.19 are taken from Prechelt [125]. A closer inspection of the results presented in section 6.3 shows that tha MFM-Extended approach exposes attacks that other tools fail to detect. Table 6.20 lists instances of attacks that were found in the SOCO dataset using the MFM-Extended approach that were not detected by other methods. The A1 group has 17 queries, and observing the

MFM-Extended approach results (A1.E), 4 out of 17 files were detected as having an attack that relates to Locally Confusing JPlag Attacks [125]. Regarding the B1 group, which has 20 queries, approach B1.E identified 5 files that also contain Locally Confusing attacks. The two main types of locally confusing attacks detected are: 1- Modification of control structure and 2- Mathematical identities.

6.5 Conclusion

The chapter presents the results of evaluating the MFM framework (see Chapter 5 for details) when adopting the Modified and Extended programming metrics. In order to compare the performance of the framework using each set of metrics, experiments were conducted using a number of queries. The MFM approach was also compared to other plagiarism detection methods. The experimental results demonstrated that the MFM framework was able to detect similar files that JPlag failed to detect. In addition, the MFM-Extended approach outperformed the MFM-Modified approach when comparing the results obtained using SOCO queries. The next chapter discusses the overall thesis including the summary, contributions and limitations of the research. It also discusses future work.

Chapter 7

Conclusion

7.1 Introduction

The thesis investigates the suitability of style metrics for source code plagiarism detection. New style metrics are proposed which are suitable for source code plagiarism detection. The proposed metrics are embedded into a framework which is evaluated using a large benchmark source code dataset. The performance of the proposed framework is also compared against other plagiarism detection methods. This chapter summarises the main contributions of this thesis and emphasises the significance of the contributions in relation to existing research. Finally, it outlines some suggestions for future work in this area.

7.2 Summary of the Research

The main aim of this thesis is to examine coding style analysis using a structure based approach for detecting source code plagiarism. In order to achieve this goal, the research describe in this thesis has three aspects:

1: Relevant literature and identification of suitable dataset. In chapter 2, the relevant literature was reviewed and gaps in source code plagiarism detection were identified. It was found that not many studies exist which have covered style based source code plagiarism detection techniques. The results of the current study contribute to the literature in this area. There are also few benchmark datasets which are suitable for source code similarity detection studies. In chapter 3, the suitability of the BlackBox dataset was investigated via an exploratory study using two techniques - Grouping and Visualisation.

2: Design and Implementation of new style metrics. Programming style metrics using a structure based approach were implemented, and these are described in Chapter 4. Two new families of style metrics were proposed: Modified and Extended.

3: Design and evaluation of a source code plagiarism detection framework. In chapter 5, a framework for source code similarity detection framework was proposed. This was modelled on a well established statistical approach to document analysis and applied Singular Value Decomposition to metric-file matrices in order to reduce the dimensionality of the data and to reveal the underlying relations between files. In Chapter 6, the results of the framework were presented and discussed with reference to standard JPlag and SOCO queries. The results were then compared to other source code plagiarism detection methods using the benchmark SOCO dataset, and were evaluated using the $Precision$, $Recall$ and $F - measure$

evaluation measures typically applied in the context of plagiarism detection.

7.3 Study Contributions

This thesis has three novel contributions.

1. The first contribution is related to BlackBox characteristics. BlackBox is a massive dataset [22]. In order to identify a valid sample to work with in this study, the research has followed a specific procedure (see chapter 3). Five groups were analysed, two of which were found to contain some similar files. The number of plagiarised file pairs/groups in BlackBox was not enough to thoroughly evaluate the performance of plagiarism detection methods based on style analysis. For this reason, BlackBox was not used for evaluating the plagiarism detection framework proposed in this thesis, but the research did identify datasets that can be used by other researchers who are working on source code similarity tasks.

2. Existing programming style metrics (Layout, Style and Structure) are currently used to detect authorship. The research described in this thesis differs from these current trends in that existing programming style metrics were adapted for source code plagiarism detection. In order to adapt these metrics for this study aim, we have re-implemented existing programming style metrics and derived two new structure based approaches based on Modified and Extended families of metrics (chapter 4). This addresses limitations in existing coding style analysis to improve the similarity detection between source code files.

3. The most important contributions of the study is a proposed novel framework (the 'metric-file matrix' framework) for statistical analysis of source code simi-

larities based on coding style metrics (chapter 5). In the MFM framework, Singular Value Decomposition is applied to 'metric-file matrices' derived from coding style metrics. This leads to more effective methods of plagiarism detection as assessed via the evaluation measures: *Precision*, *Recall* and $F-measure$ (chapter 6). For example, the use of the Extended family of metrics (chapter 4) within MFM framework can identify two plagiarism attacks (Modification of control structure attack and Mathematical Identity attack) that other tools such as the JPlag tool could not detect.

To the author's knowledge, this research is the first attempt to address the limitations in source code similarity detection using existing programming style metrics.

7.4 Limitations and Future Work

This study focuses on Java programming language. Therefore, future work includes adopting the proposed framework and applying it for other programming languages. In order to do this, the style metrics will need to be edited to be suitable for the specific programming language. The generic nature of the MFM framework means that such changes to families of style metrics can be readily applied.

Future work also includes applying the proposed framework to larger source code datasets which are available online, such as the GitHub repository. Given that the framework is suitable for SOCO, which is a large dataset, it is expected that it should be effective in detecting similar source code in other large datasets. Other studies can also include datasets composed of academic submissions for source code detection. Every tool needs an interface, and future work also includes creating an interface which allows the user to view the similar files and code fragments.

References

[1] A. Ramırez-de-la Cruz, G. Ramırez-de-la Rosa, C. S.-S. W. A. L.-R. H. J.-S. C. R.-L. [n.d.], 'Detection of source code reuse by means of combining different types of epresentations', *FIRE [4]* .

[2] Ajmal, O., Missen, M. S., Hashmat, T., Moosa, M. and Ali, T. [2013], Eplag: A two layer source code plagiarism detection system, *in* 'Digital Information Management (ICDIM), 2013 Eighth International Conference on', IEEE, pp. 256–261.

[3] Alter, O., Brown, P. O. and Botstein, D. [2000], 'Singular value decomposition for genome-wide expression data processing and modeling', *Proceedings of the National Academy of Sciences* **97**(18), 10101–10106.

[4] Ammann, M. and Cameron, R. D. [1994], Measuring program structure with inter-module metrics, *in* 'Computer Software and Applications Conference, 1994. COMPSAC 94. Proceedings., Eighteenth Annual International', IEEE, pp. 139–144.

[5] Arabyarmohamady, S., Moradi, H. and Asadpour, M. [2012], A coding style-based plagiarism detection, *in* 'Interactive Mobile and Computer Aided Learning (IMCL), 2012 International Conference on', IEEE, pp. 180–186.

[6] Arwin, C. and Tahaghoghi, S. M. [2006], Plagiarism detection across programming languages, *in* ‘Proceedings of the 29th Australasian Computer Science Conference-Volume 48’, Australian Computer Society, Inc., pp. 277–286.

[7] Austin, M. J. and Brown, L. D. [1999], ‘Internet plagiarism: Developing strategies to curb student academic dishonesty’, *The Internet and Higher Education* **2**(1), 21–33.

[8] Bandara, U. and Wijayarathna, G. [2013], ‘Source code author identification with unsupervised feature learning’, *Pattern Recognition Letters* **34**(3), 330–334.

[9] Barbosa, A. d. A., Costa, E. d. B. and Brito, P. H. [2018], Adaptive clustering of codes for assessment in introductory programming courses, *in* ‘International Conference on Intelligent Tutoring Systems’, Springer, pp. 13–22.

[10] Baxter, I. D., Yahin, A., Moura, L., Sant’Anna, M. and Bier, L. [1998], Clone detection using abstract syntax trees, *in* ‘Software Maintenance, 1998. Proceedings., International Conference on’, IEEE, pp. 368–377.

[11] Belta, C. and Kumar, V. [2002], ‘An SVD-based projection method for interpolation on se (3)’, *IEEE transactions on Robotics and Automation* **18**(3), 334–345.

[12] Bengio, Y. et al. [2009], ‘Learning deep architectures for AI’, *Foundations and Trends in Machine Learning* **2**(1), 1–127.

[13] Berry, M. W. [1992], ‘Large-scale sparse singular value computations’, *The International Journal of Supercomputing Applications* **6**(1), 13–49.

[14] Berry, M. W., Dumais, S. T. and OBrien, G. W. [1995], ‘Using linear algebra for intelligent information retrieval’, *SIAM Review* **37**(4), 573–595.

[15] Berry, R. E. and Meekings, B. A. [1985], 'A style analysis of C programs', *Communications of the ACM* **28**(1), 80–88.

[16] Biemann, C. and Nygaard, V. [2010], Crowdsourcing wordnet, *in* 'The 5th International Conference of the Global WordNet Association (GWC-2010)'.

[17] Bishop-Clark, C. [1995], 'Cognitive style, personality, and computer programming', *Computers in Human Behavior* **11**(2), 241–260.

[18] Bowyer, K. W. and Hall, L. O. [1999], Experience using "MOSS" to detect cheating on programming assignments, *in* 'Frontiers in Education Conference, 1999. FIE'99. 29th Annual', Vol. 3, IEEE, pp. 13B3–18.

[19] Breiman, L. [2001], 'Random forests', *Machine Learning* **45**(1), 5–32.

[20] Brin, S., Davis, J. and Garcia-Molina, H. [1995], Copy detection mechanisms for digital documents, *in* 'ACM SIGMOD Record', Vol. 24, ACM, pp. 398–409.

[21] Brown, N. C. and Altadmri, A. [2017], 'Novice Java programming mistakes: large-scale data vs. educator beliefs', *ACM Transactions on Computing Education (TOCE)* **17**(2), 7.

[22] Brown, N. C. C., Kölling, M., McCall, D. and Utting, I. [2014], Blackbox: A large scale repository of novice programmers' activity, *in* 'Proceedings of the 45th ACM technical symposium on Computer Science Education', ACM, pp. 223–228.

[23] Burrows, S., Tahaghoghi, S. M. and Zobel, J. [2007], 'Efficient plagiarism detection for large code repositories', *Software: Practice and Experience* **37**(2), 151–175.

[24] Caliskan-Islam, A., Harang, R., Liu, A., Narayanan, A., Voss, C., Yamaguchi, F. and Greenstadt, R. [2015], De-anonymizing programmers via code stylometry, *in* '24th USENIX Security Symposium (USENIX Security 15)', pp. 255–270.

[25] Cederberg, S. and Widdows, D. [2003], Using LSA and noun coordination information to improve the precision and recall of automatic hyponymy extraction, *in* 'Proceedings of the Seventh Conference on Natural Language Learning at HLT-NAACL 2003-Volume 4', Association for Computational Linguistics, pp. 111–118.

[26] Chen, X., Francia, B., Li, M., Mckinnon, B. and Seker, A. [2004], 'Shared information and program plagiarism detection', *IEEE Transactions on Information Theory* **50**(7), 1545–1551.

[27] Chilowicz, M., Duris, É. and Roussel, G. [2013], 'Viewing functions as token sequences to highlight similarities in source code', *Science of Computer Programming* **78**(10), 1871–1891.

[28] *Code Conventions for the Java TM Programming Language* [1999], `http://www.oracle.com/technetwork/java/codeconvtoc-136057.html`. (Accessed on 20/12/2016).

[29] CodeJam [2012], 'Qualification round 2012'.
URL: *https://code.google.com/codejam/contest/1460488/dashboard*

[30] Cohen, L., Manion, L. and Morrison, K. [2013], *Research Methods in Education*, Routledge.

[31] Conte, S. D., Dunsmore, H. E. and Shen, V. Y. [1986], *Software Engineering Metrics and Models*, Benjamin-Cummings Publishing Co., Inc.

[32] Cosma, G. and Joy, M. [2006], Source-code plagiarism: A UK academic perspective, Technical report, Department of Computer Science, University of Warwick.

[33] Cosma, G. and Joy, M. [2008], 'Towards a definition of source-code plagiarism', *IEEE Transactions on Education* **51**(2), 195–200.

[34] Culwin, F. and Lancaster, T. [2000*a*], 'A descriptive taxonomy of student plagiarism', *Awaiting publication, available from South Bank University, London* .

[35] Culwin, F. and Lancaster, T. [2000*b*], A review of electronic services for plagiarism detection in student submissions, *in* 'LTSN-ICS 1st Annual Conference', Citeseer, pp. 23–25.

[36] Culwin, F. and Lancaster, T. [2001], 'Plagiarism issues for higher education', *Vine* **31**(2), 36–41.

[37] Culwin, F. and Naylor, J. [1995], Pragmatic anti-plagiarism, *in* 'Proceedings 3rd All Ireland Conference on the Teaching of Computing, Dublin'.

[38] Dasgupta, C. [2010], That is not my program: investigating the relation between program comprehension and program authorship, *in* 'Proceedings of the 48th Annual Southeast Regional Conference', ACM, p. 103.

[39] Davies, S. R. [2018], 'An ethics of the system: Talking to scientists about research integrity', *Science and Engineering Ethics* pp. 1–19.

[40] Decoo, W. [2002], *Crisis on Campus: Confronting Academic Misconduct*, MIT Press.

[41] Delev, T. and Gjorgjevikj, D. [2017], 'Comparison of string matching based algorithms for plagiarism detection of source code'.

[42] *Detection of SOurce COde Re-use* [2014]. (Accessed on 18/07/2016).
URL: *http://users.dsic.upv.es/grupos/nle/soco/*

[43] Di Marco, A. and Navigli, R. [2013], 'Clustering and diversifying web search results with graph-based word sense induction', *Computational Linguistics* **39**(3), 709–754.

[44] Ding, H. and Samadzadeh, M. H. [2004], 'Extraction of java program fingerprints for software authorship identification', *Journal of Systems and Software* **72**(1), 49–57.

[45] DJurić, Z. and Gašević, D. [2012], 'A source code similarity system for plagiarism detection', *The Computer Journal* **56**(1), 70–86.

[46] Domin, C., Pohl, H. and Krause, M. [2016], 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', ACM, pp. 1173–1179.

[47] Ďuračík, M., Kršák, E. and Hrkút, P. [2018], Source code representations for plagiarism detection, *in* 'International Workshop on Learning Technology for Education in Cloud', Springer, pp. 61–69.

[48] Faidhi, J. A. and Robinson, S. K. [1987], 'An empirical approach for detecting program similarity and plagiarism within a university programming environment', *Computers & Education* **11**(1), 11–19.

[49] Fenn, D. J., Porter, M. A., Williams, S., McDonald, M., Johnson, N. F. and

Jones, N. S. [2011], 'Temporal evolution of financial-market correlations', *Physical Review E* **84**(2), 026109.

[50] Fenwick Jr, J. B., Norris, C., Barry, F. E., Rountree, J., Spicer, C. J. and Cheek, S. D. [2009], Another look at the behaviors of novice programmers, *in* 'ACM SIGCSE Bulletin', Vol. 41, ACM, pp. 296–300.

[51] Fitzgerald, S., Simon, B. and Thomas, L. [2005], Strategies that students use to trace code: an analysis based in grounded theory, *in* 'Proceedings of the first international workshop on Computing education research', ACM, pp. 69–80.

[52] Flores, E., Barrón-Cedeño, A., Moreno, L. and Rosso, P. [2015], 'Cross-language source code re-use detection using latent semantic analysis.', *J. UCS* **21**(13), 1708–1725.

[53] Flores, E., Barrón-Cedeño, A., Rosso, P. and Moreno, L. [2011], Towards the detection of cross-language source code reuse, *in* 'International Conference on Application of Natural Language to Information Systems', Springer, pp. 250–253.

[54] Flores, E., Rosso, P., Moreno, L. and Villatoro-Tello, E. [2014], Pan@fire: Overview of track on the detection of source code re-use, *in* 'Sixth Forum for Information Retrieval Evaluation (FIRE 2014), Bangalore, India'.

[55] *Forum for Information Retrieval Evaluation* [2008]. (Accessed on 10/05/2018).
URL: *https://www.isical.ac.in/ fire/2008/index.html*

[56] Freire, M. [2008], Visualizing program similarity in the AC plagiarism detection system, *in* 'Proceedings of the Working Conference on Advanced Visual Interfaces', ACM, pp. 404–407.

[57] Ganguli, M. [2003], *Making Use of JSP*, John Wiley & Sons.

[58] Ganguly, D. and Jones, G. J. [2014], Dcu@ fire-2014: an information retrieval approach for source code plagiarism detection, *in* 'Proceedings of the Forum for Information Retrieval Evaluation', ACM, pp. 39–42.

[59] Garcıa-Hernández, R. and Lendeneva, Y. [2014], 'Identification of similar source codes based on longest common substrings', *FIRE [4]* .

[60] Gitchell, D. and Tran, N. [1999], Sim: a utility for detecting similarity in computer programs, *in* 'ACM SIGCSE Bulletin', Vol. 31, ACM, pp. 266–270.

[61] Gondaliya, T. P., Joshi, H. D. and Joshi, H. [2014], 'Source code plagiarism detection 'SCPDet': A review', *International Journal of Computer Applications* **105**(17).

[62] Graven, O. H. and MacKinnon, L. M. [2008], 'A consideration of the use of plagiarism tools for automated student assessment', *IEEE Transactions on Education* **51**(2), 212–219.

[63] Grubb, P. and Takang, A. A. [2003], *Software Maintenance: Concepts and Practice*, World Scientific.

[64] Hammond, M. [2002], 'Cyber-plagiarism: are fe students getting away with words?'.

[65] Holter, N. S., Mitra, M., Maritan, A., Cieplak, M., Banavar, J. R. and Fedoroff, N. V. [2000], 'Fundamental patterns underlying gene expression profiles: simplicity from complexity', *Proceedings of the National Academy of Sciences* **97**(15), 8409–8414.

[66] Hope, D. and Keller, B. [2013], Maxmax: a graph-based soft clustering algorithm applied to word sense induction, *in* 'International Conference on Intelligent Text Processing and Computational Linguistics', Springer, pp. 368–381.

[67] Howard, R. M., Serviss, T. and Rodrigue, T. K. [2010], 'Writing from sources, writing from sentences', *Writing and Pedagogy* **2**(2), 177–192.

[68] Huang, Y. J., Powers, R. and Montelione, G. T. [2005], 'Protein NMR recall, precision, and f-measure scores (rpf scores): structure quality assessment measures based on information retrieval statistics', *Journal of the American Chemical Society* **127**(6), 1665–1674.

[69] Ikeda, H., Nagaoka, R., Lafond, M., Yoshizawa, S., Iwasaki, R., Maeda, M., Umemura, S.-i. and Saijo, Y. [2018], 'Singular value decomposition of received ultrasound signal to separate tissue, blood flow, and cavitation signals', *Japanese Journal of Applied Physics* **57**(7S1), 07LF04.

[70] Israel, G. D. [1992], *Determining sample size*, University of Florida Cooperative Extension Service, Institute of Food and Agriculture Sciences, EDIS.

[71] Jadud, M. C. [2005], 'A first look at novice compilation behaviour using bluej', *Computer Science Education* **15**(1), 25–40.

[72] Jadud, M. C. [2006], An exploration of novice compilation behaviour in BlueJ, PhD thesis, University of Kent.

[73] Jadud, M. C. and Dorn, B. [2015], Aggregate compilation behavior: Findings and implications from 27,698 users, *in* 'Proceedings of the Eleventh Annual International Conference on International Computing Education Research', ACM, pp. 131–139.

[74] Ji, J.-H., Woo, G. and Cho, H.-G. [2007], A source code linearization technique for detecting plagiarized programs, *in* 'ACM SIGCSE Bulletin', Vol. 39, ACM, pp. 73–77.

[75] JIANG, J.-h. and Ke, W. [2018], 'Similarity code file detection model based on frequent itemsets', *DEStech Transactions on Computer Science and Engineering* (CCNT).

[76] Johnson, J. H. [1993], Identifying redundancy in source code using fingerprints, *in* 'Proceedings of the 1993 Conference of the Centre for Advanced Studies on Collaborative Research: Software Engineering-Volume 1', IBM Press, pp. 171–183.

[77] Jones, E. L. [2001*a*], 'Metrics based plagarism monitoring', *Journal of Computing Sciences in Colleges* **16**(4), 253–261.

[78] Jones, E. L. [2001*b*], 'Metrics based plagarism monitoring', *Journal of Computing Sciences in Colleges* **16**(4), 253–261.

[79] Joy, M. and Luck, M. [1999], 'Plagiarism in programming assignments'.

[80] Joy, M., Sinclair, J., Boyatt, R., Yau, J.-K. and Cosma, G. [2013], 'Student perspectives on source-code plagiarism', *International Journal for Educational Integrity* **9**(1), 13–19.

[81] Karimi, Z., Baraani-Dastjerdi, A., Ghasem-Aghaee, N. and Wagner, S. [2016], 'Links between the personalities, styles and performance in computer programming', *Journal of Systems and Software* **111**, 228–241.

[82] Karnalim, O. and Budi, S. [2018], 'The effectiveness of low-level structure-based approach toward source code plagiarism level taxonomy', *arXiv preprint arXiv:1805.11035* .

[83] Karnalim, O. and Sulistiani, L. [2018], 'Which source code plagiarism detection approach is more humane?', *arXiv preprint arXiv:1809.08559* .

[84] Kernighan, B. W. and Plauger, P. J. [1978], 'The elements of programming style', *B, by Kernighan, Brian W.; Plauger, PJ New York: McGraw-Hill, c1978.* **1**.

[85] Keuning, H., Heeren, B. and Jeuring, J. [2017], 'Code quality issues in student programs. technical report series, no. uu-cs-2017-006. issn 0924-3275'.

[86] Khurana, M. and Singh, H. [2018], 'Data computation and secure encryption based on gyrator transform using singular value decomposition and randomization', *Procedia Computer Science* **132**, 1636–1645.

[87] King, P., Naughton, P., DeMoney, M., Kanerva, J., Walrath, K. and Hommel, S. [1999], 'Java code conventions', *Sun Microsystems Inc* **47**.

[88] Kölling, M. [2008], Using BlueJ to introduce programming, *in* 'Reflections on the Teaching of Programming', Springer, pp. 98–115.

[89] Kölling, M. [2015], 'Lessons from the design of three educational programming environments: Blue, BlueJ and Greenfoot', *International Journal of People-Oriented Programming (IJPOP)* **4**(1), 5–32.

[90] Kölling, M. and Barnes, D. J. [2017], 'Objects first with java: A practical introduction using BlueJ'.

[91] Kölling, M., Quig, B., Patterson, A. and Rosenberg, J. [2003], 'The BlueJ system and its pedagogy', *Computer Science Education* **13**(4), 249–268.

[92] Kontogiannis, K. A., DeMori, R., Merlo, E., Galler, M. and Bernstein, M. [1996], Pattern matching for clone and concept detection, *in* 'Reverse Engineering', Springer, pp. 77–108.

[93] Koschke, R. [2007], Survey of research on software clones, *in* 'Dagstuhl Seminar Proceedings', Schloss Dagstuhl-Leibniz-Zentrum für Informatik.

[94] Krejcie, R. V. and Morgan, D. W. [1970], 'Determining sample size for research activities', *Educational and Psychological Measurement* **30**(3), 607–610.

[95] Krsul, I. and Spafford, E. H. [1997], 'Authorship analysis: Identifying the author of a program', *Computers & Security* **16**(3), 233–257.

[96] Kuo, J.-Y., Cheng, H.-K. and Wang, P.-F. [2018], Program plagiarism detection with dynamic structure, *in* '2018 7th International Symposium on Next Generation Electronics (ISNE)', IEEE, pp. 1–3.

[97] Kustanto, C. and Liem, I. [2009], Automatic source code plagiarism detection, *in* 'Software Engineering, Artificial Intelligences, Networking and Parallel/Distributed Computing, 2009. SNPD'09. 10th ACIS International Conference on', IEEE, pp. 481–486.

[98] Lancaster, T. [2003], Effective and Efficient Plagiarism Detection, PhD thesis, South Bank University.

[99] Liu, C., Chen, C., Han, J. and Yu, P. S. [2006], GPLAG: detection of software plagiarism by program dependence graph analysis, *in* 'Proceedings of the 12th ACM SIGKDD international conference on Knowledge discovery and data mining', ACM, pp. 872–881.

[100] Lukashenko, R., Graudina, V. and Grundspenkis, J. [2007], Computer-based plagiarism detection methods and tools: an overview, *in* 'Proceedings of the 2007 International Conference on Computer Systems and Technologies', ACM, p. 40.

[101] Luquini, E. and Omar, N. [2011], Programming plagiarism as a social phenomenon, *in* 'Global Engineering Education Conference (EDUCON), 2011 IEEE', IEEE, pp. 895–902.

[102] Makuc, Ž. [2013], Methods to assist plagiarism detection, PhD thesis, Univerza v Ljubljani.

[103] Marshall, S. and Garry, M. [2005], How well do students really understand plagiarism?, *in* 'Proceedings of the 22nd annual conference of the Australasian Society for Computers in Learning in Tertiary Education (ASCILITE)', pp. 457–467.

[104] Martin, B. [1994], 'Plagiarism: a misplaced emphasis', *Journal of Information Ethics* **3**(2), 36–47.

[105] Maurer, H. A., Kappe, F. and Zaka, B. [2006], 'Plagiarism-a survey.', *J. UCS* **12**(8), 1050–1084.

[106] McConnell, S. [2004], *Code Complete*, Pearson Education.

[107] Meyer, C., Heeren, C., Shaffer, E. and Tedesco, J. [2011], Comoto: the collaboration modeling toolkit, *in* 'Proceedings of the 16th Annual SIGCSE Conference on Innovation and Technology in Computer Science Education, ITiCSE 2011, Darmstadt, Germany, June 27-29, 2011', ACM, pp. 143–147.

[108] Miaoulis, G. and Michener, R. D. [1976], *An Introduction to Sampling*, Kendall.

[109] Misic, M., Sustran, Z. and Protic, J. [2016], 'A comparison of software tools for plagiarism detection in programming assignments', *International Journal of Engineering Education* **32**(2), 738–748.

[110] NIST/SEMATECH [2013], ‘e-handbook of statistical methods’. (Accessed on 26/09/2017).

URL: *http://www.itl.nist.gov/div898/handbook/prc/section1/prc14.htm*

[111] Ohno, A. [2013], A methodology to teach exemplary coding style considering students, *in* ‘2013 IEEE Frontiers in Education Conference (FIE)’, IEEE, pp. 1908–1910.

[112] Ohno, A. and Murao, H. [2008], A quantification of students coding style utilizing hmmbased coding models for in-class source code plagiarism detection, *in* ‘Innovative Computing Information and Control, 2008. ICICIC’08. 3rd International Conference on’, IEEE, pp. 553–553.

[113] Ohno, A. and Murao, H. [2009], ‘A new similarity measure for in-class source code plagiarism detection’, *International Journal of Innovative Computing, Information and Control* **5**(11), 4237–4247.

[114] Ohno, A. and Murao, H. [2011], ‘A two-step in-class source code plagiarism detection method utilizing improved CM algorithm and SIM’, *International Journal of Innovative Computing, Information, and Control* **7**(8), 4729–4739.

[115] Okutan, A. [2018], ‘Use of source code similarity metrics in software defect prediction’, *arXiv preprint arXiv:1808.10033* .

[116] Oman, P. W. and Cook, C. R. [1988], ‘A paradigm for programming style research’, *ACM SIGPLAN Notices* **23**(12), 69–78.

[117] Oman, P. W. and Cook, C. R. [1990], A taxonomy for programming style, *in* ‘Proceedings of the 1990 ACM Annual Conference on Cooperation’, ACM, pp. 244–250.

[118] Parker, A. and Hamblen, J. O. [1989], 'Computer algorithms for plagiarism detection', *IEEE Transactions on Education* **32**(2), 94–99.

[119] Paul, S. and Prakash, A. [1994], 'A framework for source code search using program patterns', *IEEE Transactions on Software Engineering* **20**(6), 463–475.

[120] Poon, J. Y., Sugiyama, K., Tan, Y. F. and Kan, M.-Y. [2012], 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', ACM, pp. 122–127.

[121] Porter, M. [2006], A network analysis of committees in the United States House of Representatives, *in* 'APS Meeting Abstracts'.

[122] Porter, M. A., Mucha, P. J., Newman, M. and Friend, A. [2007], 'Community structure in the united states house of representatives', *Physica A: Statistical Mechanics and its Applications* **386**(1), 414 – 438.

[123] Power, L. G. [2009], 'University students' perceptions of plagiarism', *The Journal of Higher Education* **80**(6), 643–662.

[124] Pratama, M., Kemas, R. and Anisa, H. [2017], Digital News Graph Clustering using Chinese Whispers Algorithm, *in* 'Journal of Physics: Conference Series', Vol. 801, IOP Publishing, p. 012062.

[125] Prechelt, L., Malpohl, G. and Philippsen, M. [2002], 'Finding plagiarisms among a set of programs with JPlag', *J. UCS* **8**(11), 1016.

[126] Pyle, D. [1999], *Data Preparation for Data Mining*, Vol. 1, Morgan Kaufmann Publishers, Inc.

[127] Ranade, J. and Nash, A. [1993], *The Elements of C Programming Style*, McGraw-Hill, Inc.

[128] Rangel, F., González, F., Restrepo, F., Montes, M. and Rosso, P. [2016], Pan@fire: Overview of the pr-soco track on personality recognition in source code, *in* 'Forum for Information Retrieval Evaluation', Springer, pp. 1–19.

[129] Roy, C. K. and Cordy, J. R. [2007], 'A Survey on Software Clone Detection', *Queen's University, School of Computing* .

[130] Sallis, P. [1994], 'Contemporary computing methods for the authorship characterisation problem in computational linguistics', *New Zealand Journal of Computing* **5**(1), 85–95.

[131] Sallis, P., Aakjaer, A. and MacDonell, S. [1996], Software forensics: old methods for a new science, *in* 'Software Engineering: Education and Practice, 1996. Proceedings. International Conference', IEEE, pp. 481–485.

[132] Shan, S., Guo, F. and Ren, J. [2012], Similarity detection method based on assembly language and string matching, *in* 'Advances in Electronic Commerce, Web Application and Communication', Springer, pp. 363–367.

[133] Singh, H. [2018], 'Watermarking image encryption using deterministic phase mask and singular value decomposition in fractional mellin transform domain', *IET Image Processing* **12**(11), 1994–2001.

[134] Son, J.-W., Park, S.-B. and Park, S.-Y. [2006], Program plagiarism detection using parse tree kernels, *in* 'Pacific Rim International Conference on Artificial Intelligence', Springer, pp. 1000–1004.

[135] Song, H.-J., Park, S.-B. and Park, S. Y. [2015], 'Computation of program

source code similarity by composition of parse tree and call graph', *Mathematical Problems in Engineering* **2015**.

[136] Spafford, E. H. and Weeber, S. A. [1993], 'Software forensics: Can we track code to its authors?', *Computers & Security* **12**(6), 585–595.

[137] Stewart, G. W. [1993], 'On the early history of the singular value decomposition', *SIAM Review* **35**(4), 551–566.

[138] Tahaei, N. and Noelle, D. C. [2018], Automated plagiarism detection for computer programming exercises based on patterns of resubmission, *in* 'Proceedings of the 2018 ACM Conference on International Computing Education Research', ACM, pp. 178–186.

[139] Tassel, D. V. [1978], *Program Style, Design, Efficiency, Debugging and Testing*, Prentice Hall PTR.

[140] Tauer, J. and Ledgard, H. F. [1987], *C With Excellence: Programming Proverbs*, Sams Indianapolis, IN, USA.

[141] Tufano, M., Watson, C., Bavota, G., Di Penta, M., White, M. and Poshyvanyk, D. [2018], Deep learning similarities from different representations of source code, *in* 'International Conference on Mining Software Repositories'.

[142] Vaish, A. and Kumar, M. [2018], 'Color image encryption using singular value decomposition in discrete cosine stockwell transform domain', *Optica Applicata* **48**(1).

[143] Van Haaster, K. and Hagan, D. [2004], 'Teaching and learning with bluej: an evaluation of a pedagogical tool.', *Issues in Informing Science & Information Technology* **1**.

[144] Verco, K. L. and Wise, M. J. [1996*a*], 'Plagiarism à la mode: a comparison of automated systems for detecting suspected plagiarism', *The Computer Journal* **39**(9), 741–750.

[145] Verco, K. L. and Wise, M. J. [1996*b*], 'Software for detecting suspected plagiarism: Comparing structure and attribute-counting systems', *Proceedings of the 1st Australasian conference on Computer science education* **96**, 81–88.

[146] Walenstein, A., El-Ramly, M., Cordy, J. R., Evans, W. S., Mahdavi, K., Pizka, M., Ramalingam, G. and von Gudenberg, J. W. [2007], Similarity in programs, *in* 'Dagstuhl Seminar Proceedings', Schloss Dagstuhl-Leibniz-Zentrum für Informatik.

[147] Whale, G. [1990], 'Identification of program similarity in large populations', *The Computer Journal* **33**(2), 140–146.

[148] Wise, M. J. [1993], 'String similarity via greedy string tiling and running karp-rabin matching', *Online Preprint, Dec* **119**.

[149] Yamamoto, T., Matsushita, M., Kamiya, T. and Inoue, K. [2005], Measuring similarity of large software systems based on source code correspondence, *in* F. Bomarius and S. Komi-Sirviö, eds, 'Product Focused Software Process Improvement', Springer Berlin Heidelberg, Berlin, Heidelberg, pp. 530–544.

[150] Zimerman, M. [2012], 'Plagiarism and international students in academic libraries', *New Library World* **113**(5/6), 290–299.

[151] Zini, M., Fabbri, M., Moneglia, M. and Panunzi, A. [2006], Plagiarism detection through multilevel text comparison, *in* 'Automated Production of Cross Media Content for Multi-Channel Distribution, 2006. AXMEDIS'06. Second International Conference on', IEEE, pp. 181–185.