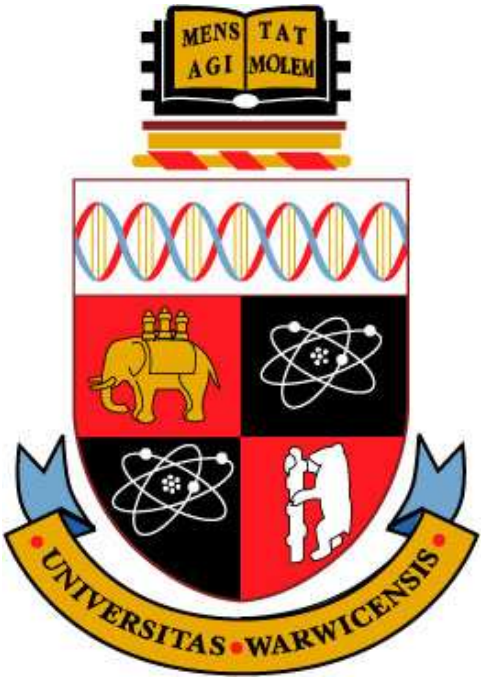

An Approach to Source-Code Plagiarism Detection and Investigation Using Latent Semantic Analysis

by

Georgina Cosma

A thesis submitted in partial fulfilment of the requirements for the degree of

Doctor of Philosophy in Computer Science

University of Warwick, Department of Computer Science

July 2008

Contents

List of Figures

List of Tables

Acknowledgements

I would like to express my deep and sincere gratitude to my supervisor, Dr Mike Joy, for his expert guidance, constructive criticism and continuous support and encouragement throughout my study. It has been a privilege to work with him.

I am deeply grateful to my advisor, Dr Steve Russ for his detailed and constructive comments, and for his important support throughout this work.

Special thanks to Professor Erkki Sutinen for accepting my request to be the external examiner.

Thank you also to past and present members of the Intelligent and Adaptive System Group, particularly Jane Yau, Amirah Ismail, Shanghua Sun, Jenny Liu, Shanshan Yang for their support and friendship.

My special thanks go to my good friend Jane Yau who I will miss dearly and has been such a wonderful and supportive friend. I really hope that we can keep in touch.

I am deeply grateful to my parents for their continuous support and encouragement throughout my studies. Last but not least, I would like to thank my husband for his unconditional support, advice and logical way of thinking that have been of great value for me. I cherish his care, love, understanding and patience. Also, one of the happiest experiences during the period of my PhD has been the birth of our lovely daughter 'Eirene' who has been a source of constant joy for my husband and me.

Declaration

The contents of this thesis are a result of my own work, and it contains nothing that is based on collaborative research. No part of the work contained in this thesis has been submitted for any degree or qualification at any other university. Parts of this thesis were published previously. Chapter 3 describes the main results from findings published in a detailed research report [26]. Findings discussed in Chapter 3 are also published in a conference proceeding [27], and a journal publication [28]. Parts of Chapters 2, 8, 9 and 10 are included in a paper submitted to a journal publication [29] and it is currently under review.

Abstract

This thesis looks at three aspects of source-code plagiarism. The first aspect of the thesis is concerned with creating a definition of source-code plagiarism; the second aspect is concerned with describing the findings gathered from investigating the Latent Semantic Analysis information retrieval algorithm for source-code similarity detection; and the final aspect of the thesis is concerned with the proposal and evaluation of a new algorithm that combines Latent Semantic Analysis with plagiarism detection tools.

A recent review of the literature revealed that there is no commonly agreed definition of what constitutes source-code plagiarism in the context of student assignments. This thesis first analyses the findings from a survey carried out to gather an insight into the perspectives of UK Higher Education academics who teach programming on computing courses. Based on the survey findings, a detailed definition of source-code plagiarism is proposed.

Secondly, the thesis investigates the application of an information retrieval technique, Latent Semantic Analysis, to derive semantic information from source-code files. Various parameters drive the effectiveness of Latent Semantic Analysis. The performance of Latent Semantic Analysis using various parameter settings and its effectiveness in retrieving similar source-code files when optimising those parameters are evaluated.

Finally, an algorithm for combining Latent Semantic Analysis with plagiarism detection tools is proposed and a tool is created and evaluated. The proposed tool, PlaGate, is a hybrid model that allows for the integration of Latent Semantic Analysis with plagiarism detection tools in order to enhance plagiarism detection. In addition, PlaGate has a facility for investigating the importance of source-code fragments with regards to their contribution towards proving plagiarism. PlaGate provides graphical output that indicates the clusters of suspicious files and source-code fragments.

Chapter 1

Introduction

This thesis investigates the application of an information retrieval technique, Latent Semantic Analysis (LSA), that derives semantic information from source-code files using information retrieval methods. Such information is useful during the detection of similar source-code files with regards to plagiarism detection. The performance of LSA is dependent on parameters that can influence its performance and this research aims to investigate its performance using various parameter settings. The effectiveness of LSA in similar source-code file detection when those parameters are optimised will be evaluated. We propose a technique for combining information retrieval techniques with plagiarism detection tools in order to improve the plagiarism detection and investigation process. We also propose a definition of what constitutes source-code plagiarism from the academic perspective.

1.1 Problem Description and Motivation

The task of detecting similar source-code files in programming assignments is carried out by most academics teaching programming. A review of the current literature on source-code plagiarism in student assignments revealed that there is no commonly agreed definition of what constitutes source-code plagiarism from the perspective of academics who teach programming on computing courses. Therefore, part of the research is concerned with creating a definition of what constitutes source-code plagiarism.

This thesis is also concerned with source-code similarity detection and investigation with a view to detecting plagiarism. Information retrieval methods have been applied to source-code plagiarism detection, for example Moussiades and Vakali have developed a plagiarism detection system based on the standard vector based approach and a new clustering algorithm [99].

A recent article by Mozgovoy provides a review on various source-code plagiarism detection approaches [100]. Most popular plagiarism detection algorithms use string-matching to create token string representations of programs. The tokens of each document are compared on a pair-wise basis to determine similar source-code segments between the files and compute the similarity value between files based on the similar segments found. Some well known recent structure metric systems include YAP3 [144], Plague [139], and JPlag [115]. These approaches focus on detecting plagiarism based on the source-code files structural information derived from the programming language syntax. Each file's words and characters

are converted into tokens representing their semantic meaning. The comparison process is not itself based on the semantic meaning of the file but it is rather a structural comparison, which consists of searching for files that contain matching token sequences. Algorithms that rely on detecting similar files by analysing their structural characteristics often fail to detect similar files that contain significant code shuffling, i.e. this kind of attack causes *local confusion* [115]. In addition, string-matching based systems convert source-code files into tokens using a parser. String-matching systems are language-dependent depending on the programming languages supported by their parsers, and are immune to many attacks, but as mentioned above they can be tricked by specific attacks mainly on the structure of the source-code.

With this in mind we are proposing a more flexible approach in detecting similar source-code files. Specifically, we are proposing the use of an information retrieval method, Latent Semantic Analysis (LSA) [40], to extract semantic information from source-code files and detect files that are relatively more similar than other files in the corpus.

The similarity computation algorithms of LSA and recent plagiarism detection tools (e.g. as described in Section 2.5) are different. One important feature of LSA is that it considers the relative similarity between files, i.e two files are considered similar by LSA if they are relatively more similar than other files in the corpus, whereas, recent plagiarism detection tools compute the similarity between files on a strict pair-wise basis.

We believe that LSA can help enhance plagiarism detection when integrated with current plagiarism detection tools. We hypothesise that LSA can be a very useful additional

tool that can detect suspicious source-code files that were missed by other tools and therefore reduce the number of plagiarism cases not detected. In addition LSA can help with the task of investigating source-code similarity by observing the relative similarity between source-code fragments and files.

Furthermore, the novelty of this approach is that we propose a tool for integrating LSA with plagiarism detection tools with the aim of enhancing the source-code similarity detection and investigation process with a view to detecting source-code plagiarism.

1.2 Research Hypotheses and Questions

The main questions that this research attempts to answer are:

1. What constitutes source-code plagiarism from the perspective of academics? (Chapter 3)

2. How effective is LSA for the task of similar source-code file detection when parameters are optimised? (Chapters 6 and 7)

   (a) Which parameters drive the effectiveness of LSA? (Chapter 5)

   (b) What main source-code specific pre-processing parameters exist and how do these influence the effectiveness of LSA? (Chapters 6 and 7)

3. How can LSA be integrated with plagiarism detection tools to improve plagiarism detection performance? (Chapter 8)

4. How effective is LSA when integrated with plagiarism detection tools? (Chapter 9)

5. How effective is LSA for the task of similarity investigation and visualisation of relative similarity? (Chapter 10)

The hypotheses of this research are:

- Hypothesis 1: Perspectives of academics on what constitutes source-code plagiarism vary. Conducting a survey will provide an insight into the perspective of academics on this issue. The outcome of the survey will enable for a detailed definition to be created.

- Hypothesis 2: Parameters can influence the effectiveness of the performance of LSA with regards to the detection of similar files. Optimising these parameters can improve the effectiveness of LSA with regards to detecting similar source-code files with a view on plagiarism detection.

- Hypothesis 3: LSA can be applied successfully as a technique to extract the underlying semantic information from source-code files and retrieve files that are relatively more similar than others. Using LSA the detection process becomes more flexible than existing approaches which are programming language dependent. Furthermore, integrating LSA with plagiarism detection tools can increase the number of plagiarism cases detected by retrieving similar file pairs that were missed by other tools.

- Hypothesis 4: Source-code files that have been plagiarised contain distinct source-code fragments that can characterise these files from the rest of the files in the corpus.

We hypothesise that the similarity between the plagiarised source-code fragments and the files in which they appear will be relatively higher than the files that do not contain the particular source-code fragment. In addition we hypothesise that using LSA, source-code fragments can be compared to source-code files to determine their degree of relative similarity to files.

1.3 Research Contributions

The research proposes a novel and detailed definition of what constitutes source-code plagiarism from the academic perspective. This is the first definition that attempts to define source-code plagiarism based on academics' perceptions. Grey areas regarding what constitutes source-code plagiarism are also discussed.

We propose and evaluate a new use of LSA to support the plagiarism detection process. With this approach and results, we propose a new tool to combine structural and semantic information retrieval techniques to improve plagiarism detection.

Finally, we propose and evaluate a new approach for investigating the similarity between source-code files with regards to gathering evidence of plagiarism. Graphical evidence is presented that allows for the investigation of source-code fragments with regards to their contribution toward evidence of plagiarism. The graphical evidence indicates the relative importance of the given source-code fragments across files in a corpus. Source-code fragments are categorised based on their importance as evidence toward indicating

plagiarism. This is done by using Latent Semantic Analysis to detect how important they are within the specific files under investigation in relation to other files in the corpus.

1.4 Meanings of the Word Semantic

The term *semantic* is concerned with the study of meaning in an expression. It is described differently by different fields. In the field of programming language theory, the term is commonly used in the context of *semantics of programming languages*, in which *semantics* is concerned with giving a meaning to programs or functions using mathematical models. Programming language semantics fall into three categories:

1. *Operational semantics* describe the behaviour of a program, by means of how it may execute or operate, as sequences of computational steps. These sequences define the meaning of the program [114].

2. *Denotational semantics* is an approach for representing the functionality of programs (i.e. what programs do). This involves representing programs (or program constructs) as expressions by constructing mathematical objects (called denotations). Denotations represent the meanings of programs [123].

3. *Axiomatic semantics* define the meanings of program constructs by describing their logical axioms. The purpose of axiomatic semantics is for proving the correctness (i.e. input-output behaviour) of a program [129].

In this research we use the term *semantics* as commonly used within the field of linguistics [85]. In this context, semantics refers to the meaning of a program, or elements of the program such as a source-code file (e.g. a Java class), source-code fragment (e.g. a Java method), or meaning of terms. We are concerned with two types of semantics: lexical semantics and structural semantics. *Lexical semantics* is about the meaning of words, and *structural semantics* describe the relationship between the meanings of words in chunks of source-code. Source-code programs must follow the syntax rules of a programming language that define how the keywords of the language can be arranged to make a sentence (e.g. if statement, for loop etc).

In a corpus of student assignments, source-code files may have a similar semantic meaning. This is inevitable since they are developed to solve a common programming problem and share common programming language keywords. However, in the context of plagiarism, lexical and structural similarity between two files are compared. Files are considered as suspicious if they share lexical and structural similarity that are relatively similar when compared to other files in the corpus.

It is the degree of semantic similarity (or relatedness) between files, determined by measuring the relative meaning between files, that defines whether two source-code files are semantically related. Semantic similarity would be similar/same variable names that have been renamed but have the same meaning in the context being used (i.e. synonyms). In the context of plagiarism detection, files that are more similar than others in the corpus are of interest, i.e. it is those files that are considered suspicious. Semantic similarity (or

relatedness) is often measured by the relative closeness of two files (or other artifacts e.g. methods, terms) in the semantic space using similarity measures.

1.5 Thesis Organisation

Chapter 2 discusses general issues surrounding plagiarism in academia, existing plagiarism definitions, and issues concerning plagiarism in computer science. A discussion on plagiarism detection tools and their algorithms is also presented.

Chapter 3 describes the main findings from a survey carried out to gain an insight into the perspectives of UK Higher Education academics on what is understood to constitute source-code plagiarism in an undergraduate context. A new definition of source-code plagiarism is described.

Chapter 4 provides a theoretical and practical introduction to the LSA information retrieval technique using a context specific example. The foundations for understanding the basics of LSA are described by using a step-by-step procedure for discussing the application of LSA to an example source-code corpus. This part of the thesis does not cover new research, but rather introduces the reader to basic concepts and terminology.

Chapter 5 describes relevant literature focusing on the parameters that influence the performance of LSA. Parameter settings that researchers have used and recommended for effective application of LSA to natural language information retrieval tasks are described. The main focus of this Chapter is on parameter choices made by researchers for tasks spe-

cific to source-code. Finally, plagiarism detection tools whose functionality is based on information retrieval algorithms are explained.

Chapter 6 presents an analysis of the influence of parameters on the effectiveness of source-code similarity detection with LSA. It describes experiments conducted with general LSA parameters and source-code specific pre-processing parameters.

Chapter 7 describes the findings gathered from conducting a small experiment to investigate the impact of source-code specific pre-processing parameters on the effectiveness of source-code similarity detection with LSA.

Chapter 8 describes similarity in source-code files and categories of source-code similarity. It introduces a new tool, PlaGate, which is designed to allow the integration of LSA with plagiarism detection tools for enhancing the similarity detection and investigation process.

Chapter 9 presents the results of experiments performed to evaluate the detection performance of PlaGate when applied alone, and when integrated with two tools, JPlag and Sherlock.

Chapter 10 presents the results gathered from evaluating the performance of the investigation component of the PlaGate tool. The evaluation is concerned with the comparison of PlaGate's results to human judgements.

Chapter 11 provides an overview of the main results and conclusions of this research. Open questions and future work are also discussed.

Chapter 2

Plagiarism in Programming Assignments

This Chapter discusses general issues surrounding plagiarism in academia, plagiarism definitions, and plagiarism in computer science. A discussion on plagiarism detection tools and their algorithms is also presented.

2.1 The Extent of Plagiarism in Academia

Plagiarism in programming assignments is a growing concern in academia. Source-code can be obtained in various ways including the Internet, source-code banks, and text books. A recent survey was conducted by Nadelson [105] on issues surrounding academic misconduct in a large University. Nadelson gathered the perceptions of 72 academics on known

plagiarism incidents, and found that those academics suspected 570 incidents of misconduct by undergraduate and graduate students. The majority of incidents were 'accidental/unintentional plagiarism' with 134 of those incidents involving undergraduate students and 39 involving graduate students. Furthermore, a large number of incidents were reported when academics suspected that students had submitted papers copied from the Internet. Many incidents concerning 'purposeful plagiarism' were also reported. Other forms of academic misconduct reported were 'class test cheating' and 'take home test cheating'. On suspected cases of academic misconduct it was found that academics prefer to deal with such incidents informally without bringing issues out of the classroom due to discomfort with the University's formal procedures. This discomfort was mainly due to two issues – academics felt that there was not enough evidence to report these incidents, and that proceeding formally with cases of academic misconduct would reflect negatively on their performance as a academics [105]. Furthermore, Flint *et al.* [42] also identified the fact that academics often have their own personalised perception of plagiarism definitions and policies which may not be consistent with their University's plagiarism policy. In addition, Keith-Spiegel *et al.* [73] found that academics did not often pursue the University's policy on dealing with plagiarism due to concerns about confronting students and not feeling protected by University procedures, and also found that academics felt sympathy for the impact that formal procedures would have on students.

It is important that academics take action in order to improve student behaviour. They need to promote ethical behaviour mainly by taking action when they suspect misconduct

[105]. According to Hannabuss [51] plagiarism is a difficult matter because,

> "evidence is not always factual, because plagiarism has a subjective dimension (i.e. what is a lot?), because defendants can argue that they have independently arrived at an idea or text, because intention to deceive is very hard to prove."

Furthermore, suspected cases of plagiarism where the original text cannot be found are the most difficult to prove, due to lack of evidence [86, 62]. In addition, although academics may suspect plagiarism, searching for the original material and finding and collating enough evidence to convince the relevant academic panel in charge of dealing with plagiarism cases can be time demanding [86].

Studies have shown that the prevalence of plagiarism is on the increase [86]. On-line resources exist where students can hire expert coders to implement their programming assignments [57]. These opportunities make plagiarism easier for students. Concerns about the ease with which students can obtain material from on-line sources and use the material in their student work have been expressed in a number of studies [69, 121].

Dick, *et al.* and Sheard, *et al.* identify various student cheating techniques and reasons why students cheat [35, 125]. In addition, they define cheating in the form of questions that, if answered positively, could indicate cheating. Culwin, *et al.* conducted a study of source-code plagiarism in which they obtained data from 55 United Kingdom (UK) Higher Education (HE) computing schools [30]. They found that 50% of the 293 academics who participated in their survey believed that in recent years plagiarism has increased. Asked to

estimate the proportion of students undertaking source-code plagiarism in initial programming courses, 22 out of 49 staff responded with estimates ranging from 20% to more than 50%.

Studies have been carried out on the impact of plagiarism on learning and teaching [34] and suggest ways of preventing plagiarism in academia. Carroll and Appleton have devised a good practice guide suggesting techniques for dealing with plagiarism [22].

2.2 Definitions of Plagiarism

Plagiarism in academic institutions is often expressed as copying someone else's work (i.e., another students or from sources such as books), and failing to provide appropriate acknowledgment of the source (i.e., the originator of the materials reproduced). The act of plagiarism is still regarded as an offence regardless whether it was intentional or unintentional.

Hannabuss [51] defined plagiarism as "the unauthorised use or close imitation of the ideas and language/expression of someone else". In the context of academic work, plagiarism can range from the citation of a few sentences without acknowledging the original author to copying an entire document. Plagiarism is an academic offence and not a legal offence, and exists as institutional rules and regulations [104, 86]. Therefore, what constitutes plagiarism is perceived differently across institutions based on their rules and regulations. All Universities regard plagiarism as a form of cheating or academic misconduct, but their

rules and regulations for dealing with suspected cases of plagiarism vary, and the penalties imposed on cheating depend on factors such as the severity of the offence and whether the student admits to the offence. These penalties vary amongst institutions, and include giving a zero mark for the plagiarised work, resubmission of the work, and in serious cases of plagiarism the penalty can be expulsion from the University.

Interestingly, Flint *et al.* conducted a survey of 26 academics to gain their perceptions of student plagiarism. The academics that participated in their survey were employed at different departments and schools at a single University. They found that staff have their own "personalised definitions" on plagiarism. The staff perceptions were influenced by previous experiences they had with plagiarism issues in higher education, and it was also found that academic perceptions were not consistent with their corresponding institution's policy definition on plagiarism. Such inconsistencies in plagiarism definitions across academics employed at a single institution suggests inconsistencies in following the University policy when plagiarism is detected [42, 97]. This can lead to inconsistent policy application amongst academics in the same department/institution and can also cause different treatment/punishment of suspected plagiarism cases. This inconsistency in academic perception on plagiarism can inevitably cause confusion amongst students on what is and what is not plagiarism [42].

Plagiarism can take many forms, including the following [97].

- *Word-by-word copying*, which involves directly copying sentences or chunks of sentences from other peoples work without providing quotations and/or without appro-

priately acknowledging the original author.

- *Paraphrasing*, which involves closely rewriting (i.e. only changing some of the words but not making enough changes) text written by another author and appropriately citing the original author.

- *Plagiarism of secondary sources*, which involves referencing or quoting original sources of text taken from a secondary source without obtaining and looking up the original source.

- *Plagiarism of the form of a source*, is when the structure of an argument in a source is copied without providing acknowledgments that the 'systematic dependence on the citations' was taken from a secondary source. This involves looking up references and following the same structure of the secondary source.

- *Plagiarism of ideas*, which involves using ideas originally expressed in a source text without 'any dependence on the words or form of the source'.

- *Blunt plagiarism or authorship plagiarism* which is taking someone else's work and putting another's name to it.

Much work has been done on the perceptions of students, and why students plagiarise. Marshall and Garry [96] conducted a survey to gather the perceptions of 181 students with regards to what the students understand as plagiarism. The students were presented with small scenarios that described issues on copyright and plagiarism. They were required

to rate the seriousness of the described behaviour in the scenarios; how the described behaviours would be regarded by other students, the University and the general public; and were also asked to indicate whether they had previously engaged in a behaviour similar to that described in the given scenarios. They found that students generally have a poor understanding of what is plagiarism and the actions that constitute plagiarism. Their survey results revealed that the vast majority of students (i.e. 94%) identified scenarios describing obvious plagiarism such as "copying the words from another source without appropriate reference or acknowledgment", however the responses among students were inconsistent regarding scenarios on how to correctly use materials from other sources. This involved scenarios on plagiarism of secondary sources, plagiarism of the form of a source and paraphrasing, where 27%, 58%, and 62% of students correctly identified this as plagiarism respectively [96].

Many factors can motivate students to plagiarise. Factors such as inadequate time management, workload, laziness, not understanding what constitutes plagiarism, fear of failure, high achievement expectations, cryptomnesia, thrill of breaking the rules and risk of getting caught, work ethics, competitive achievement, low self esteem, time pressure, and to increase their mark [9, 98, 17]. A more detailed discussion on why students cheat is also given by Bennett [11].

Bennett [11] conducted a detailed literature review on factors motivating students to plagiarise and classified these factors into five categories:

1. *Means and opportunity*: the fact that resources are readily available and easily ac-

cessible by students over the Internet makes it convenient for students to gain instant and easy access to large amounts of information from many sources. Furthermore, many Internet sites exist that provide essays to students, and also many of these sites provide chargeable services to students for writing their essays and papers.

2. *Personal traits*: factors such as the desire to gain high marks, fear of failure, pressure from family members, desire to achieve future job prospects and career development, and competitive behaviour amongst students can cause them to plagiarise.

3. *Individual circumstances*: students who work to finance their studies have less time to study and time pressures can cause them to cheat.

4. *Academic integration*: students that do not integrate or 'fit in' with University life, and are therefore alienated from their courses or instructors, are more likely to cheat than students who are better integrated in University life.

5. *Other factors*: these include the student's previous experiences of plagiarism prior to attending higher education, their education experience of developing good writing skills, their nationality (i.e. foreign students with language difficulties are more likely to cheat than home students fluent in the language). Other factors include age and gender [17, 96].

Dick *et al.* [35] categorised the types of cheating behaviour related to plagiarism offences into: copying, exams, collaboration, and deception. *Copying* involves taking material from other sources, for example, students' sharing their work, stealing another's work,

and taking source-code fragments from various sources and collating them together to create a single program. Types of *cheating in exams* include using materials such as cheat sheets, lecture notes, books in a closed book exam, talking in a foreign language during an exam, looking at another student's exam paper, and stealing an exam paper from an academic's office. Cheating by inappropriate *collaboration* involves students collaborating inappropriately for a particular assessment. Such types of inappropriate collaboration include students splitting the work between them and combining the solutions to create a single solution and all submitting the work as their own, working together to solve programming problems which were meant to be the work of an individual student, and gaining assistance with assessments from out of campus professional services. *Deception* involves students not being truthful about their circumstances in an attempt to unfairly gain an advantage, for example, an assessment submission date extension.

2.3 Plagiarism in Computer Science

Plagiarism in source-code files occurs when source-code is copied and edited without proper acknowledgment of the original author [30, 62]. According to Joy and Luck [62], lexical changes are those changes that can be done to the source-code without affecting the parsing of the program, and structural changes are those changes made to the source code that will affect the parsing of the code and involve program debugging. Examples of lexical changes include changing comments and identifier names; and examples of structural changes involve reordering and replacing statements.

Decoo's book on academic misconduct discusses various issues surrounding academic plagiarism, and briefly discusses software plagiarism at the levels of user-interface, content and source-code [31]. In order to identify cases of plagiarism, one must have a clear idea of what actions constitute plagiarism. Sutherland-Smith completed a survey to gather the perspectives of teachers in the faculty of Business and Law at South-Coast University in Australia [133]. Their findings revealed varied perceptions on plagiarism among academics teaching the same subject, and the author suggests that a "collaborative, cross-disciplinary re-thinking of plagiarism is needed." In addition, a review of the current literature on source-code plagiarism reveals a lack of research on the issue of what is considered source-code plagiarism from the perspective of academics who teach programming on computing courses.

Many definitions of plagiarism exist and these are mostly found in dictionaries. One such definition is one obtained from an online dictionary [113],

> "the unauthorized use or close imitation of the language and thoughts of another author and the representation of them as one's own original work."

Higher and further academic institutions often have their own policies and definitions as to what constitutes plagiarism. Plagiarism is considered to be a serious academic offence. Furthermore, some University policies also consider it as plagiarism if students submit material or part of material that they have previously produced and submitted for another assessment in which academic credit was gained. This type of academic offence is referred

to as *self-plagiarism*.

A gap in the literature appears on detailed definitions of source-code plagiarism in academia. The definitions that exist in literature were created by academics who have developed source-code plagiarism detection tools and in their publications attempt to define plagiarism. Their definitions are often accompanied by a list of plagiarism attacks.

Joy and Luck [62] define plagiarism as "unacknowledged copying of documents or programs". They identify reasons why students often cheat in programming assignments. The reasons they provide are centered around students programming abilities. They identify *weak students* that collaborate with their fellow students to produce an assignment that requires students to work on their own; weak students who copy and edit another students work without their permission; and *poorly motivated* (but not weak) students who copy and modify another student's work in order to avoid doing the work themselves.

Manber [93] considers two files to be similar,

> "if they contain a significant number of substrings that are not too small ... Similar files may be different versions of the same program, different programs containing a similar procedure, different drafts of an article."

Based on this, Manber developed a tool, Sif, with the aim of detecting similar files with similarity levels as little as 25%. Although their system was not built with the aim of detecting plagiarism, the author proposes that the tool could be used for this task. This description however, is not oriented to student assignments.

According to Jones [59],

> "A plagiarized program is either an exact copy of the original, or a variant obtained by applying various textual transformations".

Jones [59] provides the following list of transformations:

- Verbatim copying,
- Changing comments,
- Changing white space and formatting,
- Renaming identifiers,
- Reordering code blocks,
- Reordering statements within code blocks,
- Changing the order of operands/operators in expressions,
- Changing data types,
- Adding redundant statements or variables, and
- Replacing control structures with equivalent structures.

All the descriptions we have mentioned so far appear to be short and rather incomplete. The above definitions confuse similarity and plagiarism. It must be kept in mind that even

if similarity is found between files, plagiarism can only be determined by investigating the similarity carefully to determine whether it was *suspicious* (as defined below) or not-suspicious. For example, not suspicious similarity between source-code files may exist due to source-code examples and solutions the students were given during classes [26]. Suspicious similarity between source-code files involves source-code fragments that are distinct in program logic, approach and functionality, and this is the kind of similarity that could be used as strong evidence for proving plagiarism [26].

Similarity between two files is not based on the entire files, but rather on the source-code fragments they contain. Investigation involves scrutinizing the matching source-code fragments and judging whether the similarity between them appears to be suspicious or not suspicious. More detail on source-code similarity is described in Section 8.2. For now, we define the conditions under which two or more files may be considered to be *suspicious*:

Definition: *If a significant amount of similarity is found then the files under investigation can be considered similar or suspicious. Suspicious files share similar source-code fragments which characterise them as distinct from the rest of the files in the corpus. The majority of similar source-code fragments found in suspicious files appear in a similar form only in the files under investigation, and share distinct and important program functionality or algorithmic complexity.*

A detailed discussion of what constitutes source-code plagiarism is given in Chapter 3.

2.4 Transition from Source-Code Similarity to Source-Code Plagiarism

Source-code plagiarism detection in programming assignments is a task many higher education academics carry out. Source-code plagiarism in programming assignments occurs when students reuse source-code authored by someone else, either intentionally or unintentionally, and fail to adequately acknowledge the fact that the particular source-code is not their own [28]. Once similarity between students work is detected, the academic proceeds with the task of investigating this similarity. The investigation process involves comparing the detected source-code files for plagiarism by examining their similar source-code fragments.

It is important that instances of plagiarism are detected and gathering sound evidence to confidently proceed with plagiarism is a vital procedure. Joy and Luck [62] identify the issue of the burden of proof on gathering appropriate evidence for proving plagiarism,

> "Not only do we need to detect instances of plagiarism, we must also be able to demonstrate beyond reasonable doubt that those instances are not chance similarities."

Other academics have raised issues related to plagiarism evidence [105, 51]. Figure 2.1 illustrates the process of detecting similarity and reaching a decision.

The detection stage often involves using computer-aided detection tools, often referred

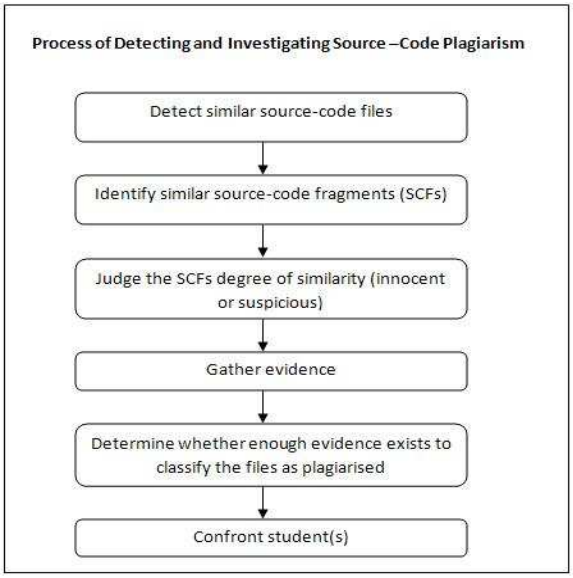

Figure 2.1: Similarity detection and investigation

to as plagiarism detection tools. Although these tools are called plagiarism detection tools, they detect similarities between students source-code programs, but it is up to the academic to judge whether plagiarism is the reason behind the similarity found in the detected programs. As Figure 2.1 illustrates, once similarity is detected, the academic goes through the file, identifies the matching source-code fragments. The next step is to determine whether the detected similarity between these is innocent or suspicious (as described in Section 8.3). Once this is done, evidence is gathered and if enough evidence exists then the files in question can be considered suspicious. However, before a final decision is reached, a typical process would be that the students (or student in the case of copying from sources such as books) involved are confronted with the evidence and finally a decision as to whether plagiarism was the case, is reached.

2.5 Source-Code Plagiarism Detection Tools

Many different plagiarism detection tools exist and these can be categorised depending on their algorithms. This Section describes source-code plagiarism detection tools using the categories identified by Mozgovoy [100]. These include *fingerprint based systems*, and *content comparison techniques*. Various other classifications exist in the literature [79, 137, 138].

2.5.1 Fingerprint based systems

Tools based on the fingerprint approach create *fingerprints* for each file, which contain statistical information about the file, such as average number of terms per line, number of unique terms, and number of keywords. Files are considered similar if their fingerprints are close to each other. This closeness is usually determined by measuring the distance (using a distance function in a mathematical sense) between them [100].

In earlier years, fingerprints were employed in *attribute counting* systems. The first known plagiarism detection system was an attribute counting program developed by Ottenstein for detecting identical and nearly-identical student work [109, 108]. The program used Halstead's software metrics to detect plagiarism by counting operators and operands for ANSI-FORTRAN modules [49, 50]. The metrics suggested by Halstead were:

- number of unique operators
- number of unique operands

- total number of occurrences of operators
- total number of occurrences of operands

Robinson and Soffa developed a plagiarism detection program that combined new metrics with Halsteads metrics in order to improve plagiarism detection [118]. Their system, called ITPAD, consisted of three steps: lexical analysis, analysis of program structure for characteristics, and analysis of the characteristics. ITPAD (Institutional Tool for Program ADvising) breaks each program into blocks and builds a graph to represent the program structure. It then generates a list of attributes based on the lexical and structural analysis and compares pairs of programs by counting these characteristics.

Rambally and Sage [116] created an attribute counting system, which accepts student's programs, parses them and then creates a *knowledge system* which contains knowledge vectors, where each vector holds information about the attributes in a student's program. Rambally and Sage use similar attributes identified by previously discussed systems, however, they take a different approach which counts the loop-building attributes within a program. Instead of counting variants of loop-building statements separately, they do not discriminate between different types of loops, they include the count of occurrences of all these statements into one attribute count. Once the knowledge vectors are created the programs are classified in a *decision tree*. Based on the decision tree, the programs which contain similarities are identified.

Since then, more advanced plagiarism detection tools have been developed. The lit-

erature often refers to these tools as *structure metric systems*. Structure metric systems compare structure of programs for similarity. Mozgovoy [100] has classified these as *content comparison techniques*, and these are discussed in Section 2.5.2.

Comparisons of attribute-counting and string-matching based systems have shown that attribute counting methods alone were not adequate enough for detecting plagiarism [139, 137, 138]. More recent plagiarism detection tools such as MOSS [2] combine the fingerprinting approach with the structure metric approach.

MOSS (Measure of Software Similarity) [2, 122] is based on a string-matching algorithm that functions by dividing programs into k-grams, where a k-gram is a contiguous substring of length k. Each k-gram is hashed and MOSS selects a subset of these hash values as the program's fingerprints. Similarity is determined by the number of fingerprints shared by the programs, i.e., the more fingerprints they share, the more similar they are. For each pair of source-code fragments detected, the results summary includes the number of tokens matched, the number of lines matched, and the percentage of source-code overlap between the detected file pairs [122].

2.5.2 Content comparison techniques

Content Comparison techniques are often referred to as *structure-metric* systems in the literature. These systems convert programs into tokens and then search for matching contiguous sequence of substrings within the programs. Similarity between programs depends on the percentage of the text matched. Mozgovoy [100] classified content comparison techniques

into *string matching-based algorithms*, *parameterised matching algorithms* and parse trees comparison algorithms.

String-matching algorithms

Most recent plagiarism detection systems rely on comparing the structure of programs. These systems use attribute-counting metrics but they also compare the program structure in order to improve plagiarism detection. String-matching based systems employ comparison algorithms that are more complex than attribute counting algorithms. Most string-matching algorithms work by converting the programs into tokens and then use a sophisticated searching algorithm to search for common substrings of text within two programs.

Such systems were initially proposed by Donaldson *et al.* [37] who identified simple techniques that novice programming students use to disguise plagiarism. These techniques are:

- renaming variables,
- reordering statements that will not affect the program result,
- changing format statements, and
- breaking up statements, such as multiple declarations and output statements.

The techniques identified by Donaldson *et al.* [37] are of the most basic forms of attack. Whale [139], and Joy and Luck [62] also provide a more detailed list of attacks. A much

more detailed list is also provided by Prechelt *et al.* [115]. We also provide an exhaustive list containing plagiarism attacks, found in [26].

The program developed by Donaldson *et al.* [37] scans source-code files and stores information about certain types of statements. Then, statement types significant in describing the structure are assigned a single code character. Each assignment is then represented as a string of characters. If the string representations are identical or similar, then the pair of programs is returned as similar.

Some recent string-matching based systems include Plague [139], YAP3 (Yet Another Plague) [144], JPlag [115], and Sherlock [62].

In most string-matching based systems, including the ones mentioned above, the first stage is called tokenisation. At the tokenisation stage each source-code file is replaced by predefined and consistent tokens, for example different types of loops in the source-code may be replaced by the same token name regardless of their loop type (e.g. while loop, for loop). Each tokenised source-code file is thus represented as a series of token strings. The programs are then compared by searching for matching substring sequences of tokens. Programs that contain matching number of tokens above a given threshold are returned as similar. Similarity between two files is most commonly computed by the coverage of matched tokens between the detected files.

Plague [139] first creates a sequence of tokens for each file, and compares the tokenised versions of selected programs using a string matching technique. Results are displayed as

a list of matched pairs ranked in order of similarity, measured by the length of the matched portioned of the token sequences between the two files. Plague is known to have a few problems [25]. One of the problems with Plague is that it is time-consuming to convert it to another programming language. Furthermore, the results are displayed in two lists ordered by indices which makes interpretation of results not immediately obvious. Finally, Plague is not very efficient because it relies upon a number of additional Unix tools which causes portability issues.

YAP3 converts programs into a string of tokens and compares them by using the token matching algorithm, *Running-Karp-Rabin Greedy-String-Tiling algorithm (RKR-GST)*, in order to find similar source-code segments [144]. YAP3 pre-processes the source-code files prior to converting them into tokens. Pre-processing involves removing comments, translating upper case letters to lower case, mapping synonyms to a common form (i.e., function is mapped to procedure), reordering the functions into their calling order, and removing all tokens that are not from the lexicon of the target language (i.e., removing all terms that are not language reserved terms). YAP3 was developed mainly to detect breaking of code functions into multiple functions, and to detect the reordering of independent source-code segments. The algorithm works by comparing two strings (the pattern and the text) which involves searching the text to find matching substrings of the pattern. Matches of substrings are called tiles. Each tile is a match which contains a substring from the pattern and a substring from the text. Once a match is found the status of the tokens within the tile are set to marked. Tiles whose length is below a minimum-match length threshold are ignored. The

RKR-GST algorithm aims to find maximal matches of contiguous substring sequences that contain tokens that have not been covered by other substrings, and therefore to maximise the number of tokens covered by tiles.

JPlag [115] uses the same comparison algorithm as YAP3, but with optimised run time efficiency. In JPlag the similarity is calculated as the percentage of token strings covered. One of the problems of JPlag is that files must parse to be included in the comparison for plagiarism, and this causes similar files to be missed. Also, JPlag's user defined parameter of minimum-match length is set to a default number. Changing this number can alter the detection results (for better or worse) and to alter this number one may need an understanding of the algorithm behind JPlag (i.e RKR-GST). JPlag is implemented as a web service and contains a simple but efficient user interface. The user interface displays a list of similar file pairs and their degree of similarity, and a display for comparing the detected similar files by highlighting their matching blocks of source-code fragments.

Sherlock [62] also implements a similar algorithm to YAP3. Sherlock converts programs into tokens and searches for sequences of lines (called runs) that are common in two files. Similarly to the YAP3 algorithm Sherlock searches for runs of maximum length. Sherlock's user interface displays a list of similar file pairs and their degree of similarity, and indicates their matching blocks of source-code fragments found within detected file pairs. In addition, Sherlock displays quick visualisation of results in the form of a graph where each vertex represents a single source-code file and each edge shows the degree of similarity between the two files. The graph only displays similarity (i.e., edges) between files

above the given user defined threshold. One of the benefits of Sherlock is that, unlike JPlag, the files do not have to parse to be included in the comparison and there are no user defined parameters that can influence the system's performance. Sherlock, is an open-source tool and its token matching procedure is easily customisable to languages other than Java [62]. Sherlock is a stand-alone tool and not a web-based service like JPlag and MOSS. A stand alone tool may be more preferable to academics with regards to checking student files for plagiarism when taking into consideration confidentiality issues.

Plaggie [1] is a similar tool to JPlag, but employs the RKR-GST without any speed optimisation algorithms. Plaggie converts files into tokens and uses the RKR-GST algorithm to compare files for similarity. The idea behind Plaggie is to create a tool similar to JPlag but that is stand-alone (i.e. installed on a local machine) rather than a web-based tool, and that has the extra functionality of enabling the academic to exclude code from the comparison, i.e. code given to students in class.

Another plagiarism detection tool is that developed by Mozgovoy *et al.*, called Fast Plagiarism Detection System (FDPS) [102, 103]. FDPS aims to improve the speed of plagiarism detection by using an indexed data structure to store files. Initially files are converted into tokens and tokenised files are stored in an indexed data structure. This allows for fast searching of files, using an algorithm similar to that in YAP3. This involves taking substrings from a *test file* and searching for matching substring in the *collection file*. The matches are stored in a repository and thereafter used for computing the similarity between files. Similarity is the ratio of the total number of tokens matched in the collection file to the

total number of tokens in the test file. This ratio gives the coverage of matched tokens. File pairs with a similarity value above the given threshold are retrieved. One of the problems with the FDPS tool is that its results cannot be visualised by means of displaying similar segments of code [101]. Therefore, the authors combined FPDS with the Plaggie tool for conducting the file-file comparison and visualising the results.

Parameterized matching algorithms

Parameterized matching algorithms are similar to the ordinary token matching techniques (as discussed in Section 2.5.2) but with a more advanced tokenizer [100]. Basically, these parameterized matching algorithms begin by converting files into tokens. A parameterized match (also called p-match) matches source-code segments whose variable names have been substituted (renamed) systematically.

The Dup tool [6] is based on a p-match algorithm, and was developed to detect duplicate code in software. It detects identical and parameterized sections of source-code. Initially, using a lexical analyzer the tool scans the source-code file, and for each line of code, a transformed line is created. The source-code is transformed into parameters, which involves transforming identifiers and constants into the same symbol P, and a list of the parameter candidates. For example line $x = fun(y)+3*x$ is transformed into $P = P(P)+P*P$ and a list containing x, fun, y, 3 and x is generated. Then the lexical analyser creates an integer that represents each transformed line of code. Given a threshold length, Dup aims to find source-code p-matches between files. Similarity is determined by computing the percentage

of p-matches between two files. The Dup tool outputs the percentage of similarity between the two files, a profile showing the p-matched lines of code between two files, and a plot which displays the location of the matched code. According to [46], the Dup tool fails to detect files if false statements are inserted, or when a small block of statements is reordered. Other parameterized matching algorithms exist in the literature [3, 7, 56, 119, 44].

Parse Trees Comparison Algorithms

Parse tree comparison algorithms implement comparison techniques that compare the structure of files [100]. The Sim utility [46] represents each file as a parse tree, and then using string representations of parse trees it compares file pairs for similarity. The comparison process begins by converting each source-code file into a string of tokens where each token is replaced by a fixed set of tokens. After tokenisation, the token streams of the files are divided into sections and their sections are aligned. This technique allows shuffled code fragments to be detected.

Sim employs ordinary string matching algorithms to compare file pairs represented as parse trees. As with most string matching algorithms, Sim searches for maximal common subsequence of tokens. Sim outputs the degree of similarity (in the range of 0.0 and 1.0) between file pairs, and was developed to detect similarity in C programs. According to the authors of Sim, it can be easily modified to work with programs written in languages other than C. It can detect similar files that contain common modifications such as name changes, reordering of statements and functions, and adding/removing comments and white spaces

[46].

A more recent application of parse trees comparison algorithms is that implemented in the Brass system [10]. The functionality of Brass involves representing each program as a *structure chart*, which is a graphical design consisting of a tree representation of each file. The root of the tree specifies the file header, and the child nodes specify the statements and control structures found in the file. The tree is then converted into a binary tree and a symbol table (data dictionary) which holds information about the variables and data structures used in the files. Comparison in Brass is a three part process. The first two algorithms involve comparing the structure trees of files and the third involves comparing the data dictionary representations of the files. Initially, similar file pairs are detected using the first comparison algorithm. Thereafter, the second and third comparison algorithms are applied to the detected file pairs.

Tree comparison involves using algorithms more complex than string matching algorithms. From this it can be assumed that systems based in tree comparison algorithms are slower than systems based on string matching algorithms [100]. The filtering technique by Brass, speeds up the comparison process.

According to Mozgovoy [100] not much research has been carried out in the area of parse tree algorithms, and it is not known whether these algorithms perform better than string matching algorithms with regards to detecting similar source-code file pairs.

2.6 Conclusion

In this Chapter we have surveyed a wide variety of source-code plagiarism detection tools and source-code plagiarism definitions. Although, tools designed to detect similar files in student assignments are often referred to as *plagiarism detection tools*, they are essentially *similarity detection tools*. These tools are programmed to detect similar source-code files and it is up to the academic to investigate the similarities between the detected file pairs and decide whether they are suspicious of plagiarism.

The literature review presented in this Chapter suggests that a gap exists on what constitutes source-code plagiarism in academia. Most of the existing definitions are very brief and have been created by academics, who developed plagiarism detection tools and in their publications attempt to provide a definition of source-code plagiarism. We have conducted a survey to gather academics perceptions on what constitutes source-code plagiarism, and findings of the survey are described in Chapter 3.

Chapter 3

Towards a Definition of Source-Code Plagiarism

A recent review of the literature on source-code plagiarism in student assignments (refer to Chapter 1) revealed that there is no commonly agreed definition of what constitutes source-code plagiarism from the perspective of academics who teach programming on computing courses.

In this Chapter we discuss the findings from a survey carried out to gather an insight into the perspectives of UK Higher Education (HE) academics of what is understood to constitute source-code plagiarism in an undergraduate context and we use the survey findings to propose a definition of what constitutes source-code plagiarism.

3.1 Methodology

An on-line questionnaire was distributed to academics across UK HE institutions. The mailing list of academics was supplied by the Higher Education Academy Subject Centre for Information and Computing Sciences (HEA-ICS). The UK has approximately 110 HE level computing departments, for which the HEA-ICS provides support. The mailing list contained addresses for 120 academics, most of whom were assumed to have expertise in teaching programming.

The survey instructions specified that only academics who are currently teaching (or have previously taught) at least one programming subject should respond. The survey was completed anonymously, but a section in which the academics could optionally provide personal information was included.

A total of 59 responses were received, of which 43 provided the name of their academic institution. These 43 academics were employed at 37 departments in 34 different institutions; 31 were English universities and 3 were Scottish universities. Responses received from two or more academics from the same institution were consistent with each other.

The questionnaire was comprised mostly of closed questions requiring multiple-choice responses. The majority of questions were in the form of small scenarios describing various ways students have obtained, used, and acknowledged material.

The questionnaire contained two sections. Section one consisted of questions concerned with issues on what constitutes source-code plagiarism from the perspective of aca-

demics. Each question comprised several scenarios describing different ways students have used and acknowledged material from sources. A total of seven questions were included in section one of the questionnaire, and the scenarios of these questions covered issues on the following.

1. Source-code material subject to plagiarism,
2. Copying, adapting, converting source-code from one programming language to another, and using software for automatically generating source-code,
3. Methods used by students to gain material and present that material as their own work,
4. Group actions, self-plagiarism and collusion,
5. Intentional and unintentional plagiarism in graded and non-graded assignments,
6. Plagiarising by false referencing (i.e. falsification and fabrication), and
7. Considerations on plagiarism occurrences and student collaboration.

Most scenarios required respondents to select, from a choice of responses, the type of academic offence (if any) that in their opinion applied to each scenario. Choice of responses included 'plagiarism (or type of plagiarism)', 'other academic offence', 'not an academic offence', and 'do not know'.

The second section of the survey was concerned with gathering academics' responses on issues concerning plagiarism and the importance of an assignment in terms of its con-

tribution toward the overall module mark, and the amount of similarity that two pieces of source-code must contain in order for academics to take certain actions on plagiarism. Academics were also presented with two similar pieces of source-code and were asked to provide ratings as to how similar the pieces of source-code were, and the likelihood that plagiarism occurred. The data gathered from section two was mainly numerical, and statistical tests were applied for analysing the responses. These findings are described in detail in our research report [26]. During the design of the questionnaire, it was suspected that some academics might be reluctant to provide answers to questions asking for quantitative information due to the subjectivity issues surrounding the topic of similarity thresholds and source-code plagiarism. However, for research purposes, such questions were considered as important because they could provide an insight into similarity thresholds between source-codes for certain actions to be taken, that would be important during the development of software to detect source-code plagiarism.

Gathering the comments of academics on the various issues concerning plagiarism was important because of the variety both of university regulations in this area, and of the academics opinions on such a sensitive issue. Therefore, a text-box was included below each question for academics to provide any comments they had about issues related to the question asked. A detailed analysis of the responses to all survey questions is reported elsewhere [26]. The purpose of this survey was not to address in depth subjective issues, such as plagiarism intent and plagiarism penalties that could depend on student circumstances and university policies.

3.2 Survey Results

This section discusses academics' responses on issues surrounding source-code reuse and acknowledgement, assignment contribution, actions when plagiarism is suspected, and student collaboration in programming assignments. In the discussion that follows, the term *module* denotes a single subject of study; for example, the *Programming for Scientists module* is part of the undergraduate degree course (programme) *Computer Science*.

3.2.1 Plagiarised material

Plagiarism in programming assignments can go beyond the copying of source-code; it can include *comments*, *program input data*, and *interface designs*.

The process of investigating whether source-code plagiarism occurred may involve looking at other parts of a programming assignment since in some circumstances source-code alone may not be sufficient for identifying and proving plagiarism. A programming assignment may include design diagrams, source-code and other documentation. Academics were presented with the scenarios described below, and were asked to select one answer: 'Agree', 'Disagree', 'Neither agree or disagree'.

- Plagiarism in programming assignments can involve the:
  - **Scenario A:** Source-code of a computer program
  - **Scenario B:** Comments within the source-code

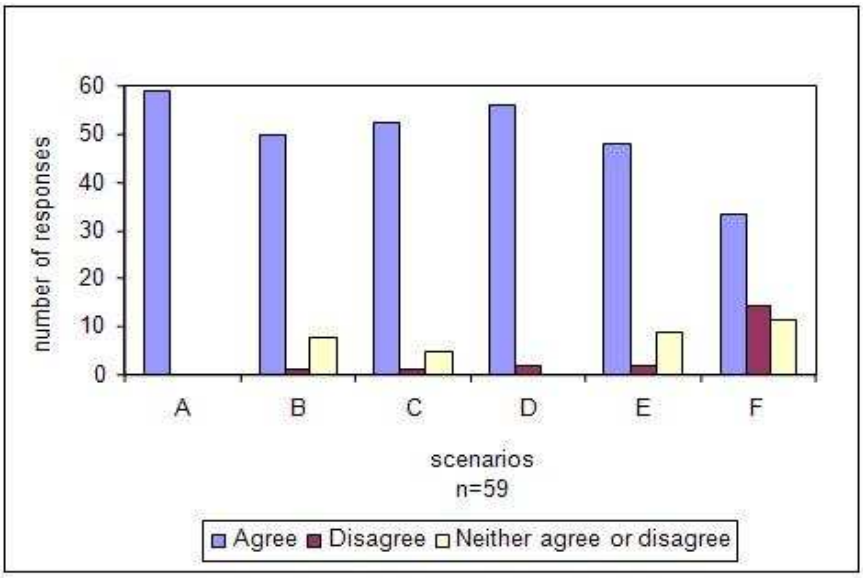

Figure 3.1: Scenarios and responses (1)

- **Scenario C:** Design material of a computer program
- **Scenario D:** Documentation of a computer program
- **Scenario E:** User interface of a computer program
- **Scenario F:** Program input data, i.e. for testing the program

The responses gathered are illustrated in Figure 3.1

All academics agreed that in a programming assignment source-code can be plagiarised. *Comments* within source-code can be plagiarized and may contribute towards identifying source-code plagiarism cases. The majority of academics agree that comments can be plagiarised and that comments may also help identify source-code plagiarism cases,

"comments are not in themselves a 'code plagiarism' issue but may help iden-

tify a case."

Program input data and the *user-interface* can be subject to plagiarism if they form part of the requirement in the assignment specification. The majority of respondents agreed that program input data can be subject to plagiarism but this alone cannot contribute to the identification of plagiarism. Three academics commented that copying input data is an issue if students are assessed on their testing strategies. When students are assessed on their testing strategies, assessment for plagiarism would occur by observing the testing strategy, including the datasets (e.g., input data) used for testing the program, and the testing material, including the test plan, system design documentation, technical documentation, and user manuals. One academic noted:

> "The input data are not necessarily plagiarism, but a testing strategy can be (i.e. what and how to perform the testing, datasets used, any UAT forms, etc.). The user interface is not necessarily generated by the user, but by the development tool, so it cannot be considered plagiarism in this case."

Interface designs submitted by students that appear suspicious need to be investigated for plagiarism if the assignment requires students to develop their own interface designs. Academics commented that whether a user-interface can be subject to plagiarism depends on the assignment requirements, and if a user-interface is required from the students then it can be subject to plagiarism,

> "The item on user-interface is really dependent on the task: if the UI was not

the essence of the program, then I wouldn't think this is an issue."

The majority of academics agreed that any material that is unreferenced can constitute plagiarism. One academic observed:

> "I require students to write their own code and documentation, except that there are safe-harbour provisions for permitted sources of each. They must cite their use to be able to claim the safe-harbour provision. The permitted sources are explicitly identified for the project."

3.2.2 Adapting, converting, generating, and reusing source-code

Academics were provided with scenarios concerned with the copying, adapting, and converting of source-code from one programming language to another, and using code-generating software for automatically creating source-code. A code-generator is an application that takes as input meta-data (e.g. a database schema) and creates source-code that is compliant with design patterns. An example of shareware code-generator software is JSPMaker [63] — when given a database this software quickly and easily creates complete source-code and a full set of JavaServer Pages [12] that have database connectivity. The given scenarios are as follows:

- **Scenario A**: A student reproduces/copies someone else's source-code without making any alterations and submits it without providing any acknowledgements.

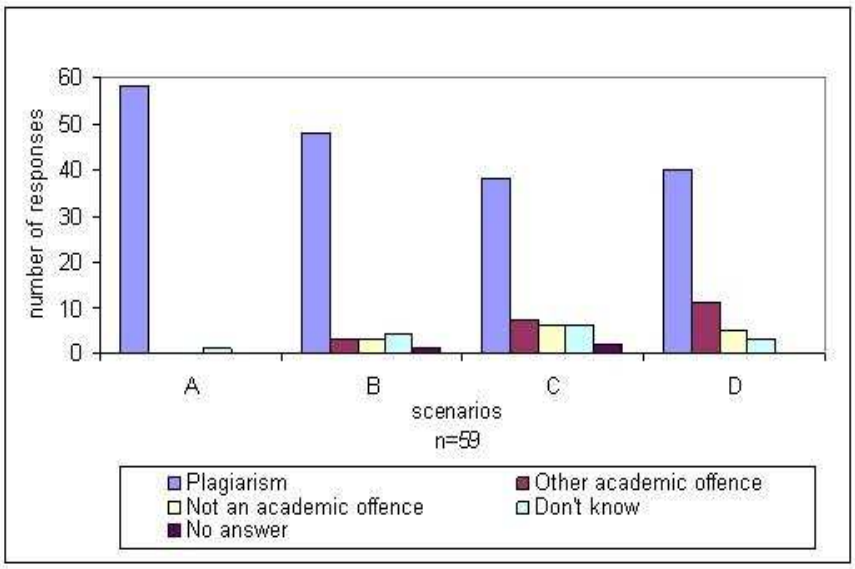

Figure 3.2: Scenarios and responses (2)

- **Scenario B**: A student reproduces/copies someone else's source-code, adapts the code to his/her own work and submits it without providing any acknowledgements.

- **Scenario C**: A student converts all or part of someone else's source-code to a different programming language and submits it without providing any acknowledgements.

- **Scenario D**: A student uses code-generating software (software that one can use to automatically generate source-code by going through wizards) and removes the acknowledgement comments that were automatically placed into the code by the software and submits it without providing any acknowledgements.

Responses to the scenarios are shown in Figure 3.2. Academics' comments on these scenarios raise important issues that are unique to source-code plagiarism.

Responses to scenarios A and B indicate a wide agreement that *reproducing/copying someone else's source-code with or without making any alterations and submitting it without providing any acknowledgements* constitutes source-code plagiarism. However, concerns were expressed on source-code reuse and acknowledgement. One academic commented:

> " ... in O-O environments where reuse is encouraged, obviously elements of reuse are not automatically plagiarism. I think I'd be clear on the boundaries and limits in any given circumstance, and would hope to be able to communicate that clarity to my students, but obviously there will potentially be problems. Use of the API would be legitimate without acknowledgement — or with only the implicit acknowledgement."

Regarding scenario B, many of the academics commented that adapting source-code may constitute plagiarism depending on the degree of adaptation, i.e., how much code is a copy of someone else's work and the extent to which that code has been adapted without acknowledgement. For example, a program may not be considered to be plagiarized if it was started by using existing source-code and then adapted to such an extent that it is beyond all recognition, so that there is nothing left of the original code to acknowledge. More of the respondents, however, have raised the issue of source-code reuse and acknowledgement. Specifically, one academic remarked:

> "... code copied from a web-site that assists in a specific task is potentially

> good practice. However, code that is a 100% copy is a different issue. I would also be concerned about the context of this copying. If the only deliverable were to be code and documentation the offence is clear. In this sense I suppose it is an issue of how much of the overall assignment is actually a copy of other work (without acknowledgement)."

On the issue of converting *all or part of someone else's source-code to a different programming language, and submitting it without providing any acknowledgements* (scenario C), several academics remarked that if the code is converted automatically without any or much effort from the student, then this procedure can constitute plagiarism. However, if a student takes the ideas or inspiration from code written in another programming language and creates the source-code entirely "from scratch", then this procedure is not likely to constitute plagiarism. Furthermore, in their comments academics have pointed out that taking source-code written in one programming language and converting it to a *similar* programming language, such as from C++ to Java, can constitute plagiarism. One academic, referring to scenarios A to D, emphasized:

> "In each case there must be some presumed benefit to the student in doing so (why did they do it otherwise?) and disruption to the assessment system. Even where the advantage might be minimal — e.g., from Prolog to C — a good student would almost certainly acknowledge the issue and use it to discuss the differences."

Academics were asked whether plagiarism takes place if *"a student uses code-generating software, removes the acknowledgement comments that were automatically placed into the code by the software, and submits it without providing any acknowledgements"*. The majority of the respondents considered unacknowledged use of code-generating software as plagiarism unless permission for use of such software is described in an assignment specification.

The findings suggest that students should be required to acknowledge any material they use that is not their own original work even when source-code reuse is permitted. All material should be acknowledged *regardless of licensing permissions*(e.g. open source, free-use, fair-use).

3.2.3 Methods of obtaining material

There are several ways students can gain source-code written by another author and present that material as their own work. In this question, the academic was asked to provide his/her opinion as to which academic offence applies to given scenarios. The scenarios describe methods used by students to gain material and present that material as their own work. The purpose of this question is to determine which student actions indicate source-code plagiarism.

The following scenarios were presented:

- **Scenario A**: A student pays another person (other than a student on the same module)

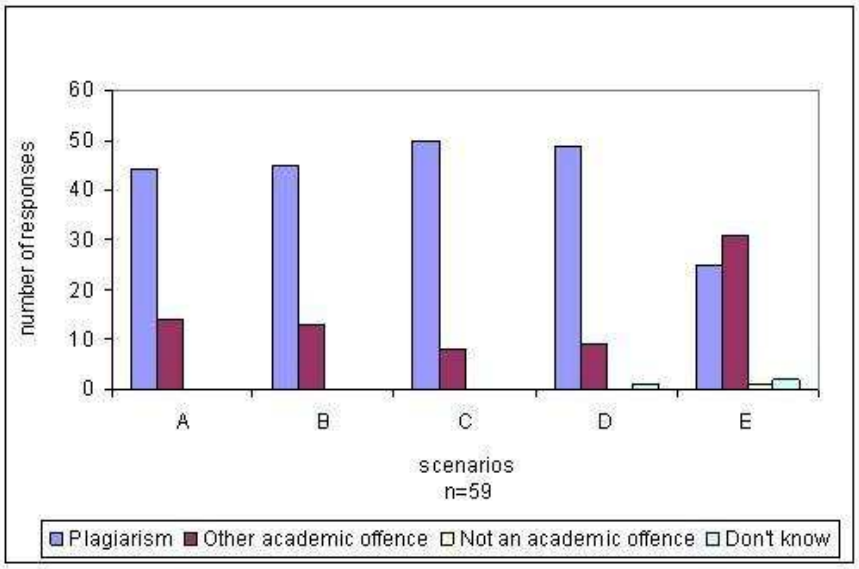

Figure 3.3: Scenarios and responses (3)

to create part or whole of source-code and submits it as his/her own work.

- **Scenario B**: A student pays a fellow student on the same module to create part or whole of source-code and submits it as his/her own work.

- **Scenario C**: A student steals another student's source-code and submits it as his/her own work.

- **Scenario D**: A student steals another student's source-code, edits it and submits it as his/her own work.

- **Scenario E**: A student intentionally permits another student to copy all or part of his/her programming assignment (including the source-code).

The responses of academics for are shown in Figure 3.3.

In scenarios A to D, plagiarism was committed alongside another academic offence(s) such as cheating and stealing, hence some academics chose 'plagiarism' and others 'other academic offence'. Many academics commented that these scenarios constitute plagiarism as well as other offences. One academic who considered scenarios A and B as 'other academic offence' and scenarios C and D as 'plagiarism' commented that

> "...paying someone or deliberately letting someone else use your code would both infringe the no-plagiarism declaration a student signs on submitting coursework - so would be plagiarism."

Another academic who considered scenarios A and B as 'other academic offence' and scenarios C and D as 'plagiarism' commented that

> "Serious cheating or deliberate plagiarism, a 'red-card offence'. I tell the students I am assessing them, not their sources, and this is an attempt to gain a qualification by fraud."

A third academic considered all scenarios to be 'plagiarism' justified by the following comment.

> "The act of submitting somebody else's work as your own without acknowledgment is generally plagiarism - in some of these cases other academic offences have been committed too. In particular stealing another student's work is an academic offence (and potentially a criminal or civil offence) in its own

right."

Regarding scenario E, comments received indicate that some of the academics consider this scenario to be collusion and some consider it plagiarism. Some of the academics that have considered this scenario as collusion commented, "intentionally permits..." is collusion., "It's a moot point what the name is, but I think I'd call the last one collusion.", and some of the academics that considered this scenario as 'plagiarism' provided comments such as, "I would see the last case as colluding in plagiarism - still an offence.", "The last is complicity in plagiarism, which is a form of academic misconduct", "At my university plagiarism and collusion are regarded as, and treated as, the same offence, so I'm not used to making any distinction between the two."

The findings show that there was a wide agreement between academics that this scenario is an academic offence, and whether it is regarded as collusion, plagiarism or another academic offence depends on university regulations. Some universities consider collusion and plagiarism as separate offences, while other universities do not make any distinctions between the two.

3.2.4 Source-code referencing and plagiarism

The respondents were asked to provide their views on scenarios regarding source-code referencing and plagiarism. The purpose of the given scenarios was to determine whether inappropriate referencing can suggest plagiarism. The results of question 6 are presented in

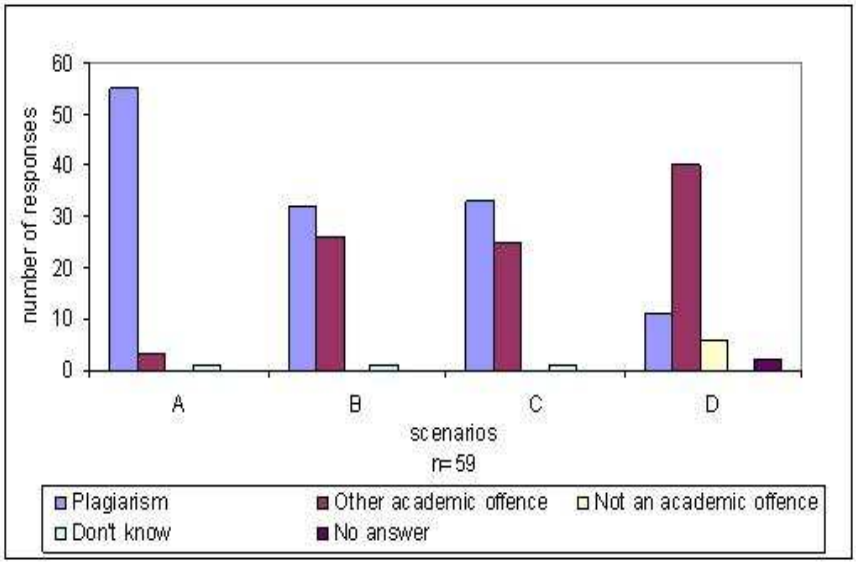

Figure 3.4: Scenarios and responses (4)

Figure 3.4, which corresponds to the following scenarios.

- **Scenario A**: Not providing any acknowledgements
- **Scenario B**: Providing pretend references (i.e. references that were made-up by the student and that do not exist)
- **Scenario C**: Providing false references (i.e. references exist but do not match the source-code that was copied)
- **Scenario D**: Modifying the program output to make it seem as if the program works

The results show that academics consider copying source-code from a book or a web-site, and not providing any acknowledgements as source-code plagiarism. For scenario A, one academic commented that there is a difference between poor referencing and plagia-

rism, and that the difference between poor referencing and plagiarism is one that it is not easily determined.

For scenarios B and C, academics commented that these scenarios show intent by the student to plagiarise. Regarding scenario D, if source-code was copied and unacknowledged this can constitute plagiarism as well as other academic offences characterised by academics as 'plagiarism', "falsification", "fraud", "cheating" and as "an act that it raises ethical, scientific and academic integrity issues".

The findings show that there was a wide agreement between academics that scenarios B, C, and D suggest an academic offence, however whether these are regarded as 'plagiarism' or 'another academic offence' seems to depend on university regulations.

3.2.5 Self-plagiarism and collusion in source-code

Academics were asked to provide their opinion as to which academic offence applies to given scenarios regarding group actions, *self-plagiarism* and *collusion*. In non-programming assignments, self-plagiarism occurs when a student reuses parts of an assignment previously submitted for academic credit and submits it as part of another assignment without providing adequate acknowledgement of this fact. In programming modules where source-code reuse is taught, self-plagiarism may not be considered as an academic offence. Collusion, occurs when two or more students collaborate on an assignment when the assignment requires students to work individually.

The scenarios presented to academics are as follows:

- **Scenario A**: For a group assignment, students in different groups exchange parts of source-code with the consent of their fellow group members, and integrate the borrowed source-code within their work as if it was that group's own work.
- **Scenario B**: For a group assignment, students in different groups exchange parts of source-code, without their fellow group members knowing, and integrate the borrowed codes within their work as if it was that group's own work.
- **Scenario C**: Assume that students were not allowed to resubmit material they had originally created and submitted previously for another assignment. For a graded assignment, a student has copied parts of source-code that he had produced for another assignment without acknowledging it.
- **Scenario D**: Two students work together for a programming assignment that requires students to work individually and the students submit very similar source-codes.

The responses of academics for are shown in Figure 3.5.

The majority of the academics consider scenarios A, and B as plagiarism. Considering the comments given by academics to previous questions, these scenarios can constitute plagiarism, collusion or another academic offence depending on university regulations. Scenario B can constitute plagiarism, collusion as well as another academic offence since material was obtained without permission (the type of academic offence depends on the academic regulations of each university). Regarding scenarios A and B, some academics

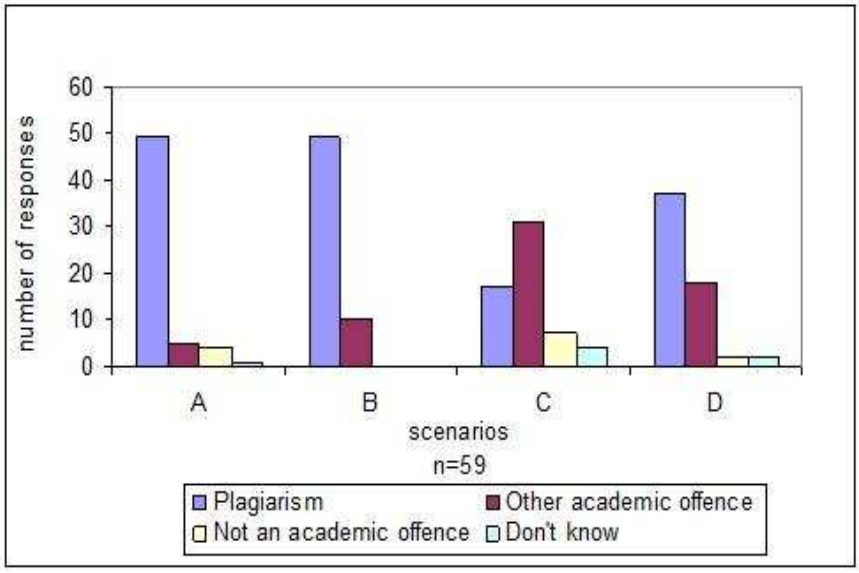

Figure 3.5: Scenarios and responses (5)

expressed some concerns on the issue of plagiarism in group assignments. One academic commented,

> "I avoid group assignments for just these reasons. I permit students to work together as long as they individually write their own code and documentation. I regard this as pedagogically valuable",

while another academic observed

> "...Some students learn better while working in groups but they must ensure that the work submitted is original and their own. The tell tale here is the comments (and spelling mistakes) are identical."

The majority of academics considered scenario D as an academic offence, 'plagiarism'

or ‘another academic offence’. However, as mentioned above the name depends on university regulations. Academics commented,

> “the last one [scenario D] is very tricky to decide; I’m very happy for students to work together, but very unhappy when I have to make this kind of decision. I’d be inclined to say ‘plagiarism’, but treat a first offence leniently,”

another academic considered this scenario as

> “very common and usually accompanied with denials that they had submitted the same code and/or they didn’t know they weren’t allowed to work together.”

In student assignments, self-plagiarism occurs when a student copies entire or parts of his/her own assignment and submits it as part of another assignment without providing proper acknowledgement of this fact. However, when we asked academics whether it constitutes plagiarism if a student resubmits source-code they have originally created and submitted previously for another assignment (see scenario C) we received some controversial responses. The majority of the academics (48 out of 59) characterised this scenario as an academic offence (17 as plagiarism and 31 as other academic offence). In their comments, those academics characterised this scenario as “self-plagiarism”, “breach of assignment regulations if resubmission is not allowed”, and “fraud if resubmission is not acknowledged”.

Some academics argued that in object-oriented environments, where reuse is encouraged, it is inappropriate to prevent students from reusing source-code produced as part of

another programming assignment. The comments and responses provided by the academics who do not consider the given scenario to constitute plagiarism or another academic offence point to the controversial issue on source-code reuse mentioned above. One academic who provided a 'do not know' response remarked that students should reuse source-code where possible, while another, who was clear that the given scenario 'was not an academic offence', emphasized

> "I find it hard to assume that students were not allowed to resubmit material."

A third academic, who also stated that the given scenario 'was not an academic offence', asked rhetorically

> "Would this ever happen in an object-oriented programming module when we behove students not to reinvent the wheel?"

In conclusion, since 48 out of 59 academics characterised the action of resubmitting source-code produced as part of another assessment as a type of academic offence (plagiarism or other) we can conclude that resubmitting source-code without providing appropriate acknowledgements may lead to an academic offence if this is not allowed for the particular assignment.

3.2.6 Plagiarism in graded and non-graded assignments

Academics were presented with brief scenarios concerning intentional and unintentional plagiarism in *graded* and *non-graded* assignments. By non-graded assignments, we refer to assignments that do not contribute to the overall module mark (such as some laboratory and tutorial exercises).

In this question, academics were asked to provide their opinions as to which academic offence applies to given scenarios. The purpose of this question is to determine whether intentions and assignment importance influence the decision as to whether plagiarism has occurred.

Figure 3.6 corresponds to the following scenarios:

- **Scenario A**: For a graded assignment, a student has copied source-code from a book and has intentionally not provided any acknowledgements.
- **Scenario B**: For a graded assignment, a student has copied source-code from a book and has unintentionally not provided any acknowledgements.
- **Scenario C**: For a non-graded assignment, a student has copied source-code from a book and has intentionally not provided any acknowledgements.
- **Scenario D**: For a non-graded assignment, a student has copied source-code from a book and has unintentionally not provided any acknowledgements.

The responses of academics for are shown in Figure 3.6.

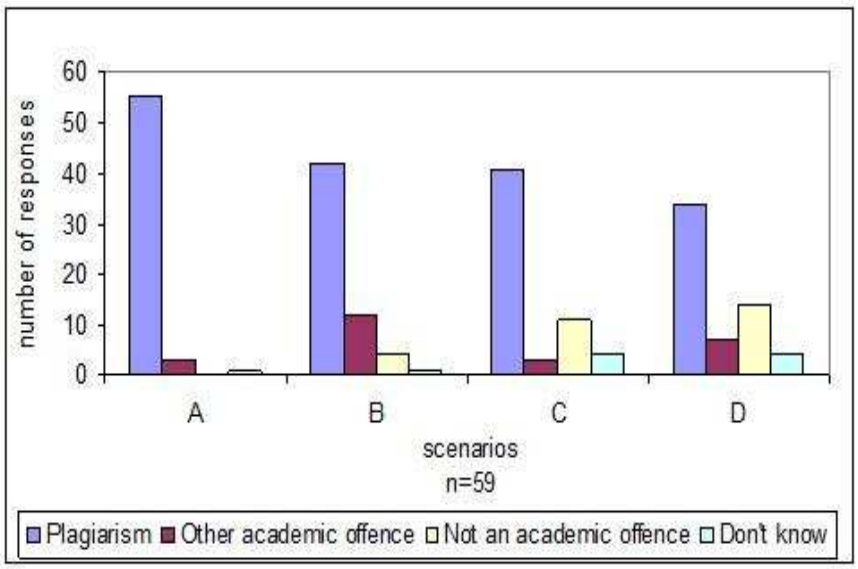

Figure 3.6: Scenarios and responses (6)

The results show that plagiarism can occur regardless of whether or not the student intended to provide acknowledgements to the sources they used. Academics commented that the degree of intent determines the seriousness of the plagiarism and consequently the penalty applied to the students work. Hence, plagiarism can occur intentionally or unintentionally, and the penalty imposed to the student will depend on the degree of the student's intention to commit plagiarism,

> "It is a case of degree here. Typically students do not just fail to reference one source but many. For non graded work (presumably being used formatively) it would be better to highlight the error without formal censure."

The results also suggest that plagiarism can occur regardless of whether an assignment is graded or non-graded. Two academics commented that plagiarism punishments for

graded and non-graded assignments may vary. The first academic who provided a 'don't know' response for scenarios C and D commented that

> "[for scenarios C and D] it depends on what the rules were for that non-graded assignment. If they plagiarise then it is plagiarism irrespective of whether the assignment is graded, it just means that the penalty may be different."

The second academic who provided a 'plagiarism' response for both scenarios C and D commented that

> "the act of plagiarism should be related to whether an assignment is graded or not. The intention also would have to do with the amount of penalty applied to."

In addition, many academics commented that action against students should not be taken for non-graded assignments, and in such cases, the students should be approached and informed or warned about plagiarism implications on graded assignments

> "If the assignment is not contributing towards the mark for the module then the correct protocol should be brought to the attention of the student."

In addition one academic who considered scenarios C and D as 'not an academic offence' commented,

> "My logic here is that you can't penalise someone for an assignment they didn't have to do. The last two questions are still plagiarism in my view but you couldn't take any action against the student for it."

Furthermore, some of the academics commented that their university regulations on plagiarism would only apply to graded work and plagiarism is only an academic offence if it concerns work submitted for credit. One academic stated

> "last two *are* cases of plagiarism but academic regulations might define 'offence' in terms of intention to gain higher grades."

Another academic observed that the

> "last two are not offences but morally incorrect and could lead to the practice being repeated for assessed work when it will be plagiarism."

In conclusion, copying without providing any acknowledgements can constitute plagiarism whether this is done intentionally or unintentionally. Plagiarism can occur in both graded and non-graded assignments, regardless of the assignments contribution to the overall course mark. The degree of intent determines the seriousness of the plagiarism. The penalty imposed from plagiarism in graded and non-graded assignments may differ and may depend on the degree of intent to commit plagiarism.

Some university regulations on plagiarism seem to apply to graded assignments only. Many academics have commented that when plagiarism is detected in non-graded assign-

ments, the students should be approached and informed or warned about its implications on graded assignments. Such measures are not always explicitly described in university regulations.

3.2.7 Assignment contribution

When prompted to consider the question: *'What would have to be the minimum weight of an assignment towards the overall module mark for you to proceed with investigation into plagiarism?'*, responses were largely uniform. As Figure 3.7 illustrates, the majority (32 out of 51) or respondents cited a value in the range of 0-5 %, suggesting a "zero tolerance" policy. Furthermore, there was agreement that investigation into possible plagiarism cases should always be pursued and appropriate action taken (either in the form of penalties or as warnings) regardless of the assignment's contribution towards the overall module mark.

The rationale for this attitude was clearly articulated by one respondent who warned

> "If they cheat on a 5% assignment they will cheat on a larger one. A short, sharp shock can prevent the full offence."

In some instances, the severity of the offence is considered by the institution to vary depending on the contribution of the assignment,

> "Any contribution would bring the same investigation. However, for the full range of penalties to be applied a minimum contribution to the degree award of

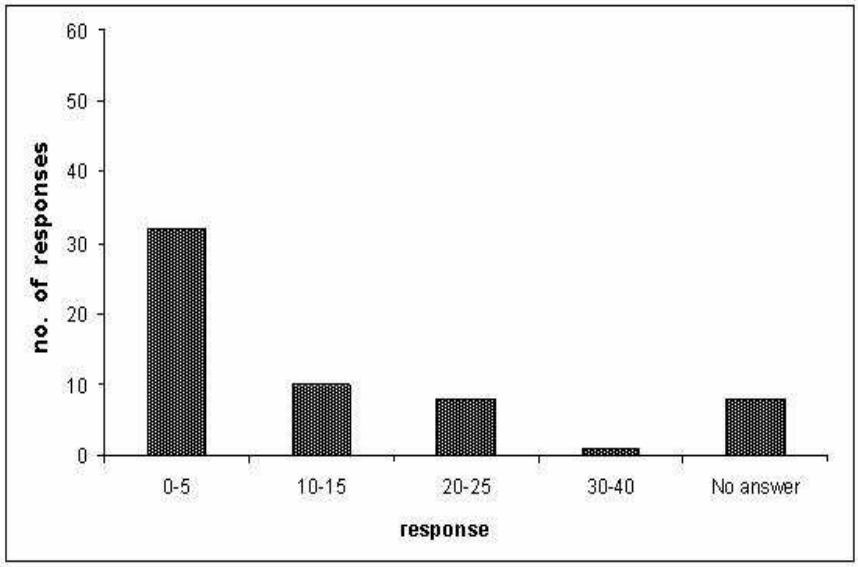

Figure 3.7: Responses to minimum assignment weight scenario.

7%, or a second or subsequent upheld accusation, is required. This is university policy."

One academic commented,

"If the assignment is not contributing towards the mark for the module then the correct protocol should be brought to the attention of the student."

3.2.8 Plagiarism occurrences and considerations on student collaboration

The survey raised issues on appropriate and inappropriate collaboration between students. There was general agreement among respondents that it is "pedagogically valuable" for students to engage actively in sharing ideas while discussing assignments as long as they do not copy each other's work. Three academics who encourage students to share ideas

commented on the benefits and pitfalls of such practice:

> "I have no problem with sharing ideas. Given the restricted types of programs undergraduate students write, it is inevitable that certain design decisions lead to similar code. "

This position was endorsed and elaborated by another academic who articulated a straightforward view on the problem,

> "I am personally not too worried about students discussing their work, it is the outright copying that I find most offensive and educationally pointless."

A third academic observed that it is

> "Important to remember that we often expect people to engage actively in discussing content and assignments. Also, as preparation for future working in teams, we often need to be much clearer in what we require of students — and what we don't."

Respondents noted that occurrences of plagiarism vary (both in type and in frequency) depending on the tasks being undertaken at the time. Respondents reported that occurrences of plagiarism when students are testing and debugging their software appeared to depend on the type of testing being carried out by the students. They remarked that occurrences of plagiarism during white box testing (tests requiring access to the code of the program under

test) tend to be more frequent than during black box testing (tests conducted at the software interface level) due to the nature of these tests.

In addition, respondents noted that the distribution of marks awarded for the components of an assignment influences in which of those component tasks plagiarism may occur. For example, if the credit awarded for the design of a program is relatively high compared to the credit for the coding, then students are more likely to plagiarize when performing the design task.

Sharing ideas and sharing work were considered as two very different issues. Although academics expressed no objections to students sharing ideas, they opposed the practice of students collaborating and submitting similar work when assignments required them to work individually.

3.3 Source-Code Plagiarism: Towards a Definition

Based on the responses summarized above, the following is suggested as a new definition of what constitutes source-code plagiarism in an academic context.

Source-code plagiarism in programming assignments can occur when a student *reuses* (3.3.1) source-code authored by someone else and, intentionally or unintentionally, fails to *acknowledge it adequately* (3.3.3), thus submitting it as his/her own work. This involves *obtaining* (3.3.2) the source-code, either with or without the permission of the original au-

thor, and *reusing* (3.3.1) source-code produced as part of another assessment (in which academic credit was gained) without adequate acknowledgement (3.3.3). The latter practice, *self-plagiarism*, may constitute another academic offence.

3.3.1 Reusing

Reusing includes the following:

1. Reproducing/copying source-code without making any alterations

2. Reproducing/copying source-code and adapting it minimally or moderately; minimal or moderate adaptation occurs when the source-code submitted by the student still contains fragments of source-code authored by someone else.

3. Converting all or part of someone else's source-code to a different programming language may constitute plagiarism, depending on the similarity between the languages and the effort required by the student to do the conversion. Conversion may not constitute plagiarism if the student borrows ideas and inspiration from source-code written in another programming language and the source-code is entirely authored by the student.

4. Generating source-code automatically by using code-generating software; this could be construed as plagiarism if the use of such software is not explicitly permitted in the assignment specification.

Where source-code reuse is not allowed, reusing (3.3.1) source-code authored by someone else (or produced by that student as part of another assessment) and providing acknowledgements may constitute a breach of assignment regulations, rather than plagiarism (or self-plagiarism).

3.3.2 Obtaining

Obtaining the source-code either with or without the permission of the original author includes:

1. Paying another individual to create a part of or all of their source-code
2. Stealing another student's source-code
3. Collaborating with one or more students to create a programming assignment which required students to work individually, resulting in the students submitting similar source-codes; such inappropriate collaboration may constitute plagiarism or *collusion* (the name of this academic offence varies according to the local academic regulations).
4. Exchanging parts of source-code between students in different groups carrying out the same assignment with or without the consent of their fellow group members.

Incidents of source-code plagiarism can co-occur with other academic offences (such as theft, cheating, and collusion) depending on academic regulations. The list above is in-

dicative of key areas where this form of plagiarism occurs, but it is certainly not exhaustive, since there are numerous ways that students can obtain source-code written by others.

3.3.3 Inadequately acknowledging

Inadequately acknowledging source-code authorship includes:

1. Failing to cite the source and authorship of the source-code, within the program source-code (in the form of an in-text citation within a comment) and in the appropriate documentation.

2. Providing fake references (i.e. references that were made-up by the student and that do not exist); this is a form of academic offence, often referred to as *fabrication*, which may co-occur with plagiarism.

3. Providing false references (i.e. references exist but do not match the source-code that was copied); another form of academic offence, often referred to as *falsification*, which may co-occur with plagiarism.

4. Modifying the program output to make it seem as if the program works when it is not working; this too is a form of academic offence akin to falsification, which may co-occur with plagiarism.

3.4 Conclusion

In this Chapter we discussed academics responses on issues surrounding source-code reuse and acknowledgement. A general consensus exists among academics that a *zero tolerance* plagiarism policy is appropriate; however some issues concerning source-code reuse and acknowledgement raised controversial responses.

The responses revealed that a wide agreement exists among academics on the issue of what can constitute source-code plagiarism. Because of the object-oriented nature of some programming languages, some academics have identified important issues concerned with source-code reuse and acknowledgement (including self-plagiarism). They also noted differences between the approach to plagiarism adopted for graded and non-graded work. The survey findings were used to suggest a definition of what constitutes source-code plagiarism from the perspective of academics who teach programming on computing courses.

Much survey-based research exists addressing the prevalence of source-code plagiarism in academia. However, surveys on the issue of what constitutes source-code plagiarism in UK universities are rare in academic scholarship. In addition, there appears to be no commonly agreed description of what constitutes source-code plagiarism from the perspective of academics who teach programming on computing courses.

Differences among university policies, assignment requirements, and personal academic preferences, can create varied perceptions among academics and students on what constitutes source-code plagiarism. The fact that source-code reuse is encouraged in object-

oriented programming may lead students to take advantage of this situation, and use or adapt source-code written by other authors without providing adequate acknowledgements.

Since reuse is encouraged in object-oriented programming, some academics have expressed different opinions on issues surrounding source-code reuse and acknowledgement. The majority of respondents agreed that, when reuse is permitted, students should adequately acknowledge the parts of the source-code written by other authors (or that the students have submitted as part of another assessment) otherwise these actions can be construed as plagiarism (or self-plagiarism).

Responses show that university policies influence the actions academics can take when they detect plagiarism. Not all universities, however, apply these policies to assignments that are not submitted for academic credit.

Academics teaching programming should inform students clearly of their preferences especially on source-code reuse and acknowledgement. Avoiding confusion among academics and students is likely to reduce the occurrences of plagiarism.

Chapter 4

A Small Example of Applying Latent Semantic Analysis to a Source-Code Corpus

This Chapter provides a theoretical and practical introduction to the LSA information retrieval technique using a context specific example. We describe the foundations for understanding the basics of LSA using a step-by-step procedure for discussing the application of LSA to an example source-code corpus. The evaluation measures described in this Chapter will be referred to later on in the thesis.

4.1 Information Retrieval and the Vector Space Model

Information retrieval is an area concerned with the representation, storage, retrieval, and maintenance of information items. In the past 30 years the area of information retrieval has expanded to include tasks such as document classification and categorisation, systems architecture, data visualisation, user interfaces, information filtering, and cross-language document retrieval [5]. Amongst information retrieval systems is the Vector Space Model (VSM) [120]. Its first use was in the SMART (Saltons Magic Automatic Retriever of Text) retrieval system developed at Cornell University in the 1960's.

In the Vector Space Model, a corpus of documents is represented as an $m \times n$ term-by-document matrix *A*, where *m* (the rows of the matrix) is the number of unique words in the corpus, *n* (the columns of the matrix) is the number of documents, and each element of matrix A, a_{ij}, is the frequency (or weighted frequency) at which word *i* occurs in document *j*. Thus in the VSM, the rows of matrix A represent the term vectors, and the columns of matrix A represent the document vectors.

The idea behind the VSM is that similarities between vectors can be computed using geometric relationships. Documents represented as vectors can be compared to user queries (represented as vectors), to determine their similarity, using a similarity measure such that documents similar to the query can be retrieved. Similarity can be measured using similarity coefficients [48] that measure the distance between a document and a query vector. The traditional method for computing the similarity between two vectors is the cosine measure

of similarity which measures the size of the angle between the two vectors.

4.2 What is Latent Semantic Analysis?

Latent Semantic Analysis (LSA) is a variant of the VSM for information retrieval. Literature on the application of LSA for information retrieval dates back to 1988. LSA is also known as Latent Semantic Indexing (LSI), and the term LSI is used for tasks concerning the indexing or retrieval of information, whereas the term LSA is used for tasks concerned with everything else, such as automatic essay grading and text summarisation.

Dumais *et al.* [40] proposed the use of LSA as a potential technique for overcoming deficiencies of text retrieval systems such as the VSM. Because of the variability in word usage, i.e. variability in the words people use to describe the same word or concept (synonymy), term matching methods often fail to retrieve information relevant to the user's query. In fact, empirical evidence suggests that the likelihood of two people choosing the same keyword to describe a familiar object or concept is between 10% and 15% [45]. Furthermore, one word can have more than one meaning (polysemy), which leads to irrelevant information being retrieved. From this perspective, exact lexical-matching methods are deficient for information retrieval. Dumais [40] proposed LSA as an information retrieval technique to overcome the problems of exact term-matching models.

The LSA technique comprises of mathematical algorithms that are applied to text collections. Initially a text collection is pre-processed and represented as a term-by-documents

matrix containing terms and their frequency counts in files. Matrix transformations (or term weighting algorithms) are applied to the values of terms such that the values of terms in documents are adjusted depending on how frequently they appear within and across documents in the collection. A mathematical algorithm called Singular Value Decomposition (SVD) decomposes this term-by-document matrix into separate matrices that capture the similarity between terms and between documents across various dimensions in space. The aim is to represent the relationships between terms in a reduced dimensional space such that noise (i.e. variability in word usage) is removed from the data and therefore uncovering the important relations between terms and documents obscured by noise [16]. LSA aims to find underlying (*latent*) relationships between different terms that have the same meaning but never occur in the same document.

Take for example, two documents which both describe how a computer monitor works but one uses the terms *computer monitor* and the other documents uses the term *computer screen* but both documents have similar content. LSA derives the meaning of terms by estimating the structure of term usage amongst documents through SVD. This underlying relationship between terms is believed to be mainly (but not exclusively) due to transitive relationships between terms, i.e., terms are similar if they co-occur with the same terms within documents [76]. Traditional text retrieval systems cannot detect this kind of transitive relationship between terms and the consequence of this is that relevant documents may not be retrieved. LSA categorises terms and documents into a semantic structure depending on their semantic similarity, hence *latent semantic* in the title of the method.

LSA derives the meaning of terms by analysing their context of word usage and thus it is *guessing* the meaning of a word. It represents each document as a vector in the vector space, and treats each document as a *bag of words* ignoring word order. Researchers have explored the level of meaning that LSA can extract from English texts without the use of word order [80, 84, 117, 145]. Landauer *et al.* [84] compared essays written by students with target instructional texts (i.e. by computing the cosine between the student essay vector and the target essay vector) and human judgments. Statistical correlations revealed little difference between LSA scores and human judgments. The authors conclude that LSA represents meaning from text as accurately as humans however it does not require the use of word order and syntax as humans do.

The work done by researchers on applying LSA to automate the essay grading process [84, 117, 47, 143, 67, 65] is rather similar to the task of using LSA to detect similar source-code files. This involves giving LSA a source-code file (represented as a target file vector) and using it to retrieve similar files. LSA then returns a similarity value between the target file and its similar files.

SVD appears to be the power behind LSA. The relations between terms are not explicitly modeled in the creation of the LSA space, and this makes it difficult to understand how LSA works in general [94]. Many models for understanding LSA have been proposed [148, 131, 36, 76]. One recent model for understanding the performance of LSA is that by Kontostathis and Pottenger [76], who provide a framework for understanding the LSA performance in the context of information search and retrieval. They provide mathemati-

cal proof that the SVD algorithm encapsulates co-occurrence and it is from this that LSA generates semantic knowledge of text.

LSA derives semantic similarities from analysing the co-occurrence of words in a corpus and relies on high dimensional vector spaces to represent meanings [33]. The meaning of a word is determined by analysing its relationship to other words rather than defining the meaning of a word by its properties (i.e., function, role, etc.). For example, the meaning of a word is defined by its degree of association with other words, for example the word *computer* is very close to words *keyboard* and *screen*, and far from words like *swim* and *umbrella* [33]. LSA needs enough information to analyse in order to derive the semantics of words successfully.

LSA is known as an intelligent information retrieval technique and is typically used for indexing large text collection and retrieving files based on user queries. In the context of text-retrieval LSA has been applied to a variety of tasks including indexing and information filtering [43]; essay grading [84, 117, 47, 143, 67, 65]; modeling children's semantic memory [33]; cross-language information retrieval [146]; information visualisation [82]; indexing essays [32, 38, 16]; text summarisation [74, 130]; giving feedback on essays [145, 19]; grading of free-text responses [140, 110]; e-assessment [134]; student tutoring [142]; source-code clustering and categorisation [71, 89, 77, 91, 92, 95], and for detecting plagiarism in natural language text [19].

4.3 An Introduction to the Mathematics of LSA

LSA starts by transforming the pre-processed corpus of files into a $m \times n$ matrix $A = [a_{ij}]$, in which each row m represents a term vector, each column n represents a file vector, and each cell a_{ij} of the matrix A contains the frequency at which a term *i* appears in file *j* [16].

Term weighting is then applied to matrix A. The purpose of term weighting is to increase or decrease the importance of terms using *local* and *global* weights in order to improve retrieval performance. With *document length normalization* the term values are adjusted depending on the length of each file in the corpus [14].

The matrix is then submitted for Singular Value Decomposition (SVD) to derive the latent semantic structure model. Singular Value Decomposition decomposes matrix A into the product of three other matrices an $m \times r$ term-by-dimension matrix, *U*, an $r \times r$ singular values matrix, Σ, and an $n \times r$ file by dimension matrix, V. The original matrix A can be reconstructed by multiplying the three matrices through $U\Sigma V^T$ where V^T denotes the transpose of matrix *V*.

The rank *r* of matrix A is the number of nonzero diagonal elements of matrix Σ. SVD can provide a rank-k approximation to matrix A, where *k* represents the number of dimensions (or factors) chosen, and $k \leq r$. This process is known as dimensionality reduction, which involves truncating all three matrices to *k* dimensions. This is done by selecting the first *k* columns from matrices U, Σ, and V, and the rest of the values are deleted (or set to zero). The reduced matrices are denoted by U_k, Σ_k, and V_k where U_k is a $m \times k$ matrix,

Σ_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 through $A_k = U_k \Sigma_k V_k^T$.

It is important when computing the SVD that *k* is smaller than the rank *r*, because it is this feature that reduces noise in data and reveals the important relations between terms and documents [16, 13]. In addition, by reducing the sizes of matrices, computation time is also reduced and this increases computational efficiency. Techniques for aiding with the selection of dimensionality are discussed in Chapter 5. The most effective and commonly used method for selecting the number of dimensions is through experimentation. In order to perform such experiments, a number of queries and their relevant files must be known such that the information retrieval performance of a system is evaluated when using various *k* dimensional settings.

One common task in information retrieval systems involves a user placing a query in order to retrieve files of interest. A user's query comprises of a set of words, and in an LSA based system, a query must be represented as a query vector and then projected into the term-file space and then compared to all other file vectors.

The elements of the query vector contain the weighted term-frequency counts of the terms that appear in the query, using the same weighting scheme applied to matrix A.

Thus, given a query vector *q*, whose non-zero elements contain the weighted term frequency values of the terms, the query vector can be projected to the *k*-dimensional space using Function 4.1 [16].

$$Q = q^T U_k \Sigma_k^{-1}, \tag{4.1}$$

where on the left hand side of the equation, Q is a mapping of q into latent semantic space, and on the right hand side of the equation q is the vector of terms in the user's weighted query; q^T is the transpose of q; and $q^T U_k$ is the sum of the k-dimensional term vectors specified in the query, multiplied by the inverse of the singular values Σ_k^{-1}. The singular values are used to separately weight each dimension of the term-file space [16].

Once the query vector is projected into the term-file space it can be compared to all other existing file vectors using a similarity measure. One very popular measure of similarity computed the *cosine* between the query vector and the file vector. Typically, using the cosine measure, the angle between the query vector and all file vectors is computed and the files are ranked according to their similarity to the query, i.e. how close they are to the query in the term-file space. All files or those files with a similarity value exceeding a threshold, are returned to the user in a ranked list sorted in descending order of similarity values, i.e. with the files most similar to the query displayed in the top ranked. The quality of the results can be measured using evaluation measures, such as those discussed in Section 4.10.

In the term-by-file matrix A that has columns a_j the cosine similarity between the query vector $q = (t_1, t_2, \ldots, t_m)^T$ and the n file vectors is given by [14]:

$$cos\Theta_j = \frac{a_j^T q}{\parallel a_j \parallel_2 \parallel q \parallel_2} = \frac{\sum_{i=1}^{m} a_{ij} q_i}{\sqrt{\sum_{i=1}^{m} a_{ij}^2} \sqrt{\sum_{i=1}^{m} q_i^2}} \tag{4.2}$$

Table 4.1: U_2, a 2×2 matrixΣ_2

Term name	U_2			Σ_2	File ID		V_2		
addtosubtree	-0.039	-0.173		1.744	0.000		A1	-0.221	-0.293
addword	-0.087	-0.134		0.000	1.458		A2	-0.221	-0.293
binarytreenode	-0.039	-0.173					B1	-0.244	-0.314
boolean	-0.087	-0.134					B2	-0.244	-0.314
break	-0.092	-0.146					C1	-0.118	-0.454
checkword	-0.087	-0.134					C2	-0.128	-0.460
compareto	-0.039	-0.173					D1	-0.420	0.236
elementat	-0.092	-0.146					D2	-0.414	0.275
else	-0.039	-0.173					E1	-0.450	0.202
equals	-0.092	-0.146					E2	-0.450	0.202
equalsignorecase	-0.087	-0.134							
false	-0.087	-0.134							
floor	-0.210	0.113							
for	-0.225	-0.097							
getleft	-0.039	-0.173							

for $j = 1, \ldots, n$ where *n* is equal to the number of files in the dataset (or the number of columns in the term-by-file matrix A).

4.3.1 An illustration of the LSA process

In this Section we show the output matrices created after applying SVD to the corpus described in Section 4.4. When computing dimensionality reduction the value of k is set to 2 dimensions.

Once SVD is applied to the matrix shown in Table 4.6, a 44×2 matrix U_2, a 2×2 matrix Σ_2, and a 10×2 V_2 are created. Table 4.1 shows an extract of the U_2 matrix and the Σ_2, and V_2 matrices. The name of each term, and each file ID was included in Table 4.1 for clarity.

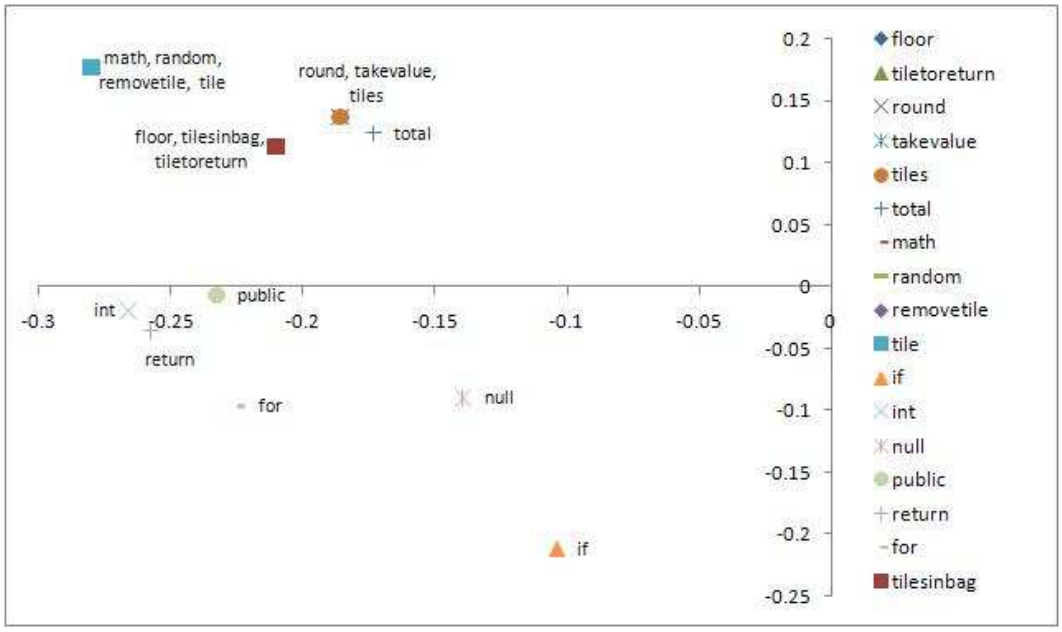

Figure 4.1: Terms and files in a two-dimensional LSA space.

As an example for visualising the similarity between terms in the LSA space, Figure 4.1 shows the terms described in Table 4.8 and the files in which they belong to, plotted in a two-dimensional space. To create the two-dimensional space, SVD was computed by setting the value of k to 2, and then plotting the values of the U_2 and V_2 matrices. The values in the first dimension were the x co-ordinates, and the values in the second dimension were the y co-ordinates.

The results in Figure 4.1, show that the terms that occurred in similar files, were close together in the LSA space, and terms that occur without any particular pattern, such as frequently occurring terms, received negative co-coordinate values and where not grouped together in the LSA space. Figure 4.2 illustrates the angle between terms and files. Sim-

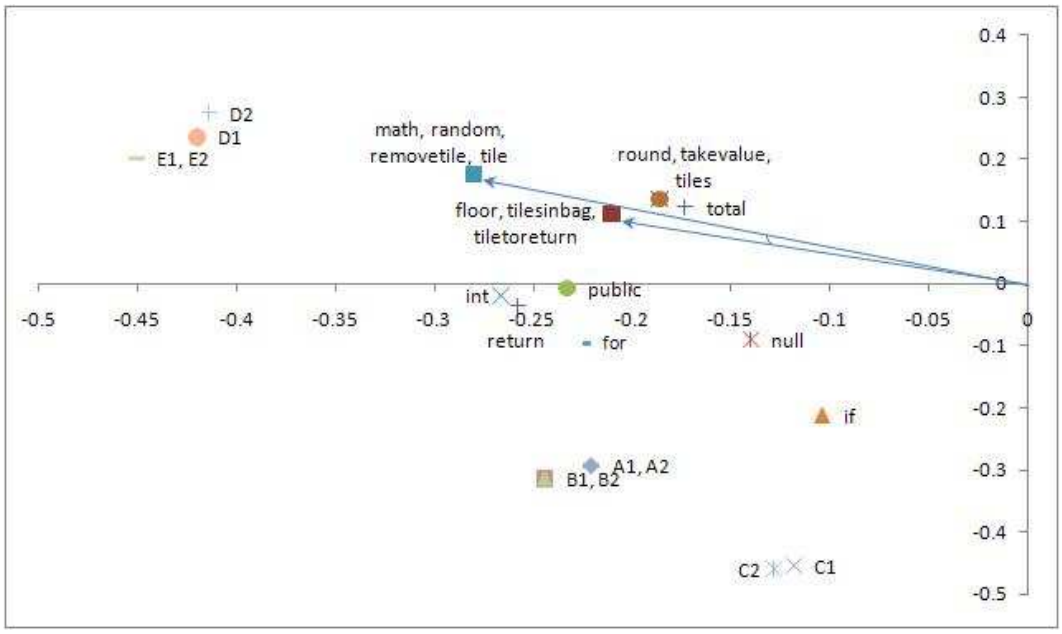

Figure 4.2: Terms and files in a two-dimensional LSA space - illustrating the cosine.

ilarly, when projecting a query Q to the LSA space, the cosine between the query Q and each of the terms is computed, and files closest to the query, i.e. whose similarity is above a given threshold, are retrieved.

In the discussion that follows, we demonstrate, using a small example, how LSA can be applied to a corpus of files.

4.4 The Example Source-Code Corpus

The example corpus comprises of five source-code file pairs, and each file contains a source-code fragment. Each of these file pairs contain source-code whose structure is either identical or very similar. However, the files within those file pairs contain different terms used to

describe the same concepts. The file pairs are: A1-A2, B1-B2, C1-C2, D1-D2, and E1-E2. Files D1-D2, E1-E2 are considered relatively more similar than the rest of the source-code files since their source-code fragments have similar functionality. The files comprising the small source-code corpus are as follows.

A1 - locate:

```
private int locate(Object thing)   {
int CNumber = 0;
int ArrIndex= -1;

for (int i = 0; i < spelling.size(); i = i + 1)
{
CNumber = CNumber + 1;
if (spelling.elementAt(i).equals(thing))
{
ArrIndex = i;
break;
}
}
return ArrIndex;
}
```

A2: findLinear

```
private int findLinear(Object thing)  {
int CompNumber = 0;
int ArrayIndex= -1;

for (int i = 0; i < spellings.size();
i = i + 1)
{
CompNumber = CompNumber + 1;
if (spellings.elementAt(i).equals(thing))
{
ArrayIndex = i;
```

```
break;
}
}
return ArrayIndex;
}
```

B1: addWord

```
public void addWord(String word){
store[s]=word;
s++;}
public boolean checkWord(String word){
for (int i=0; i<s; i++)
{
if (store[i].equalsIgnoreCase(word))
{
return true;
}
}
return false;}
}
```

B2: addword

```
public void addWord(String word){
store[p]=word;
p=p+1;}
public boolean checkWord(String word){
for(int i=0; i<p; i=i+1)
{
if(store[i].equalsIgnoreCase(word))
return true;
}
return false; }
}
```

C1: addToSubTree

```
private void addToSubTree(BinaryTreeNode n, String v)
```

```
{
if (v.compareTo(n.getValue()) < 0)
{
if (n.getLeft()==null)
n.setLeft(new BinaryTreeNode(v));
else addToSubTree(n.getLeft(), v);}
else
{
if (n.getRight()==null) n.setRight(new BinaryTreeNode(v));
else addToSubTree(n.getRight(), v);
}
}
```

C2: addToSubTree

```
private void addToSubTree(BinaryTreeNode n, String s)
{
if (n!=null)
{
String nValue = (n.getValue());
if (s.compareTo(nValue) <= 0)
{
if (n.getLeft()==null)
n.setLeft(new BinaryTreeNode(new String(s)));
else
addToSubTree(n.getLeft(), s);
}
else
{
if (n.getRight()==null)
n.setRight(new BinaryTreeNode(new String(s)));
else
addToSubTree(n.getRight(), s);
}
}
}
```

D1: removeTile

```
public Tile removeTile()
```

```
{
Tile r1;
int randnumber;
if(total==0)
return null;
randnumber = (int) Math.round(Math.random()*1000);
r1=tiles.takeValue(randnumber%total);
total=total-1;
return r1;
}
```

D2: removeTile

```
public Tile removeTile()
{
Tile rt;
int randno;
randno = (int) Math.round(Math.random()*109);
rt=tiles.takeValue(randno-t);
t=t+1;
total=total--;
return rt;
}
```

E1: removeTile

```
public Tile removeTile()
{
int randomMyTile = 0;
randomMyTile = (int) Math.floor(Math.random()* NoofTilesInBag);
Tile tileToReturn = tilesInBag [aRandomTile];
for (int i = aRandomTile;
i < NoofTilesInBag; i++)
{
tilesInBag [i] = tilesInBag [i+1];
}
tilesInBag [NoofTilesInBag-1] = null;
NoofTilesInBag--;
```

```
return tileToReturn;
}
```

E2: removeTile

```
public Tile removeTile()
{
int randomTile = 0;
randomTile = (int) Math.floor(Math.random()*numberOfTiles);
Tile tileToReturn = tilesInBag [randomTile];
for (int i = randomTile;
i < numberOfTiles; i++)
{
tilesInBag [i] = tilesInBag [i+1];
}
tilesInBag [numberOfTiles-1] = null;
numberOfTiles--;
return tileToReturn;
}
```

4.5 Pre-Processing the Corpus

Prior to applying LSA to a text collection, the task of pre-processing the corpus is performed such that only words remain in the data. The following is a list of possible pre-processing parameters specific to source-code.

- removing comments,
- removing Java reserved terms,
- merging/separating terms containing multiple words, and

- removing obfuscation parameters found within terms, e.g. student_name, _ is an obfuscation parameter).

Some possible pre-processing parameters not specific to source-code files are:

- removing terms occurring in one file,
- removing terms occurring in all files,
- removing terms solely composed of numeric characters,
- removing syntactical tokens (e.g. semi-colons, colons),
- removing terms consisting of a single letter, and
- converting upper case letters to lower case.

For this experiment, a MATLAB Toolbox for Generating Term-Document Matrices from Text Collections, namely the TMG tool, [147] was used to conduct the task of pre-processing the files. During pre-processing, terms solely composed of numeric characters, and syntactical tokens (e.g. semi-colons, colons) were removed from the source-code files. Terms consisting of a single letter, and terms that occurred in one file were also discarded because such terms hold no information about relationships across files. Source-code identifiers containing multiple words were treated as a single identifier, i.e. obfuscators that joined two words together were removed such that the two words were treated as a single word. This experiment is not concerned with investigating the impact of various pre-

processing parameter settings on the performance of LSA, this is done in Chapters 5 and 6.

Here is an example of what the A1-locate source-code fragment looks like after applying pre-processing:

```
private int locate object thing int cnumber int
arrindex for int spelling size
cnumber cnumber if spelling elementat equals thing
arrindex break return arrindex
```

4.6 Creating the Term-by-File Matrix

After pre-processing the files, a 44×10 term-by-file matrix is created, as shown in Table 4.2. The elements of this matrix contain the frequency in which a term occurs in a file, for example, in file A1, the first column of matrix A, the term *break* occurs once, and the term *int* occurs four times.

4.7 Applying term weighting

This Section is concerned with applying a term weighting scheme [38] to the term-by-file matrix A. term weighting is optional and its purpose is to increase or decrease the importance of terms using local and global weights in order to improve retrieval performance.

Table 4.2: Term-by-file matrix A

	A1	A2	B1	B2	C1	C2	D1	D2	E1	E2
addtosubtree	0	0	0	0	3	3	0	0	0	0
addword	0	0	1	1	0	0	0	0	0	0
binarytreenode	0	0	0	0	3	3	0	0	0	0
boolean	0	0	1	1	0	0	0	0	0	0
break	1	1	0	0	0	0	0	0	0	0
checkword	0	0	1	1	0	0	0	0	0	0
compareto	0	0	0	0	1	1	0	0	0	0
elementat	1	1	0	0	0	0	0	0	0	0
else	0	0	0	0	3	3	0	0	0	0
equals	1	1	0	0	0	0	0	0	0	0
equalsignorecase	0	0	1	1	0	0	0	0	0	0
false	0	0	1	1	0	0	0	0	0	0
floor	0	0	0	0	0	0	0	0	1	1
for	1	1	1	1	0	0	0	0	1	1
getleft	0	0	0	0	2	2	0	0	0	0
getright	0	0	0	0	2	2	0	0	0	0
getvalue	0	0	0	0	1	1	0	0	0	0
if	1	1	1	1	3	4	1	0	0	0
int	4	4	1	1	0	0	2	2	3	3
math	0	0	0	0	0	0	2	2	2	2
new	0	0	0	0	2	4	0	0	0	0
null	0	0	0	0	2	3	1	0	1	1
object	1	1	0	0	0	0	0	0	0	0
private	1	1	0	0	1	1	0	0	0	0
public	0	0	2	2	0	0	1	1	1	1
random	0	0	0	0	0	0	1	1	1	1
removetile	0	0	0	0	0	0	1	1	1	1
return	1	1	2	2	0	0	2	1	1	1
round	0	0	0	0	0	0	1	1	0	0
setleft	0	0	0	0	1	1	0	0	0	0
setright	0	0	0	0	1	1	0	0	0	0
size	1	1	0	0	0	0	0	0	0	0
store	0	0	2	2	0	0	0	0	0	0
string	0	0	2	2	1	4	0	0	0	0
takevalue	0	0	0	0	0	0	1	1	0	0
thing	2	2	0	0	0	0	0	0	0	0
tile	0	0	0	0	0	0	2	2	2	2
tiles	0	0	0	0	0	0	1	1	0	0
tilesinbag	0	0	0	0	0	0	0	0	4	4
tiletoreturn	0	0	0	0	0	0	0	0	2	2
total	0	0	0	0	0	0	4	2	0	0
true	0	0	1	1	0	0	0	0	0	0
void	0	0	1	1	1	1	0	0	0	0
word	0	0	4	4	0	0	0	0	0	0

Local weights determine the value of a term in a particular file, and *global weights* determine the value of a term in the entire file collection. Various local and global weighting schemes exist [14] and these are applied to the term-by-file matrix to give high weights to important terms, i.e. those that occur distinctively across files, and low weights to terms that appear frequently in the file collection.

Long files have a larger number of terms and term frequencies than short files and this increases the term matches between a query and a long file, thus increasing the retrieval chances of longer files over small ones. When applying *document length normalization* [128] the term values are adjusted depending on the length of each file in the collection. The value of a term in a file is $l_{i,j} \times g_i \times n_j$, where $l_{i,j}$ is the local weighting for term i in file j, g_i is the global weighting for term i, and n_j is the document-length normalization factor [14].

Tables 4.3, 4.4, and 4.5 contain some of the most commonly used term weighting formulas [14]. Symbol f_{ij} defines the number of times (term-frequency) term *i* appears in file *j*; let

$$b(f_{ij}) = \begin{cases} 1, & \text{if } f_{ij} > 0, \\ 0, & \text{if } f_{ij} = 0, \end{cases}$$

$$p_{ij} = \frac{f_{ij}}{\sum_j f_{ij}}$$

When discussing our findings in this thesis, we will use a three-letter string to represent

Table 4.3: Formulas for local term-weights (l_{ij})

Symbol	Name	Formula
b	Binary	$b(f_{ij})$
l	Logarithmic	$log_2(1 + f_{ij})$
n	Augmented normalised term frequency	$(b(f_{ij}) + (f_{ij}/max_k f_{kj}))/2$
t	Term frequency	f_{ij}
a	Alternate log	$b(f_{ij})(1 + log_2 f_{ij})$

Table 4.4: Formulas for global term-weights (g_i)

Symbol	Name	Formula
x	None	1
e	Entropy	$1 + (\sum_j (p_{ij} log_2(p_{ij}))/log_2 n)$
f	Inverse document frequency (IDF)	$log_2(n/\sum_j b(f_{ij}))$
g	GfIdf	$(\sum_j f_{ij})/(\sum_j b(f_{ij}))$
n	Normal	$1/\sqrt{\sum_j f_{ij}^2}$
p	Probabilistic inverse	$log_2((n - \sum_j b(f_{ij}))/\sum_j b(f_{ij}))$

each term weighting scheme with the particular local, global and normalization factor combinations. For example, the *tec* weighting scheme uses the *term frequency* (t) local term weighting, the *entropy* (e) global term weighting, and the *cosine document normalization factor* (c). In this tutorial the *tnc* weighting scheme will be applied.

4.7.1 Applying term weighting to matrix A

Table 4.6 shows the weighted matrix B. Comparing the term-by-file matrix A (Table 4.2) which holds the original term values, with term-by-file matrix B (Table 4.6) which holds the weights of terms after applying the tnc term weighting, show that term values have been adjusted.

Table 4.5: Formulas for document length normalisation (n_j)

Symbol	Name	Formula
x	None	1
c	Cosine	$(\sum_i (g_i l_{ij})^2)^{-1/2}$

4.8 Singular Value Decomposition

After creating matrix B, SVD and dimensionality reduction are applied to the matrix, as discussed in Section 4.3. For our corpus we will set the value of *k* to 4 dimensions. Computing the SVD of matrix B using four dimensions returns a 44×4 term-by-dimension matrix U_4, a 4×4 singular values matrix, Σ_4, and a 10×4 file-by-dimension matrix V_4. The rank-*k* approximation to matrix B, can be constructed through $B_4 = U_4 \Sigma_4 V_4^T$. The term-by-file matrix B_4 is shown in Table 4.7.

4.9 Impact of LSA on Relations between Terms and Files

There are three types of comparisons that can be made from the SVD model [32]:

- those comparing a term and a file (i.e. how associated are term *i* and file *j*),
- those comparing two terms (i.e. how similar are terms *i* and *j*), and
- those comparing two files (i.e. how similar are files *i* and *j*).

Table 4.6: Term-by-file matrix B derived after applying term weighting to matrix A

	A1	A2	B1	B2	C1	C2	D1	D2	E1	E2
addtosubtree	0.00	0.00	0.00	0.00	0.29	0.26	0.00	0.00	0.00	0.00
addword	0.00	0.00	0.31	0.31	0.00	0.00	0.00	0.00	0.00	0.00
binarytreenode	0.00	0.00	0.00	0.00	0.29	0.26	0.00	0.00	0.00	0.00
boolean	0.00	0.00	0.31	0.31	0.00	0.00	0.00	0.00	0.00	0.00
break	0.36	0.36	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
checkword	0.00	0.00	0.31	0.31	0.00	0.00	0.00	0.00	0.00	0.00
compareto	0.00	0.00	0.00	0.00	0.29	0.26	0.00	0.00	0.00	0.00
elementat	0.36	0.36	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
else	0.00	0.00	0.00	0.00	0.29	0.26	0.00	0.00	0.00	0.00
equals	0.36	0.36	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
equalsignorecase	0.00	0.00	0.31	0.31	0.00	0.00	0.00	0.00	0.00	0.00
false	0.00	0.00	0.31	0.31	0.00	0.00	0.00	0.00	0.00	0.00
floor	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.41	0.41
for	0.21	0.21	0.18	0.18	0.00	0.00	0.00	0.00	0.23	0.23
getleft	0.00	0.00	0.00	0.00	0.29	0.26	0.00	0.00	0.00	0.00
getright	0.00	0.00	0.00	0.00	0.29	0.26	0.00	0.00	0.00	0.00
getvalue	0.00	0.00	0.00	0.00	0.29	0.26	0.00	0.00	0.00	0.00
if	0.09	0.09	0.08	0.08	0.23	0.27	0.09	0.00	0.00	0.00
int	0.27	0.27	0.06	0.06	0.00	0.00	0.13	0.15	0.22	0.22
math	0.00	0.00	0.00	0.00	0.00	0.00	0.26	0.29	0.29	0.29
new	0.00	0.00	0.00	0.00	0.19	0.33	0.00	0.00	0.00	0.00
null	0.00	0.00	0.00	0.00	0.21	0.27	0.13	0.00	0.14	0.14
object	0.36	0.36	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
private	0.26	0.26	0.00	0.00	0.21	0.18	0.00	0.00	0.00	0.00
public	0.00	0.00	0.25	0.25	0.00	0.00	0.15	0.17	0.17	0.17
random	0.00	0.00	0.00	0.00	0.00	0.00	0.26	0.29	0.29	0.29
removetile	0.00	0.00	0.00	0.00	0.00	0.00	0.26	0.29	0.29	0.29
return	0.12	0.12	0.21	0.21	0.00	0.00	0.25	0.14	0.14	0.14
round	0.00	0.00	0.00	0.00	0.00	0.00	0.36	0.41	0.00	0.00
setleft	0.00	0.00	0.00	0.00	0.29	0.26	0.00	0.00	0.00	0.00
setright	0.00	0.00	0.00	0.00	0.29	0.26	0.00	0.00	0.00	0.00
size	0.36	0.36	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
store	0.00	0.00	0.31	0.31	0.00	0.00	0.00	0.00	0.00	0.00
string	0.00	0.00	0.18	0.18	0.08	0.29	0.00	0.00	0.00	0.00
takevalue	0.00	0.00	0.00	0.00	0.00	0.00	0.36	0.41	0.00	0.00
thing	0.36	0.36	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
tile	0.00	0.00	0.00	0.00	0.00	0.00	0.26	0.29	0.29	0.29
tiles	0.00	0.00	0.00	0.00	0.00	0.00	0.36	0.41	0.00	0.00
tilesinbag	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.41	0.41
tiletoreturn	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.41	0.41
total	0.00	0.00	0.00	0.00	0.00	0.00	0.46	0.26	0.00	0.00
true	0.00	0.00	0.31	0.31	0.00	0.00	0.00	0.00	0.00	0.00
void	0.00	0.00	0.22	0.22	0.21	0.18	0.00	0.00	0.00	0.00
word	0.00	0.00	0.31	0.31	0.00	0.00	0.00	0.00	0.00	0.00

Table 4.7: Term-by-file matrix B_4 showing the term values after applying LSA to the weighted matrix B

	A1	A2	B1	B2	C1	C2	D1	D2	E1	E2
addtosubtree	0.00	0.00	0.00	0.00	0.28	0.28	0.01	-0.01	0.00	0.00
addword	0.00	0.00	0.31	0.31	-0.01	0.01	-0.01	-0.01	0.01	0.01
binarytreenode	0.00	0.00	0.00	0.00	0.28	0.28	0.01	-0.01	0.00	0.00
boolean	0.00	0.00	0.31	0.31	-0.01	0.01	-0.01	-0.01	0.01	0.01
break	0.36	0.36	0.00	0.00	0.00	0.00	-0.02	-0.03	0.02	0.02
checkword	0.00	0.00	0.31	0.31	-0.01	0.01	-0.01	-0.01	0.01	0.01
compareto	0.00	0.00	0.00	0.00	0.28	0.28	0.01	-0.01	0.00	0.00
elementat	0.36	0.36	0.00	0.00	0.00	0.00	-0.02	-0.03	0.02	0.02
else	0.00	0.00	0.00	0.00	0.28	0.28	0.01	-0.01	0.00	0.00
equals	0.36	0.36	0.00	0.00	0.00	0.00	-0.02	-0.03	0.02	0.02
equalsignorecase	0.00	0.00	0.31	0.31	-0.01	0.01	-0.01	-0.01	0.01	0.01
false	0.00	0.00	0.31	0.31	-0.01	0.01	-0.01	-0.01	0.01	0.01
floor	0.03	0.03	0.01	0.01	0.00	0.00	0.20	0.20	0.20	0.20
for	0.22	0.22	0.18	0.18	0.00	0.00	0.10	0.10	0.14	0.14
getleft	0.00	0.00	0.00	0.00	0.28	0.28	0.01	-0.01	0.00	0.00
getright	0.00	0.00	0.00	0.00	0.28	0.28	0.01	-0.01	0.00	0.00
getvalue	0.00	0.00	0.00	0.00	0.28	0.28	0.01	-0.01	0.00	0.00
if	0.09	0.09	0.08	0.08	0.25	0.25	0.03	0.01	0.03	0.03
int	0.27	0.27	0.06	0.06	0.00	0.00	0.16	0.16	0.20	0.20
math	0.00	0.00	0.00	0.00	0.00	0.00	0.28	0.29	0.28	0.28
new	0.00	0.00	0.00	0.00	0.26	0.26	0.01	0.00	0.00	0.00
null	0.01	0.01	0.00	0.00	0.24	0.24	0.11	0.10	0.10	0.10
object	0.36	0.36	0.00	0.00	0.00	0.00	-0.02	-0.03	0.02	0.02
private	0.26	0.26	0.00	0.00	0.20	0.19	-0.01	-0.02	0.02	0.02
public	0.00	0.00	0.25	0.25	-0.01	0.00	0.16	0.16	0.17	0.17
random	0.00	0.00	0.00	0.00	0.00	0.00	0.28	0.29	0.28	0.28
removetile	0.00	0.00	0.00	0.00	0.00	0.00	0.28	0.29	0.28	0.28
return	0.12	0.12	0.21	0.21	0.00	0.01	0.16	0.15	0.18	0.18
round	-0.03	-0.03	-0.01	-0.01	0.00	0.00	0.20	0.20	0.19	0.19
setleft	0.00	0.00	0.00	0.00	0.28	0.28	0.01	-0.01	0.00	0.00
setright	0.00	0.00	0.00	0.00	0.28	0.28	0.01	-0.01	0.00	0.00
size	0.36	0.36	0.00	0.00	0.00	0.00	-0.02	-0.03	0.02	0.02
store	0.00	0.00	0.31	0.31	-0.01	0.01	-0.01	-0.01	0.01	0.01
string	0.00	0.00	0.18	0.18	0.18	0.19	0.00	-0.01	0.00	0.00
takevalue	-0.03	-0.03	-0.01	-0.01	0.00	0.00	0.20	0.20	0.19	0.19
thing	0.36	0.36	0.00	0.00	0.00	0.00	-0.02	-0.03	0.02	0.02
tile	0.00	0.00	0.00	0.00	0.00	0.00	0.28	0.29	0.28	0.28
tiles	-0.03	-0.03	-0.01	-0.01	0.00	0.00	0.20	0.20	0.19	0.19
tilesinbag	0.03	0.03	0.01	0.01	0.00	0.00	0.20	0.20	0.20	0.20
tiletoreturn	0.03	0.03	0.01	0.01	0.00	0.00	0.20	0.20	0.20	0.20
total	-0.02	-0.02	-0.01	-0.01	0.01	0.01	0.18	0.19	0.18	0.18
true	0.00	0.00	0.31	0.31	-0.01	0.01	-0.01	-0.01	0.01	0.01
void	0.00	0.00	0.22	0.22	0.19	0.20	0.00	-0.01	0.00	0.00
word	0.00	0.00	0.31	0.31	-0.01	0.01	-0.01	-0.01	0.01	0.01

4.9.1 Term-file relations

This Section illustrates the impact of LSA on the values of terms within files. File pairs D1-D2, E1-E2, have been selected for investigation, these four files are relatively more similar than the rest of the files in the collection, as they are all concerned with removing an object (i.e. a Tile) from an array. Table 4.8 shows a selection of terms and their values, before and after applying LSA to the file collection, and those values will be used in the discussion.

Table 4.8 has been split into four sets, each set separated by a horizontal line. The first set contains terms that only occur in files E1 and E2. The second set contains terms that only occur in files D1 and D2. The third set contains terms that occur in all four files D1, D2, E1, and E2. The terms in the first three sets occur only in the files mentioned in Table 4.8 and in no other files in the corpus. The fourth set contains terms that occur across files in the corpus without any particular pattern.

Comparing the first two sets of terms, after computing LSA each of the seven terms now have very close values in all four files. It is suspected that the importance of the terms in the first two sets has changed from zero to higher values mainly due to the transitive relations between those terms and terms in sets three and four. There are various degrees of transitive relations as discussed by Kontostathis and Pottenger [76]. Table 4.8 shows that the first three terms originally appeared in files E1 and E2 in matrix A. These terms did not appear in D1 or D2, but because files D1 and E1 have other terms in common, the zero value in each of the terms in matrix B_4 has been replaced by values above zero in matrix

Table 4.8: Extracts of term-by-file matrices A, B, and B_4

	A				B				B_4			
	D1	D2	E1	E2	D1	D2	E1	E2	D1	D2	E1	E2
floor	0	0	1	1	0.00	0.00	0.41	0.41	0.20	0.20	0.20	0.20
tilesinbag	0	0	4	4	0.00	0.00	0.41	0.41	0.20	0.20	0.20	0.20
tiletoreturn	0	0	2	2	0.00	0.00	0.41	0.41	0.20	0.20	0.20	0.20
round	1	1	0	0	0.36	0.41	0.00	0.00	0.20	0.20	0.19	0.19
takevalue	1	1	0	0	0.36	0.41	0.00	0.00	0.20	0.20	0.19	0.19
tiles	1	1	0	0	0.36	0.41	0.00	0.00	0.20	0.20	0.19	0.19
total	4	2	0	0	0.46	0.26	0.00	0.00	0.18	0.19	0.18	0.18
math	2	2	2	2	0.26	0.29	0.29	0.29	0.28	0.29	0.28	0.28
random	1	1	1	1	0.26	0.29	0.29	0.29	0.28	0.29	0.28	0.28
removetile	1	1	1	1	0.26	0.29	0.29	0.29	0.28	0.29	0.28	0.28
tile	2	2	2	2	0.26	0.29	0.29	0.29	0.28	0.29	0.28	0.28
if	1	0	0	0	0.09	0.00	0.00	0.00	0.03	0.01	0.03	0.03
int	2	2	3	3	0.13	0.15	0.22	0.22	0.16	0.16	0.20	0.20
null	1	0	1	1	0.13	0.00	0.14	0.14	0.11	0.10	0.10	0.10
public	1	1	1	1	0.15	0.17	0.17	0.17	0.16	0.16	0.17	0.17
return	2	1	1	1	0.25	0.14	0.14	0.14	0.16	0.15	0.18	0.18
for	0	0	1	1	0.00	0.00	0.23	0.23	0.10	0.10	0.14	0.14

B_4. Similarly, terms in the second set occurred only in files D1 and D2, now have a value in files E1 and E2, and these values are very close to the ones in files E1 and E2.

Furthermore, terms in the fourth set, received a mixture of relatively low and high values depending on their pattern of occurrences in matrix B. For example, the value of the last term, *for*, in file E1 in Table 4.8 has been reduced from 0.23 to 0.14 which means that LSA has reduced the importance of term *for* in file E1.

Although the choice of dimensions and weighting scheme can have an impact on the LSA similarity values, it is not the purpose of this Chapter to investigate how these parameters influence LSA performance – the impact of choice of dimensionality and weighting scheme on LSA performance is discussed in Chapter 6.

4.9.2 Term-term relations

The term-by-term matrices corresponding to matrices A and B hold the similarity values between terms. A term-by-term matrix can be generated by computing XX^T where X is a term-by-file matrix. This computes the dot product between two row vectors of the term-by-file matrix, and reflects the extent to which two terms have a similar pattern of occurrences across the set of files [32]. Therefore, it can be assumed that each cell a_{ij} of the term-by-term matrix represents the similarity between a term i and a term j.

After applying SVD and reducing dimensionality to k dimensions the term-by-term matrix can be recomputed by using the U_k and S_k matrices through, $(U_k\Sigma_k)(U_k\Sigma_k)^T$ [32].

In the discussion that follows,

- TT_A denotes the term-by-term matrix corresponding to matrix A computed by AA^T
- TT_B denotes the term-by-term matrix corresponding to matrix B computed by BB^T, and
- TT_C denotes the term-by-term matrix corresponding to matrix B_4 computed by $(U_k\Sigma_k)(U_k\Sigma_k)^T$.

The term-by-term matrices are too large to show here, and therefore only an extract of them is shown in Tables 4.9, 4.10, and 4.11. These Tables show that, before computing LSA, some terms never co-occurred together, and after computing LSA, those terms have a

Table 4.9: Extract of the term-by-term matrix TT_A, showing the term-term similarity values.

	floor	for	int	math	public	random	return	tiles	tilesinbag	total
floor	x									
for	2	x								
int	6	16	x							
math	4	4	20	x						
public	2	6	14	8	x					
random	2	2	10	8	4	x				
return	2	8	24	10	13	5	x			
tiles	0	0	4	4	2	2	3	x		
tilesinbag	8	8	24	16	8	8	8	0	x	
total	0	0	12	12	6	6	10	6	0	x

similarity value.

Table 4.9, shows that the term *total* never occurs in the same document with either *floor* or *tilesinbag*, hence their co-occurrences were zero. After applying term weighting and LSA, Table 4.11 shows that the values between term pair *total* and *floor*, and the term pair *total* and *tilesinbag* now both have a similarity value of 0.15. Although these terms never occur together in the same files, they co-occur with one or more of the same terms. The relationship between these terms comes from the relation: *total* co-occurs with *math*, and *math* co-occurs with *floor*. In mathematics this kind of relationship is referred to as a transitive relation.

4.9.3 File-file relations

The process of comparing two files is similar to that of comparing two terms described in Section 4.9.2, except that in this case it is the dot product between the two column vectors

Table 4.10: Extract of the term-by-term matrix TT_B, showing the term-term similarity values after applying the tnc weighting scheme.

	floor	for	int	math	public	random	return	tiles	tilesinbag	total
floor	x									
for	0.19	x								
int	0.18	0.24	x							
math	0.23	0.14	0.21	x						
public	0.14	0.17	0.15	0.18	x					
random	0.23	0.14	0.21	0.32	0.18	x				
return	0.11	0.19	0.21	0.19	0.22	0.19	x			
tiles	0.00	0.00	0.11	0.22	0.12	0.22	0.15	x		
tilesinbag	0.33	0.19	0.18	0.23	0.14	0.23	0.11	0.00	x	
total	0.00	0.00	0.10	0.20	0.11	0.20	0.15	0.28	0.00	x

Table 4.11: Extract of the term-by-term matrix TT_C, showing the term-term similarity values after applying the tnc weighting scheme and LSA.

	floor	for	int	math	public	random	return	tiles	tilesinbag	total
floor	x									
for	0.11	x								
int	0.16	0.23	x							
math	0.23	0.13	0.21	x						
public	0.14	0.17	0.15	0.18	x					
random	0.23	0.13	0.21	0.32	0.18	x				
return	0.15	0.21	0.21	0.19	0.22	0.19	x			
tiles	0.16	0.08	0.13	0.22	0.12	0.22	0.12	x		
tilesinbag	0.16	0.11	0.16	0.23	0.14	0.23	0.15	0.16	x	
total	0.15	0.07	0.12	0.20	0.11	0.20	0.11	0.14	0.15	x

of a term-by-file matrix which tells the extent to which two files have a similar profile of terms. Therefore, it can be assumed that each cell a_{ij} of the file-by-file matrix represents the similarity between a file *i* and a file *j*. The file-by-file matrix of matrix X can be generated by computing $X^T X$.

After SVD and dimensionality reduction to k dimensions the file-by-file matrix can be recomputed by $(V_k \Sigma_k)(V_k \Sigma_k)^T$.

Thus, the i, j cell of $X_k^T X_k$ can be obtained by taking the dot product between term i and j rows of the matrix $V_k \Sigma_k$.

In the discussion that follows,

- FF_A denotes the file-by-file matrix corresponding to matrix A computed by $A^T A$,
- FF_B denotes the file-by-file matrix corresponding to matrix B computed by $B^T B$, and
- FF_C denotes the file-by-file matrix corresponding to matrix B_4 computed by $(V_k \Sigma_k)(V_k \Sigma_k)^T$.

The effect of LSA on file-file relations will be examined by looking at the relationships (or similarity) between files before and after LSA has been applied to the matrix. Table 4.12 shows the file-file similarity values after applying LSA.

Table 4.13 shows the average file-file values between similar and non-similar file pairs before and after applying LSA. Average values of similar files is denoted by *ms*, and average

Table 4.12: File-by-file matrix FF_C

File-file similarity k=4										
	A1	A2	B1	B2	C1	C2	D1	D2	E1	E2
A1	x									
A2	0.99	x								
B1	0.08	0.08	x							
B2	0.08	0.08	1.00	x						
C1	0.08	0.08	0.08	0.08	x					
C2	0.07	0.07	0.11	0.11	0.98	x				
D1	0.04	0.04	0.09	0.09	0.06	0.06	x			
D2	0.02	0.02	0.07	0.07	0.00	0.01	0.70	x		
E1	0.16	0.16	0.14	0.14	0.02	0.03	0.69	0.70	x	
E2	0.16	0.16	0.14	0.14	0.02	0.03	0.69	0.70	0.71	x

Table 4.13: Average values of relevant and non-relevant files

	FF_A	FF_B	FF_C
average similar(ms)	0.71	0.64	0.99
average non-similar (mns)	-0.06	-0.26	-0.31
difference (ms-mns)	0.77	0.90	1.30

values of non-similar files is denoted by *mns*. The ms value was computed by calculating the average of all the similarity values of file pairs shown in Table 4.12 (i.e. A1:A2=0.99, B1:B2=1.00, C1:C2=0.98, D1:D2=0.70, E1:E2=0.71, D1:E1=0.69, D1:E2=0.69, D2:E1=0.70, D2:E2=0.70) and the mns value was computed by calculating the average of the similarity values of the rest of the pairs. *Difference*, is the difference between ms and mns values. The greater the difference the better the retrieval performance, i.e. greater separation between similar and non-similar files is desired. The better system is the one that returns the highest ms value, lowest mns value, and the largest separation value.

Table 4.13 shows that originally the ms and mns values are 0.71 and -0.06 respectively, and after applying the tnc term weighting scheme the ms and mns values are reduced to

0.64 and -0.26 respectively. These results show that the tnc term weighting scheme has lowered the similarity values between similar files, but has also increased the separation between similar and non-similar files. Applying LSA to the weighted matrix, has improved results, by giving higher values to similar files, lower values to non-similar files, and a good separation between them.

4.10 Performance Evaluation Measures

The performance of an information retrieval system is measured by its retrieval performance. Two standard and most frequently used measures in information retrieval system evaluations are *recall* and *precision*. In order to evaluate an information retrieval system, queries and their relevant files must be known beforehand.

Precision represents the proportion of retrieved files that are relevant. Precision is denoted by P,

$$P = \frac{F_r}{F_t} \tag{4.3}$$

where $P \in [0, 1]$. F_r is the number of relevant files retrieved and F_t is the total number of files retrieved for a given query. Precision is 1.00 when every relevant file returned in the ranked list is relevant to the query.

Recall represents the proportion of relevant files that are retrieved. Recall is 1.00 when every relevant file is retrieved. Recall, denoted by R, is defined by

$$R = \frac{F_r}{N_r} \tag{4.4}$$

where $R \in [0, 1]$. F_r is the total number of relevant files retrieved, and N_r is the total number of files relevant to the query.

The value of recall at the R-th position in the ranked list of retrieved files, where R equals to the number of relevant files (for example, if the number of relevant files to a query is 5 then the top 5 relevant files are retrieved), is equal to the value of R-Precision, which is the value of precision at the R-th position in the ranked list where R is equal to the total number of relevant files to a query. A geometric interpretation of this is provided by Aslam *et al.* [4].

Average Precision (AP) is the average of the precisions of the relevant files in the retrieved list. This evaluation measure produces a single value summary of the ranking positions of the relevant files retrieved. This is done by averaging the precision values obtained after each new relevant file is retrieved in the ranked list of files. The closer the AP value is to 1.00 the better the system's retrieval performance.

A commonly used evaluation measure for evaluating the performance of an algorithm using more than one query, is the *Mean Average Precision* (MAP). MAP is the average of the AP values of each query.

A *recall-precision graph* is an evaluation measure that combines recall and precision. The recall-precision graph shows the value of precision at various points of recall.

Expected Search Length (ESL) is the average of the rank positions of the relevant files in the returned list of files. The result of this gives the average number of files that must be examined in order to retrieve a given number of files. The smaller the value of ESL, the highest the position of the relevant files are in the ranked list and hence the better the performance of the system.

Other evaluation measures exist that take into consideration similarity values given to retrieved files. These include the *Lowest Positive Match* (LPM), *Highest False Match* (HFM) and *Separation* (Sep) [53]. Lowest Positive Match is the lowest score given to retrieved file, and Highest False Match is the highest score given to a non-relevant file in the returned list of files. Separation is the difference between the LPM and HFM. Overall system performance is calculated by taking the ratio of *Sep./HFM*, i.e. by dividing the separation by the HFM. The highest the ratio value, the better the system performance.

4.11 A Query Example

Suppose that we are interested in retrieving files concerning terms *addtosubtree*, *binarytreenode*, *setright*, and *search*. Since the term *search* is not an indexed term, it is omitted from the query leaving terms *addtosubtree*, *binarytreenode*, and *setright*. The query is initially treated as a file, and transformed into a vector. That is, the frequencies of the terms in the query (i.e. *addtosubtree*, *binarytreenode*, and *setright*) would be the non-zero elements in the query vector, where each element corresponds to an indexed term.

q=(1 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0)^T.

The same weighting scheme applied to matrix A (i.e the tnc term weighting scheme), is then applied to the query vector. After applying the tnc term weighting scheme the query becomes:

q=(0.24 0 0.24 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0.71 0 0 0 0 0 0 0 0 0 0 0 0 0)^T.

The query vector is then projected to the term-file space using Function 4.1. The coordinate values of the query vector in the *k*-dimensional space (where the value of *k* is set to 4) are: Q=(-0.03 0.14 0.08 0.18).

The next step involves computing the cosine similarity between the query vector and all other file vectors. The output is a list that contains the retrieved files sorted in descending order of similarity to the query, which positions the most similar files in the top of the list.

In Table 4.14 the column *File ID* shows the ID's of the retrieved files, for example, in Table 4.14 the file C1 is ranked first, as most relevant to the query, and the B2 file is ranked last, as least relevant to the query. The *Similarity value* column displays the cosine similarity values, each value holds the similarity between the query vector and a file vector. The *Relevant?* column shows human relevance judgment, that is whether the retrieved files are relevant to the query, where *R* denotes that the file is relevant and *NR* that is non-relevant.

Table 4.15 shows the evaluation results for the query when using threshold value *thr*, and cutoff point *n*. When the retrieved list of files is cutoff based on the threshold value

$thr > 0.50$, evaluation measures are computed using the files which received a similarity value above 0.50 (for example, files C1 and C2 in Table 4.14). The cutoff point r, cuts the retrieved list of files at r^th position in the ranked list. In this example we have set the value of r to 10, where 10 is the number of files in the corpus, and this means that all files in the corpus are retrieved and sorted in order of their similarity to the query. Using threshold r=10, evaluation is computed by taking into consideration the positions of all files in the corpus. When using all files in the corpus, recall will always reach 1.00, and hence it should not be used as it tells us nothing on system performance.

Table 4.14: Results for query q

Rank	File ID	Similarity value	Relevant?
1	C1	1.00	R
2	C2	0.99	R
3	D1	0.04	NR
4	D2	-0.03	NR
5	E2	-0.04	NR
6	E1	-0.04	NR
7	A1	-0.06	NR
8	A2	-0.06	NR
9	B1	-0.09	NR
10	B2	-0.09	NR

Table 4.15: Evaluation results for query q

	$thr > 0.50$	$r = 10$
Recall	1.00	1.00
Precision	1.00	0.20
Average-Precision	1.00	1.00
ESL	1.50	1.50
LPM	0.99	0.99
HFM	0.00	0.04
Sep.	0.99	0.95
Sep./HFM	0.00	21.76

4.11.1 Treating a file vector as a query

Here, we take four files and treat each one as a query, thus file A1 is treated as Q1, B1 as Q2, C1 as Q3, and D1 as Q4. The cosine between the four query vectors and all files in the corpus are computed and the output is four lists of files, one list for each query, ranked in descending order of similarity to the query. Tables 4.16, 4.17, 4.18, and 4.19 show the results of each query respectively. Table 4.20 shows the results of each query using the evaluation measures discussed in Section 4.10.

Table 4.16: Results for query Q1

Rank	File ID	Similarity value	Relevant?
1	A1	1.00	R
2	A2	1.00	R
3	E1	0.10	NR
4	E2	0.10	NR
5	C1	0.00	NR
6	C2	0.00	NR
7	B1	0.00	NR
8	B2	0.00	NR
9	D1	-0.08	NR
10	D2	-0.10	NR

Table 4.17: Results for query Q2

Rank	File ID	Similarity value	Relevant?
1	B1	1.00	R
2	B2	1.00	R
3	E1	0.04	NR
4	E2	0.04	NR
5	C2	0.02	NR
6	A2	0.00	NR
7	A1	0.00	NR
8	C1	-0.02	NR
9	D1	-0.03	NR
10	D2	-0.05	NR

Table 4.18: Results for query Q3

Rank	File ID	Similarity value	Relevant?
1	C1	1.00	R
2	C2	1.00	R
3	D1	0.05	NR
4	A1	0.00	NR
5	A2	0.00	NR
6	E2	-0.02	NR
7	E1	-0.02	NR
8	B1	-0.02	NR
9	B2	-0.02	NR
10	D2	-0.03	NR

Table 4.19: Results for query Q4

Rank	File ID	Similarity value	Relevant?
1	D1	1.00	R
2	D2	1.00	R
3	E2	0.98	R
4	E1	0.98	R
5	C2	0.05	NR
6	C1	0.05	NR
7	B1	-0.03	NR
8	B2	-0.03	NR
9	A2	-0.08	NR
10	A1	-0.08	NR

Table 4.20: Evaluation results for all queries

	Q1	Q2	Q3	Q4
File ID	1	3	5	7
No. of relevant files to the query	2	2	2	4
Recall	1	1	1	1
Precision	0.20	0.20	0.20	0.40
Average-Precision	1.00	1.00	1.00	1.00
ESL	1.50	1.50	1.50	2.50
LPM	1.00	1.00	1.00	0.98
HFM	0.10	0.04	0.05	0.05
Separation	0.90	0.96	0.95	0.93
Sep./HFM	9.46	23.33	19.37	17.38

4.12 Conclusion

In this Chapter we described the Vector Space Model, and provided a theoretical and practical introduction to the Latent Semantic Analysis information retrieval technique. Using a small source-code corpus, we described a step-by-step process of applying LSA to the corpus, and computing the similarity between source-code files.

We have used a small corpus to demonstrate LSA, and we are aware that LSA must have a sufficient number of files and terms to be able to effectively generate knowledge about the meaning of words and files, and to be able to sufficiently identify groups of files that are relatively more similar than others.

Many open questions exist surrounding the theoretical and mathematical understandings of LSA, and describing exactly why LSA works still remains a challenge by Information retrieval researchers.

Chapters 5 and 6 explore the influence of parameter choice on LSA performance with regards to detecting similar source-code files on larger source-code corpora.

Chapter 5

Relevant Literature on LSA and Parameters

This Chapter discusses relevant literature with the main focus on the parameters that influence the performance of LSA. We begin the discussion by investigating parameter settings researchers have used and recommended for effective application of LSA to natural language information retrieval tasks. We then focus the discussion on parameter choices made by researchers for tasks specific to source-code. Finally, we describe some plagiarism detection tools whose functionality is based on information retrieval algorithms.

5.1 Parameter Settings for General LSA Applications

Although LSA is a well studied technique that has had many applications, there are some issues surrounding the choice of parameters that influence its performance. A review of the recent literature shows that various researchers are using various different parameter settings for achieving best LSA performance. The main parameters influencing the performance of LSA can be categorised into the following groups:

- Document pre-processing,
- Term-weighting algorithms,
- Choice of dimensionality, and
- Choice of similarity measure.

Document pre-processing operations include the following [5].

- Lexical analysis of text: this involves the identification of words in the text. This step involves identifying the spaces in the text as word separators, and consideration of digits, hyphens, punctuation marks, and the case of letters.
- Stopword elimination: this involves the elimination of words with a high frequency in the document corpus. This involves removing prepositions, conjunctions and common words that could be considered as useless for purposes of retrieval, e.g. words such as *the*, *and*, and *but*, found in the English language.

- Stemming of words: this involves transforming variants of words with the same root into a common concept. A stem is the portion of the remaining word after removing its affixes (i.e. suffixes and prefixes). An example of a stem is the word *eliminat* which is the stem of the variants *eliminated*, *eliminating*, *elimination*, and *eliminations*. In natural language texts, stemming is considered to improve retrieval performance.

- Selection of index terms: this involves selecting from the index the words to be used as index terms. Some approaches to this involve selecting only the noun terms in the text and thus eliminating verbs, adjectives, adverbs, connectives, articles and pronouns.

- Construction of thesauri: this operation involves constructing a set of important words in a subject domain, and for each of those words creating a set of related words. The main purpose of a thesaurus is to assist users with selecting searching terms.

Much research has been done on the impact of various parameters on the retrieval performance of the standard VSM. With regards to the corpus size, it is well argued that more reliable results are gained from a larger corpus size [111, 117]. Rehder *et al.* [117] investigated the efficacy of LSA as a technique for evaluating the quality of student responses against human ratings. They found that for 106 student essays, the performance of LSA improved when files contained between 70-200 words.

Landauer [84] found that the scores assigned by LSA to essays (i.e. a target essay and a short text) were closely related to human judgments as were the human judgments to each other. Landauer [81] stresses that the ability of LSA to produce similarity estimates close to human judgments is not a matter of word frequencies, term weighting, co-occurrence counts, or correlations in usage, but depends on a mathematical algorithm which has the power to infer deeper relations of meaning in text. According to Landauer [84], the performance of LSA depends on

> "the derivation of the semantic space of optimal dimensionality. The fact that LSA can capture as much of meaning as it does without using word order shows that the mere combination of words in passages constrains overall meaning very strongly."

Landauer *et al.* [83] recommend the use of 300 dimensions, Wild *et al.* [143] suggest that 10 dimensions are a good starting point, Dumais [38] found that 100 dimensions worked well for their test collections. The optimal dimensions selected by Kontostathis for 7 large corpora containing between 1,033 and 348,577 documents ranged from 75 to 500 depending on the corpus [75]. Chen *et al.* [24] implemented an LSI search engine and for a collection of 600,000 documents they used 400 dimensions. Berry *et al.*, [16] discuss the algebra behind LSA and demonstrate the SVD algorithm using a small corpus of 20 small documents (each consisting of a book title and the index terms within each book title ranged between 1 and 5) and a total of 16 index terms. They demonstrated the results when using 2,

4, and 6 dimensions and found that 2 dimensions were sufficient for retrieving the relevant files of a given query. Using a test database containing medical abstracts, Deerwester *et al.* [32] found that the performance of LSA can improve considerably after 10 or 20 dimensions, reaches a peak between 70 and 100 dimensions and then starts to slowly diminish. Jessup and Matrin [58] also found that for their datasets a choice of factors ranged from 100 to 300 factors, and Berry [13] suggests keeping at least 100 to 200 factors.

When SVD is applied without any dimensionality reduction, the performance of LSA is very close to that of the SVD. Without dimensionality reduction the number of factors k is set to r, where r is the rank of the term-by-document matrix, then the term-by-document matrix A_k will exactly reconstruct the original term-by-document matrix A [16]. Kontostathis [75] compared the performance of LSA with that of the traditional VSM on four large collections and found that the results of LSI when k=r were equivalent to the results of the standard VSM. Similar findings were reported by Pincombe [112] who found that, for a corpus of 50 documents, there was major improvement in LSA performance when the number of factors was increased from 10 to 20, and that optimal LSA performance was achieved when no dimensionality reduction was applied, i.e. k=r. Dumais applied LSA on 9 corpora where the number of documents in each corpus ranged between 20,108 and 56,920 documents [39].

Amongst the first researchers to compare the performance of LSA to that of the VSM were Lochbaum and Streeter [87]. They found that compared to the VSM, LSA had the advantage of being able to detect relevant documents when the terminology used in the query

did not match that of the document. They also reported that when the number of singular values is equal to the rank of the raw term-by-document matrix, the results returned by LSA and VSM are the same [87]. Jessup and Martin [58] found that LSA always matched or slightly outperformed the performance of the VSM across a wide range of collections.

Choice of term weighting can influence the performance of LSA and Dumais found that results improved (measured by AP) considerably when applying the local *log* and global *entropy* weighting schemes. Dumais concluded that global weights *IDF* and *entropy* increased system performance by 27% and 30% respectively, and the combination of the local weighting scheme *log* and global weighting scheme *entropy* improved performance by 40% when compared to the raw weighting scheme (i.e. no weighting)[38]. Dumais also found that, on average, the *normal* and *GfIdf* weighting algorithms had a negative impact on performance, i.e. applying those weightings decreased performance by 11% and 7% respectively when compared with raw term frequency. However, it was also found that applying the *normal* and *GfIdf* global weightings improved retrieval performance by as much as 7% and 6% respectively. Their results suggest that the corpus also influences the performance of weighting schemes [38].

The findings by Dumais [38] were similar to those reported by Harman [52]. In early literature, Harman [52] investigated the impact of term weighting for the task of query retrieval using the VSM. They found that using term-weighting improved retrieval results significantly when compared to *no term weighting* - they reported a 44% increase in average precision. Harman found that the *IDF* and *entropy* weightings produced large performance

improvements over the *raw* term weighting (i.e. raw term weighting) – *IDF* 20%, *entropy* 24%, *log-entropy* 35%, *log-entropy* combined with a document length normalisation algorithm 44%.

Other literature reports findings of system improvement in retrieval performance (by means of precision and recall) when a weighting scheme is applied to a corpus using the VSM [88, 60].

Nakov *et al.* [107] experimented with combinations of weighting algorithms considered by Dumais [38] and Jones [60] in order to evaluate their impact on LSA performance. They found that local and global weight functions are independent of each other and their performance (measured by average precision) is dependent on the corpus they are applied on. They found that for some corpora of text using the logarithm local weighting instead of raw term weighting resulted in higher precision, and on others resulted in consistently lower precision. With regards to global weighting functions, they found that applying global weighting functions *none*, *normal*, and *GfIdf*, resulted in lower precision regardless of the corpus text and local weighting function applied and the global weight *entropy* outperformed all other global weighting functions. The choice of weighting functions is dependent on the purpose the LSA application serves [107].

Sudarsun *et al.* [132] evaluated IDF+NDV which is the IDF global weight combined with Normalized Document Vector (NDV) weighting, and IWF+NDV which is the Inverse Word Frequency (IWF) weighting combined with NDV. They studied the impact of weighting functions when the corpus size increases and found that the IDF+NDV weighting func-

tion outperformed all other weighting functions they had experimented with. IDF+NDV produced 99% accuracy for a corpus of 100,000 documents, whereas IDF produced 98% accuracy.

Pincomber [112] compared three local and three global weighting schemes for correlation to a set of files containing document pairs judged as similar by humans. They found that global weighting had a greater effect on correlations than local weighting. The global weighting function *entropy* performed better than the *normal* weighting and both were shown to perform significantly better than the *IDF* global weight function. Their findings also show that use of a stop word list and adding background documents during the construction of the LSA space significantly improves performance. They found that choice of parameters has a major impact on LSA performance – the correlations between LSA and human judgments of pairwise document similarity varied between 0.60 and -0.01. The results are consistent with the literature where the *log-entropy* weighting scheme performed better in information retrieval [38] and text categorisation [107].

With regards the weighting scheme, the results by Wild *et al.* [143] were quite different to those by Pincomber [112] and the rest of the literature discussed. They found that the IDF global weighting outperformed all other weighting functions, and no clear indication as to which local weighting function performed best. They also found that combining stemming with stop-word filtering resulted in reduced average correlations with the human scores. The findings of Wild *et al.* [143], who also investigated the correlations between LSA and human scores, were consistent with those by Pincomber [112] who found that filtering stop

words using a stop-word list improves results.

Wild *et al.* [143] also experimented with three similarity measures: Pearson-correlation, Spearman's rho and Cosine. They found that best results were returned using when the Spearman's rho correlation was applied to their corpus. The most commonly used measure for comparing LSA vectors that has been shown for giving good results for information retrieval tasks is the cosine similarity measure [80].

The algebra behind the LSA algorithm was first described by Deerwester *et al.* [32] and is also further explained by Berry *et al.* [16, 15]. Those authors describe the geometry of the SVD process and the resulting matrices. They demonstrate the process of applying SVD, and show that the optimal rank-k approximation of a term-by-document matrix can be obtained by applying SVD and truncating the resulting matrices into k dimensions.

The power of LSA lies behind the SVD algorithm but also LSA gets its power from a variety of sources such as the corpus, term weighting, the cosine distance measure [141, 84].

The issue on the optimal number of factors to retain in order to best capture the semantic structure of the document collection still remains an unanswered question. The literature suggests that the choice of dimensions depends on factors such as the corpus itself, corpus size, choice of factors, the weighting scheme and pre-processing settings. In addition, these parameters must be customised for the purpose of the LSA application (e.g. whether it is for essay grading, similarity detection, or file searching by user queries).

A more critical question is that of "How can the optimal rank be chosen?" [58], in other words what is the optimal number of factors (also referred to as singular values, or dimensions) to retain and how can this number be chosen?

In the literature relevant to LSA and information retrieval, the number of factors is most frequently selected through experimentation. Researchers have investigated at techniques for automating the process of dimensionality reduction, Kakkonen *et al.* [66, 68], investigated statistical techniques and proposed the use of statistical methods (i.e. validation methods) as a mean for automating the dimensionality reduction process in LSA.

Most literature on understanding of LSA for information retrieval is through empirical testing. A theoretical and mathematical understanding of LSA that is beyond empirical evidences is important in order to gain a better understanding of LSA [36, 76]. Researchers have proposed many different theoretical and mathematical approaches to understanding LSA. Theoretical approaches include a subspace-model [148], using the multiple regression statistical technique and Bayesian methods in an attempt to explain the functionality behind LSI [131], and a probability model for LSI using the cosine similarity measure [36].

Some researchers have investigated the SVD algorithm in order to provide a better understanding of LSA. Schütze studied the values produced by the SVD algorithm and the meaning captured at various dimensions [124]. They found that all dimensions were potentially useful because different dimensions were important for different semantic distinctions. Their findings suggest that choice of dimensionality is also dependent on the purpose of the LSA application. Kakkonen *et al.* [66] investigated statistical techniques automat-

ing the process of dimensionality reduction and found that choice of dimensions is corpus dependent.

Kontostathis and Pottenger [76] examined the values produced by LSA and found that high-order co-occurrences are important in the effectiveness of LSA. They provide mathematical evidence that term co-occurrence is very important in LSA. They demonstrate that LSA gives lower similarity values to terms that co-occur frequently with many other words (reduces 'noise') and gives higher values to important term-term relations. They have examined the values between terms that have a first order co-occurrence up to fifth order co-occurrence using the original term-by-document matrices and truncated (reduced dimensionality) term-by-document matrices.

Kontostathis and Pottenger considered the average number of paths for the term pairs classified in the five different types of order-co-occurrence [76]. They found that the first order co-occurrence term pairs with the higher average number of paths tend to receive negative similarity values, pairs with fewer co-occurrence paths tend to receive lower similarity values, and pairs with a moderate number of co-occurrence paths tend to receive high similarity values. It is from this observation that they claim that LSA gives lower similarity values to terms that co-occur frequently with many other words (reduces 'noise') and gives higher values to important term-term relations. They also claim that terms with a second order co-occurrence and a high number of connectivity paths tend to receive high similarity values, while those terms with a second order co-occurrence and a moderate number of connectivity paths tend to receive negative similarity values. They claim that terms with

second order co-occurrence can be considered as the 'latent semantics' that are emphasized by LSA.

However, Landauer *et al.* [84] argues that:

> "LSA's ability [to generate knowledge from texts without the use of word order or syntax] is not a simple matter of counting and weighting words or co-occurrences, but depends on its derivation of a semantic space of optimal dimensionality from the mutual constraints reflected in the use of many words in many contexts.".

Perfetti discusses co-occurrence in language research and investigates the limitations of co-occurrence based systems [111].

The literature proposes several approaches for determining the intrinsic dimensionality of a dataset. Intrinsic (or effective) dimensionality is the number of dimensions needed to model the data without losing important information. Although the literature suggests many techniques for selecting dimensionality, there are no set dimensionality reduction methods or the number of dimensions that should be chosen, because the number of dimensions selected is likely to depend on the dataset in question.

Techniques for aiding with selection of dimensionality include Cattell's scree test [23], Kaiser's eigenvalue greater than unity rule [64], the Maximum Likelihood test [61], Horn's parallel analysis [54], Velicer's Minimum Average Partial (MAP) [136] and Bartlett's chi-square test [8]. The literature also contains studies that compare the principal component ex-

traction methods, including [55] and [41]. Furthermore, Zwick and Velicer *et al.* [149, 150] have investigated various principal component extraction methods across various factors including sample size, number of variables, number of components, and component saturation.

The experimental results of Zwick and Velicer [149] suggest that Cattell's SCREE test and Velicer's MAP test performed better than other tests when tested on a variety of situations. Compared to the other rules for determining the number of components to retain, Cattell's scree test is a graphical test, and is one of the most popular used test for quickly identifying the number of components to retain.

5.2 Parameter Settings for LSA Applications to Source-code

This Section provides an insight into research related to LSA and source-code similarity with emphasis on parameter selection.

The literature does not appear to contain any LSA based tools for detecting source-code plagiarism in student assignments or an evaluation of LSA on its applicability to detecting source-code plagiarism. Mozgovoy [100, 101] describes various source-code plagiarism detection approaches with emphasis on source-code plagiarism. He states that the current literature lacks a deep evaluation of LSA and its applicability to detecting plagiarism.

The only literature that appears is that by Nakov [106] which briefly describes findings on applying LSA to a corpus of source-code programs written in the C programming lan-

guage by Computer Science students. Their results show that LSA was able to detect the copied programs. It was also found that LSA had given relatively high similarity values to pairs containing non-copied programs. The authors assume that this was due to the fact that the programs share common language reserved terms and due to the limited number of solutions for the given programming problem. The authors mention that they have used datasets comprising of 50, 47, and 32 source-code files and they have set dimensionality to k=20. Considering the size of their corpora, their choice of dimensions appears to be too high, and we strongly suspect that this was the main reason that the authors report very high similarity values to non-similar documents. The authors justify the existence of the high similarity values to be due to files sharing language reserved terms. However, the use of a suitable weighting scheme and appropriate number of dimensions can reduce the chances of this happening. In their paper, the authors do not provide details of their choice of parameters other than their choice of dimensions (which also lacks justification). Work done by Nakov [106] is very brief and lacks proper analysis and evaluation.

LSA has been applied to source-code with the aim of categorising software repositories in order to promote software reuse. Much work has been done in the area of applying LSA to software components. Some of the tools developed include MUDABlue [71] for software categorisation, Softwarenaut [89] for exploring parts of a software system using hierarchical clustering, and Hapax [77] which clusters software components based on the semantic similarity between their software entities (whole systems, classes and methods). Other literature includes the application of LSA to categorising software repositories in

order to promote software reuse [91, 92, 95, 90].

Maletic and Valluri applied LSA to source-code and found that LSA had successfully grouped source-code into clusters based on their semantic similarity, sections of source-code that had a high degree of semantic similarity were grouped together and sections that shared no similarity to others remained apart [92]. Also, Marcus and Maletic have investigated the effectiveness of LSA for identifying semantic similarities between pieces of source-code and they found that LSA successfully grouped semantically similar documents together into clusters [91]. They have developed a system called PROCSSI (short for PROgram Comprehension Combining Semantic Information) which uses LSA to identify semantic similarities between pieces of source-code.

Kawaguchi *et al.* [70] have compared three approaches to software categorisation: clones-based similarity metrics, decision trees, and LSA, in order to find relations between software components. Their corpus consisted of 41 software systems (and 22,048 identifiers). The source-code pre-processing involved deleting all the comments, keywords (language pre-defined words) and words that appeared only in one software system from the code. The cosine measure was used for computing the similarity value of two software systems. Their results show that many similar software components have received low similarity values. Kawaguchi *et al.* [70] identified the problem being with the parameters they had used. Kawaguchi *et al.* [72] took a new approach to detecting the similarity between software systems by introducing a new clustering step. Similar software systems were retrieved based on the similarity between their identifiers. They describe MUDABlue [72],

an LSA based system they have developed for software categorisation. They have used the same pre-processing approach described in [70], which proved to be unsuccessful in their previous experiment. After pre-processing and performing LSA, the similarities between pairs of identifiers were computed using the cosine similarity measure and the similar identifiers were clustered. Each identifier cluster represents a category. Software systems are retrieved based on the identifier clusters and the similar software systems are placed into software clusters. The software clusters are titled using the names of the identifiers with the highest values. The MUDABlue system consists of a database and a web-based user interface. The database retrieves source-code categories and stores them into the database. The user interface allows for browsing the software repository using the categories contained in the database. The evaluation of MUDABlue was computed using the recall, precision, and the f-measure information retrieval evaluation measures. The performance of the system was compared to two other software tools GURU and SVM (Support Vector Machine). GURU and SVM use automatic software categorisation methods. Kawaguchi *et al.* [72] have shown that MUDABlue (f=0.72) has performed better than GURU (f=0.66) and SVM (f=0.65).

Kuhn *et al.* [77, 78] have developed a system called Hapax, for creating semantic clusters of source-code entities. Hapax is built on top of the Moose re-engineering environment. Basically Hapax is concerned with pre-processing, applying LSI and clustering software components. It takes as input a software system decomposed into various levels of granularity such as packages or classes and methods. Initially, Hapax begins by pre-processing

the source-code corpus, which involves removing the keywords of the programming language, and hence only keeping identifier names. Comments found in the source-code were kept, but English words contained in a predefined stop-list were extracted from the comments as these would not help discriminate between documents [89]. The Potter stemming algorithm was also applied such that words were reduced to their morphological root, for example, words *entity* and *entities* would become *entiti*. Compound identifier names are split into parts, for example word *FooBar* is split into two words, *foo* and *bar*. A term-by-document matrix is then created. After experimenting with various weighting schemes the authors decided to use the *ltf-idf* (log term-frequency local and idf global weighting algorithms) weighting scheme. However, they do not provide evidence of these experiments in their published work [77, 78] or proper justification for their selection. LSI is then applied to the term-by-document matrix to build an index with semantic relationships. The authors have used 10 to 25 dimensions in their experiments. The similarity between term vectors and between document vectors was computed with the cosine measure. These similarities were represented in a correlation matrix. The authors have used a hierarchical clustering algorithm for clustering the entities in the correlation matrix. This clustering algorithm represents clusters as a dendrogram (hierarchical tree) with clusters as its nodes and documents as its leaves. Based on a given threshold, or a predefined number of clusters, the dendrogram can be broken down into clusters. They use the LSI index to retrieve the semantic similarity between software entities (whole systems, classes and methods). The documents in clusters are treated as queries for retrieving the relevant terms, and clusters are labeled

using those terms. The software entities and their sub-entities were clustered according to their similarity. Kuhn *et al.* [77] applied LSI to cluster software components with the assumption that software parts that use similar terms are related. The most important terms were obtained from each cluster in order to produce the cluster descriptions. The similarities were presented in correlation matrix. Using Hapax they have performed experiments with software written in Java and Smalltalk. Hapax represents the clusters using a distribution map and a correlation matrix. Hapax was tested on three corpora; the first corpora consisted of 11,785 terms and 726 documents at the granularity of classes written in Smalltalk; the second corpus consisted of 2,600 terms and 4,324 documents at the granularity of methods written in Smalltalk, and the third corpus consisted of 2,316 terms and 806 classes written in Java.

The research by Lungu *et al.* [89] is concerned with LSA and using interactive visualisation software for interactively exploring the software entities and their relationships using hierarchical clustering. Their research is closely related to reverse software engineering. They have modified the Hapax software tool to include interactive visualisation of clustered software components. Limitations of Hapax are that it fails to properly identify clusters if the identifiers are poorly named, and comments are poorly described. The authors identify the strength of LSI to retrieve similar artifacts regardless of their attribute names, however, they also identify that if most identifiers in a system are not properly described then Hapax fails to correctly identify clusters of similar documents. Furthermore, the authors identify the issue that decomposing source-code at the granularity level of methods

threatens the performance of LSI, as small methods do not contain enough information to allow LSI to acquire knowledge about the documents (i.e. methods). Lungu *et al.* [89] have developed a tool called Softwarenaut for exploring parts of a software system using hierarchical clustering.

5.3 Vector Based Plagiarism Detection Tools

Information retrieval methods have been applied for source-code plagiarism detection by Moussiades and Vakali [99]. They have developed a plagiarism detection system called PDetect which is based on the standard vector-based information retrieval technique. PDetect represents each program as an indexed set of keywords and their frequencies found within each program, and then computes the pair wise similarity between programs. Program pairs that have similarity greater than a given cutoff value are grouped into clusters. Their results also show that PDetect and JPlag are sensitive to different types of attacks and the authors suggest that JPlag and PDetect complement each other.

LSA has been applied to natural language plagiarism detection of student essays [19]. Britt *et al.*, developed a system called SAIF which identifies suspected cases of plagiarism in student essays. SAIF works by comparing student essay pairs and those with a cosine higher than a given threshold (i.e. 0.75) are considered for possible plagiarism. SAIF was able to identify approximately 80% of texts that contained sentences that were plagiarised or quoted without the use of citation, and LSA performed particularly well at detecting

the documents from which particular essay statements originated from (i.e. it detected approximately 90% of the cases) in order to provide feedback on coverage and plagiarism.

Information retrieval methods have been also been applied for natural-text plagiarism detection. COPS [18] is a program developed for detecting copied documents in Digital Libraries. COPS decomposes each document into sentences and groups sentences into chunks, and then checks for overlapping chunks in documents. The system detects documents that contain a number of matched sentences above a given threshold. The problem with this approach is that it is confused by equations, figures, and abbreviations. COPS relies on punctuation for recognising when a sentence begins and ends, and problems were encountered with detecting documents with poor punctuation. Also, another problem of COPS was that it did not detect plagiarism when parts of the sentences overlapped with the sentences of the original document. To avoid the drawbacks of COPS they have developed a plagiarism detection system called SCAM (Stanford Copy Analysis Mechanism)[126].

SCAM is based on the standard vector-based information retrieval technique. Each document is represented as a vector containing the frequency of each possible word occurrence in a document. Similar documents are detected by comparing the vector of a given document (or query) against the vectors of documents in the document collection using a new similarity measure, the relative frequency model (RFM). The RFM similarity measure is a modified version of the cosine similarity measure. Instead of frequencies of words in the document, they use the relative frequencies of words in the cosine similarity algorithm.

In addition, the similarity measure they proposed contains a parameter whose value

must be entered by the user and it is unclear to the authors what the ideal parameter value should be for datasets other than the ones tested in their experiment [126].

One of the problems encountered by both vector-based tools PDetect [99] and COPS [126] is the number of false positives returned.

An information retrieval technique worth mentioning is the one created by Broder, called Shingles [20, 21]. This technique was developed for finding duplicate and near duplicate documents on the web. Each document is decomposed into a sequence of tokens. Each document is presented as a continuous set of tokens called a *shingle*. A shingle is known as an q-gram where q is the length of the shingle. Shingles uses a mathematical concept called *resemblance*, which measures the percentage of overlap between two documents (intersection of features over the union of features between two documents). The resemblance between two documents, A and B is a number between 0 and 1, and the nearer the resemblance is to 1 the more likely that the documents are "roughly the same", i.e. they have the same content regardless of modifications such as formatting, minor corrections, capitalisation, etc. Duplicate and near-duplicate documents are retrieved based on a given threshold.

5.4 Conclusion

LSA has been applied successfully to many applications and the literature suggests that the performance of LSA greatly depends on factors such as the corpus, and on the choice

of parameters including the weighting scheme, number of dimensions, and pre-processing parameters. In addition, parameter choice also depends on the task LSA will be applied to.

The literature review shows that many researchers have applied different weighting schemes when applying LSA to tasks such as retrieval, categorisation, classification of data, and no clear indication exists as to which parameters work best for tasks. In addition, many researchers choose their parameters without justifying their choices. Their choice of weighting scheme is one parameter which is often neglected to be appropriately justified.

The literature suggests that parameter choice can influence the performance of LSA, and suggests that when LSA is introduced to a new task that parameters should be optimised for that specific task.

Chapters 6 and 7 are concerned with evaluating the efficiency of LSA for the new task of detecting similar source-code files. We describe experiments concerned with investigating the influence of parameters that drive the effectiveness of source-code similarity detection with LSA.

Chapter 6

Parameters Driving the Effectiveness of Source-Code Similarity Detection with LSA

Source-code analysis with LSA has recently been a topic of increasing interest. Although previous authors have applied LSA to various information retrieval tasks, it is still not clear which and how parameters impact on the effectiveness of LSA. Literature suggests that parameters must be optimised for the specific task for which LSA has been applied.

This Chapter presents an analysis of the influence of parameters on the effectiveness of source-code similarity detection with LSA using real source-code datasets. It describes experiments conducted with general LSA parameters and source-code specific pre-processing

parameters. The general parameters we experimented with were term-weighting schemes and dimensionality (i.e. how many factors to retain), and the source-code specific pre-processing parameters involve source-code comments, programming language reserved terms (i.e. keywords), and skeleton code.

We show that each of the identified parameters influences the performance of LSA, and that choice of parameters is interdependent of each other.

6.1 Introduction

Previous researchers have reported that LSA works well with textual information retrieval, and suggest optimal parameter settings for best performance. However, it is not clear which and how parameters influence the effectiveness of LSA. We are aware that performance of LSA depends on the corpus itself, and thus what might be 'optimal' parameters for corpus A, may not be optimal for corpus B, hence no set ideal parameter settings exist. We aim to investigate the influence of parameters and identify those settings that appear to perform better in overall for the task of source-code similarity detection with LSA.

The experiments described in this Chapter are concerned with finding the combination of parameters that drive the effectiveness of similar source-code file detection with LSA. We are not aware of any literature investigating the behaviour of parameters for the task of similar source-code file detection. We identify pre-processing parameters specific to source-code and evaluate their impact on LSA performance.

The experiments described in this Chapter aim to answer the following questions:

- What is the impact of keeping/removing comments and/or Java reserved terms and/or and skeleton code on LSA performance?
- What is the impact of weighting algorithms on LSA performance? Which weighting algorithm works best for the task?
- How does choice of dimensionality impact on performance? What is the optimal number of dimensions for each corpus?

Document pre-processing, as described in Section 5.1, is a common procedure in information retrieval. The experiments discussed in this Chapter focus on the impact of those pre-processing parameters specific to Java source-code, i.e. source-code comments, Java reserved terms (i.e. keywords), and skeleton code. We have not performed stemming or stop-word removal on the comments within the source-code files. This raises a question:

- What is the impact of applying stemming and/or stop-word removal to source-code comments on the effectiveness of source-code similarity detection with LSA?

We will not attempt to answer the above question, because our focus is mainly on pre-processing parameters within the source-code (rather than on the comments). We are considering comments in their original form and applying only pre-processing as it is applied to source-code.

The similarity measure applied to compute the distance between two vectors influences the retrieval performance of LSA [143]. In our experiments we use the cosine similarity measure to compute the distance between two vectors, as it is the most common measure of similarity used in the literature and has been shown to produce good results.

Another open question here is,

- How does the cosine measure compare to other distance measures for the task of similar source-code file detection with LSA?

Literature claims that the *cosine document length normalisation*, which adjusts the term values according to the document length, can improve retrieval performance [128]. *Pivoted document length normalization* [127] is a technique used to modify normalization functions. With document length normalization, the probability of retrieval of a document is related to the normalization factor used while weighting the terms for those documents, i.e. the higher the normalization factor values for a document, the lower its chances of retrieval. Pivoted document length normalization aims to improve retrieval performance by correcting the problem of normalization factors, and thus adjusts the document length factor assigned to each document such that documents of all lengths are retrieved with similar chances. In effect, the pivoted document length normalization algorithm is an adjustment to document length normalization algorithms such that relevant documents of smaller size will have a better chance of being retrieved. In an experiment by Singhal *et al.* [127], an improvement of 9-12% in average precision (for 50 queries) was obtained when the pivoted cosine

document length normalisation was used over cosine document length normalization.

The results from the experiments we performed returned very high precision values and thus we did not experiment with pivoted document length normalisation, however the following question remains unanswered,

- What is the impact of applying pivoted cosine document length normalisation on LSA effectiveness for similar source-code file detection?

In this Chapter we describe the results of an experiment investigating various parameters influencing the behaviour of LSA on the application of detecting similar source-code files, and customise the parameters such that LSA can achieve its best performance. Our hypothesis is that system performance depends on factors such as the corpus, choice of pre-processing parameters, choice of weighting scheme, and choice of dimensionality. Furthermore, we hypothesise that parameters are interdependent of each other.

6.2 Experiment Methodology

For the experiments we used four Java datasets. The datasets are all real datasets created by University students as part of their course. The Java datasets were obtained from computer science programming courses from the University of Warwick. Ethical consent had been obtained for using the datasets.

In order to evaluate the retrieval performance of LSA, a set of queries and their relevant

files are required. The process for creating the sets of queries and their relevant files is as follows.

1. We initially passed the corpora into external tools JPlag and Sherlock. The output gathered together from external tools consisted of groups of similar files. Using the output of the tools, we created four preliminary lists (i.e. one corresponding to each corpus) containing groups of similar source-code files. To create a list of queries and their relevant files, one random file was selected from each file group to act as the query and the remaining files of the group were acting as the relevant files to the query.

2. Preliminary LSA experiments were performed using the preliminary lists of queries and their relevant files. The weighting schemes and dimensionality settings by which LSA performed (using the MAP evaluation measure) best were selected. The findings from this initial experiment revealed that the combination of tnc weighting scheme, RC pre-processing parameter (as described in Section 6.4), and setting the value of k between 15 and 30 dimensions worked well on all datasets.

3. Thereafter, those parameters were applied to our PlaGate tool, described in Chapter 8, in order to detect more similar file pairs, and create a complete (to the best of our knowledge) list of similar file groups.

4. A final list containing queries and their relevant files was formed, the experiment to test parameters was re-conducted and the performance of LSA using various param-

eters was re-evaluated and the findings are described in this Chapter.

During these experiments the *independent variables* vary, while the *controlled variables* remain constant. The performance of LSA is evaluated with different independent variables, to measure its performance with different behaviours. Our independent variables were the weighting scheme, pre-processing technique, and choice of dimensionality. Our controlled variables were the queries and their relevant files, the corpora, and the measure of similarity. Our dependent variables were the Mean Average Precision, LPM, HFM, and Sep./HFM evaluation measures.

We wrote source-code programs, using the MATLAB programming language, for performing the task of similar source-code file detection and evaluation, allowing us to alter the independent variables we selected in our experimental design.

6.3 Evaluation of Performance

Many IR evaluation measures exist and these were described in Chapter 4. To evaluate the overall effectiveness of each algorithm we will use Average Precision (AP) as the initial overall evaluation measure. AP appears to be an evaluation measure frequently used by the information retrieval community for evaluating system performance. AP takes into consideration the position of the relevant files in the retrieved list of files. An AP value of 1.00 indicates that only relevant files were detected in the top ranked with no irrelevant files between them.

The Mean Average Precision (MAP) evaluation measure is the mean (or average) AP values of all queries. When computing AP for each query, we computed it by taking into consideration the precision values of all relevant files for the given query (i.e. no threshold was applied to shorten the list of retrieved files). Thus AP was computed up to rank position n, where n is the total number of files in the corpus. This is because we want to evaluate the rank position of all the relevant files for each query, and by taking into consideration the position of all relevant files we get a picture of overall performance. The higher the value of the MAP, the better the performance of the system, i.e. the fewer non-relevant files exist between relevant ones.

To gain a better picture of the behaviour of a retrieval algorithm it is more precise to use more than a single measure to summarise its full behaviour [5]. We will apply the evaluation measures proposed by Hoad and Zobel [53] as additional measures for evaluating system performance, because these take into consideration similarity values between the queries and the retrieved files. When the relevant files to a query are not known beforehand (i.e in real use of the system), the similarity values given to files in the returned ranked list can help indicate and differentiate between the potentially relevant and non-relevant files for a given query.

6.4 The Datasets

The characteristics of the Java datasets, including the number of relevant file pairs detected are described in Section 9.3. Each of the datasets contained source-code files written by students who had been given frameworks of four Java classes and were required to write the source-code.

The Linux shell script *sed* utility and the TMG tool [147] were used for conducting the pre-processing tasks and creating six sub-datasets for each corpus. The pre-processing conducted is standard to that carried out by the information retrieval community when pre-processing files destined to be represented as vectors of a term-by-file matrix. During this pre-processing, terms that were solely composed of numeric characters were removed, syntactical tokens (i.e, semi-colons, and colons) were removed, terms combining letters and numeric characters were kept, terms that occurred in less than two files were removed and upper case letters were translated into lower case.

In the context of source-code files, identifiers which consisted of multiple terms separated by underscores were treated as single terms (i.e. after pre-processing student_name became one term studentname). Hyphenated terms were also merged together. The reason for merging rather than separating such identifiers is because each identifier whether it consists of one or two words (separated by underscore) represents one meaning in the source-code file. One issue with pre-processing parameters was whether to separate source-code identifiers comprising of multiple words into their constituents. Taking into consideration

the amount of time needed to develop an application to perform the task of breaking multiple words into their constituents, it was decided not to perform large experiments using this parameter. We considered it useful to perform a small experiment for investigating the impact on system effectiveness when separating identifiers comprising of multiple words. The results from that would indicate whether it would be beneficial to separate such terms with regards to system effectiveness. The small experiment is described in Chapter 7.

Other than the pre-processing mentioned above, we did not interfere with the source-code provided by students, we did not correct any code that would not compile, and did not use any stemmers or spell checkers. We then began the experiment by creating the sub-datasets for each corpus of files.

Each Java dataset is pre-processed six times using the following pre-processing parameters, thus creating six sub-datasets for each dataset:

- Keep Comments (KC)
- Keep Comments and Remove Keywords (KCRK)
- Keep Comments and Remove Keywords and Remove Skeleton code (KCRKRS)
- Remove Comments (RC)
- Remove Comments and Remove Keywords (RCRK)
- Remove Comments and Remove Keywords and Remove Skeleton code (RCRKRS)

Pre-processing by *removing comments* involves deleting all comments from source-code files and leaving files solely consisting of source-code. When *keeping comments* in files all source-code comments and the source-code are kept in the files.

A list of all known *Java reserved terms or keywords* was used as a stop-list. The words contained in the stop-list were removed from all Java files to create the relevant sub-datasets (i.e. KCRK, and RCRK).

All terms found in the *skeleton files* relevant to each corpus were added to the stop list of Java reserved terms, thus creating four new different stop lists (i.e. one for each corpus), and each stop list was applied to the relevant corpus to create the new sub-datasets (KCRKRS, and RCRKRS).

In total, we were testing 1224 different algorithm combinations (12 weighting schemes, 6 pre-processing parameters, and 17 different dimension settings) on four corpora. We decided to experiment with those weighting schemes most commonly used and tested by LSA researchers. The following twelve local, global, and document length normalisation weighting schemes were tested: txx, txc, tfx, tfc, tgx, tgc, tnx, tnc, tex, tec, lex, lec. The algorithms for these weighting schemes are presented in Chapter 4.

A range of seventeen different k dimensions will be tested, ranging from 2 to n (i.e. 2, 5, 10, 15, 20, 25, 30, 35, 40, 45, 50, 60, 70, 80, 90, 100, n) where n=r, where r is the rank of the term-by-file matrix. The number of n dimensions depends on the corpus size. We decided to stop at 100 dimensions and also to display performance at n dimensions.

Our datasets contained no more than 200 files, and thus we decided to show the effect of choosing too few and too many dimensions by evaluating LSA's performance when using a range of 2 to 100 dimensions, and also when using the maximum possible number, n, of dimensions. Mathematically, as the value of k approaches the rank r of the term-by-file matrix, LSA approaches the traditional VSM retrieval performance, thus the performance of the VSM is equivalent to LSA when k=r [75].

After creating the sub-datasets, the TMG tool was applied and a term-by-file matrix was created for each sub-dataset. We created batch scripts using the MATLAB programming language to apply LSA and evaluate system performance using various parameters (the weighting functions were provided by the TMG tool, our script called those functions). Once the batch process was finished, all the results were stored in a matrix, and these results were then copied into statistical packages for creating charts.

6.5 Experiment Results

We evaluated the application of various weighting schemes and k dimensionality settings using the AP evaluation measure. The AP for each query, and thereafter the MAP which is the mean AP of all queries, were computed. We define *MMAP* as Maximum Mean Average Precision, which is the MMAP value reached by LSA when using a certain parameter. Tables 6.1, 6.2, 6.3, and 6.4 show the MMAP values for each dataset's sub-datasets. After computing the MAP value using various weighting schemes and k dimensions, the

maximum MAP (MMAP) value reached by LSA when using each weighting scheme was recorded alongside the number of dimensions that were needed for the particular MMAP value to be achieved. For example, for dataset's A sub-dataset KC (Table 6.1), the highest MMAP value reached by the txx weighting scheme was 0.86 at 20 dimensions. The emboldened values are the highest MMAP values recorded for each sub-dataset.

When comparing the performance of LSA using various weighting algorithms, it is important to take into consideration the number of dimensions each weighting algorithm needed to reach its MMAP value. For example, observing the results for sub-dataset KC of dataset A, shown in Table 6.1, the highest MMAP recorded was that by the lec algorithm, MAP=0.96 k=40, closely followed by the tnc algorithm, MAP=0.95 k=25. The difference in MAP is only 0.01 but there is considerable difference in the number of dimensions needed, i.e. lec needed 15 more dimensions. Of course, the number of dimensions impacts on computational effort, therefore, from this perspective, we can consider tnc to be a better algorithm with regards to dataset's A sub-dataset KC. An unanswered question here is:

- How much precision should we compromise by adjusting dimensionality on improving computational effort?

We will not attempt to answer this question because our experiments are not focusing on evaluating the computational efficiency aspects of LSA.

The findings from Tables 6.1, 6.2, 6.3, and 6.4 are summarised as follows:

- Dataset A: The tnc weighting scheme achieved best MMAP value and thus outper-

Table 6.1: MMAP values for dataset A

	KC	KCRK	KCRKRS	RC	RCRK	RCRKRS	Averages
txx k	0.86 20	0.86 60	0.86 60	0.78 15	0.75 40	0.54 106	0.77 50.17
txc k	0.86 20	0.86 45	0.85 45	0.79 40	0.80 60	0.55 2	0.79 35.33
tfx k	0.94 40	0.92 40	0.92 40	0.91 35	0.87 45	0.61 70	0.86 45.00
tfc k	0.93 70	0.94 80	0.93 80	0.88 60	0.88 60	0.60 60	0.86 68.33
tgx k	0.73 25	0.70 20	0.69 15	0.74 20	0.69 15	0.54 2	0.68 16.17
tgc k	0.82 30	0.74 50	0.64 10	0.75 20	0.69 40	0.57 10	0.70 26.67
tnx k	0.92 40	0.92 40	0.92 40	1.00 35	1.00 25	0.63 70	0.90 41.67
tnc k	**0.95** **25**	**0.96** **25**	0.95 25	**1.00** **15**	**1.00** **15**	0.61 80	**0.91** **30.83**
tex k	0.87 30	0.87 30	0.88 30	0.85 30	0.82 35	0.60 60	0.82 35.83
tec k	0.94 80	0.94 80	0.94 70	0.87 70	0.87 60	0.61 80	0.86 73.33
lex k	0.94 20	0.93 30	0.93 30	0.97 20	0.97 25	0.62 70	0.90 32.50
lec k	0.96 40	0.94 20	**0.95** **20**	0.97 10	1.00 90	**0.61** **45**	0.91 37.50

Table 6.2: MMAP values for dataset B

	KC	KCRK	KCRKRS	RC	RCRK	RCRKRS	Average
txx k	0.94 60	0.91 70	0.86 80	0.90 10	0.88 45	0.85 40	0.89 50.83
txc k	**0.95** **15**	0.88 20	0.86 15	0.90 10	0.87 5	0.60 25	0.84 15.00
tfx k	0.78 45	0.78 70	0.78 70	0.74 40	0.74 40	0.73 40	0.76 50.83
tfc k	0.84 15	0.83 15	0.83 15	0.79 15	0.78 15	0.77 35	0.81 18.33
tgx k	0.92 35	0.82 60	0.77 70	0.91 25	0.88 15	0.81 40	0.85 40.83
tgc k	0.92 15	0.78 20	0.74 10	**0.95** **15**	0.89 20	0.80 20	0.85 16.67
tnx k	0.84 70	0.84 70	0.83 60	0.90 60	0.90 60	0.90 60	0.87 63.33
tnc k	0.85 10	0.85 10	0.85 10	0.91 15	**0.91** **15**	**0.91** **15**	**0.88** **12.50**
tex k	0.80 45	0.80 45	0.80 45	0.74 90	0.74 90	0.74 90	0.77 67.50
tec k	0.83 15	0.81 15	0.80 15	0.79 15	0.79 15	0.77 15	0.80 15.00
lex k	0.86 60	0.85 60	0.85 60	0.86 40	0.86 40	0.86 40	0.86 50.00
lec k	0.88 15	**0.88** **15**	**0.87** **15**	0.90 10	0.89 10	0.87 10	**0.88** **12.50**

Table 6.3: MMAP values for dataset C

	KC	KCRK	KCRKRS	RC	RCRK	RCRKRS	Average
txx	0.78	0.74	0.98	0.81	0.77	0.77	0.81
k	15	15	35	90	80	90	54.17
txc	0.81	0.76	0.96	0.82	0.78	0.78	0.82
k	40	50	45	80	90	80	64.17
tfx	0.65	0.65	0.91	0.71	0.71	0.70	0.72
k	80	70	70	70	70	70	71.67
tfc	0.73	0.71	0.94	0.75	0.70	0.69	0.75
k	80	90	25	60	50	50	59.17
tgx	0.72	0.71	0.93	0.73	0.69	0.64	0.74
k	90	80	60	50	70	70	70.00
tgc	0.75	0.74	0.92	0.74	0.69	0.67	0.75
k	80	70	60	80	80	100	78.33
tnx	0.83	0.79	0.95	0.82	0.80	0.79	0.83
k	25	25	25	20	35	35	27.50
tnc	**0.84**	**0.82**	**0.97**	**0.88**	**0.85**	**0.85**	**0.87**
k	**20**	**15**	**15**	**20**	**15**	**25**	**18.33**
tex	0.70	0.70	0.90	0.75	0.73	0.71	0.75
k	60	90	50	70	80	80	71.67
tec	0.73	0.72	0.96	0.71	0.70	0.69	0.75
k	80	80	10	60	50	80	60.00
lex	0.74	0.74	0.96	0.74	0.74	0.73	0.78
k	20	20	25	35	60	60	36.67
lec	0.78	0.77	0.93	0.78	0.78	0.75	0.80
k	35	40	25	20	25	25	28.33

Table 6.4: MMAP values for dataset D

	KC	KCRK	KCRKRS	RC	RCRK	RCRKRS	Average
txx k	0.80 25	0.77 60	0.75 45	0.83 30	0.80 60	0.79 50	0.79 45.00
txc k	0.82 20	0.77 20	0.76 20	0.84 30	0.80 10	0.79 10	0.80 18.33
tfx k	0.70 45	0.69 40	0.69 40	0.73 25	0.73 45	0.73 45	0.71 40.00
tfc k	0.74 15	0.74 15	0.74 15	0.78 25	0.77 25	0.77 25	0.76 20.00
tgx k	0.79 30	0.73 25	0.73 25	0.81 35	0.74 70	0.73 25	0.76 35.00
tgc k	0.73 30	0.70 30	0.70 30	0.79 10	0.74 15	0.73 15	0.73 21.67
tnx k	0.71 15	0.71 20	0.70 15	0.81 10	0.83 10	0.82 10	0.76 13.33
tnc k	**0.82** **10**	**0.79** **15**	**0.79** **15**	**0.92** **5**	**0.86** **15**	**0.86** **15**	**0.84** **12.50**
tex k	0.70 45	0.70 45	0.70 50	0.74 50	0.73 40	0.73 40	0.72 45.00
tec k	0.67 10	0.67 5	0.67 15	0.72 25	0.72 25	0.72 25	0.70 17.50
lex k	0.64 15	0.65 15	0.65 15	0.70 25	0.72 90	0.72 90	0.68 41.67
lec k	0.76 15	0.76 15	0.76 15	0.78 20	0.78 20	0.78 20	0.77 17.50

formed all other weighting schemes across 4 out of 6 sub-datasets. For subdataset KCRKRS, lec reached MMAP of 0.95 at 20 dimensions whereas tnc reached 0.95 at 25 dimensions which indicates similar performance. For RCRKRS, the MMAP of lec and tnc were equal (0.61) however, tnc needed 35 more dimensions to reach that MMAP value. On average, across sub-datasets the tnc weighting scheme outperformed all other weighting schemes giving the highest MMAP (0.91) at an average 30.38 dimensions. Comparing the overall results of the tnc and lec weighting schemes, both their average MMAP values were 0.91, however, tnc performed better by means of dimensionality, i.e. on average tnc needed 6.67 fewer dimensions to reach the MMAP of 0.91. Applying the tnc weighting scheme, computing the SVD using 15 dimensions, and using the RC or RCRK pre-processing parameters gives excellent retrieval results. For sub-datasets KC, KCRK, KCRKRS, RC, RCRK performance was high with only 15 to 25 dimensions depending on the sub-datasets. Clearly, removing comments, keywords, and skeleton code at once (RCRKRS) removes too much terms (and meaning) from files and thus causes major decrease in LSA performance. Comparing KCRKRS with RCRKRS, when using the tnc term-weighting scheme, we see a difference in LSA performance, i.e the MMAP value of KCRKRS is MMAP=0.95 at k=25 and the the highest MAP of RCRKRS is MMAP=0.61 k=80 dimensions, which suggests that when removing keywords and skeleton code, it is best to keep comments in as these are likely to hold some meaning about the files, but this depends on the corpus.

- Dataset B: On average the tnc and lec weighting schemes outperformed all other weighting schemes with a MMAP average of 0.88 at an average of 12.50 dimensions. The highest MMAP value was achieved by txc and tgc weighting schemes, achieving MMAP of 0.95 at 15 dimensions, when applied to the RC and KC sub-datasets. However, the pattern of behaviour of those two weighting schemes appears unpredictable across sub-datasets. In overall, the performances of tnc and lec were very close. Highest MMAP values were achieved when applying the tnc weighting scheme and using 15 dimensions – performance was 0.91 at 15 dimensions for sub-datasets RC, RCRK, RCRKRS, and performance for the remaining sub-datasets ranged between 0.85 and 0.91 when using 10 to 15 dimensions.

- Dataset C: Interpreting the results for dataset C is very straightforward – the tnc weighting scheme outperformed all other weightings. Highest MMAP performance (0.97) was achieved when using 15 dimensions and the KCRKRS sub-dataset, followed by the RC sub-dataset with MMAP of (0.88) with 20 dimensions.

- Dataset D: Again, the results are very clear – the tnc weighting scheme outperformed all other weighting schemes. Highest MMAP was 0.92 at only 5 dimensions, followed by RCRK and RCRKRS both achieving MMAP of 0.86 at 15 dimensions.

Results suggest that choice of parameters is interdependent – performance of weighting schemes depends on the choice of pre-processing parameters, the corpora and the choice of dimensionality. In overall, the average MMAP values of each dataset show that the tnc

weighting scheme performed well on most sub-datasets when using between 10 and 20 dimensions. With regards to which pre-processing parameter (i.e. sub-dataset) performed best, the results vary depending on the weighting algorithm applied. When using the tnc term-weighting scheme the highest MMAP values achieved for each dataset are as follows:

- Dataset A: RC (MMAP=1.00, k=15), RCRK (MMAP=1.00, k=15),
- Dataset B: RC (MMAP=0.91, k=15), RCRK (MMAP=0.91 k=15), RCRKRS (MMAP=0.91, k=15),
- Dataset C: KCRKRS (MMAP=0.97, k=15), and
- Dataset D: RC (MMAP=0.92, k=5).

The results show that the tnc weighting scheme and the RC pre-processing parameter performance reached highest MMAP values for datasets A, B and D. With regards to dataset C, highest performance (MMAP=0.97 k=15) was achieved using the tnc weighting scheme on the KCRKRS sub-dataset, followed by MMAP=0.88 k=20 when using the tnc weighting algorithm on the RC sub-dataset.

It is clear that choice of pre-processing has a major impact on performance. Figures 6.1, 6.2, 6.3, and 6.4 show the performance of datasets A, B, C, and D respectively, using the tnc weighting algorithm and various k dimensions. These figures illustrate the difference in system performance when using various pre-processing parameters (i.e. represented as sub-datasets). For example, in dataset A, Figure 6.1 shows that applying the RCRKRS

pre-processing has a devastating effect on system performance, which suggests that by removing comments, keywords and skeleton code altogether we remove important meaning from files.

An important finding is that although choice of pre-processing influences performance, it does not influence the number of dimensions needed. The pattern of behaviour across Figures 6.1, 6.2, 6.3, and 6.4 is very similar – MAP performance improves significantly after 10 to 15 dimensions and then remains steady or decreases when 35 dimensions are reached, and then begins to fall slowly and gradually. Another common observation is that when reaching maximum number of possible dimensions, performance decreases significantly. This shows that at maximum dimensionality, irrelevant information is captured by the LSA model, which causes LSA to not be able to differentiate between similar and non-similar files.

At n dimensions, performance reaches that of the SVM [75]. Our results show that when customising our parameters, and selecting between 10 and 35 dimensions, the performance of LSA outperforms that of the SVM. Figure 6.5 shows that when n dimensions are used, the performance of LSA worsens. When the value of k is set to n, where n is equal to the number of files in the term-by-file matrix, the performance of LSA reaches that of the VSM, and from this we can conclude that applying LSA for source-code similarity detection, performs better that the VSM.

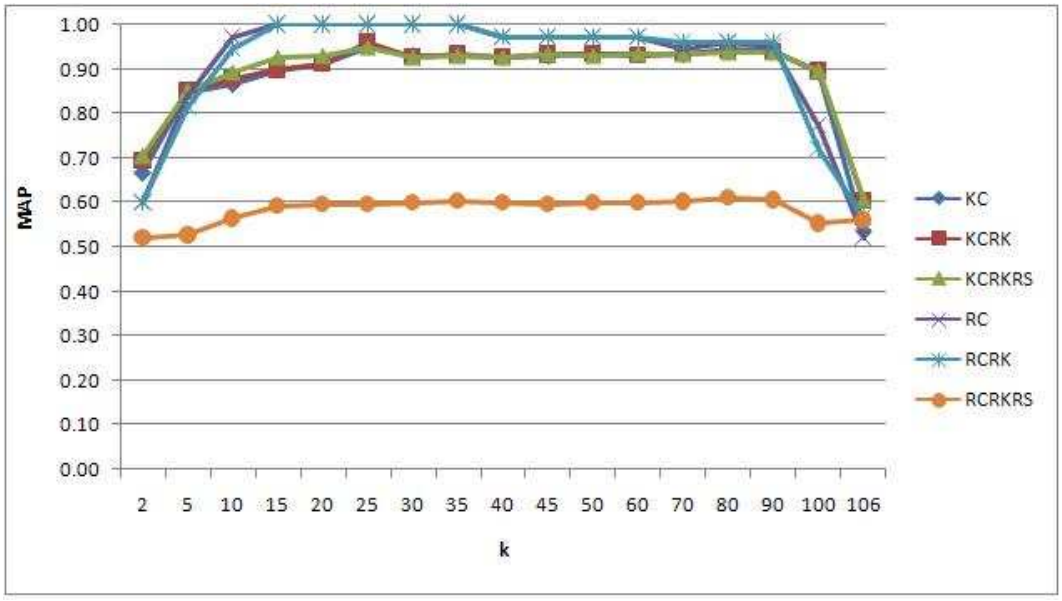

Figure 6.1: Dataset A: MAP performance using the tnc weighting scheme across various dimensions.

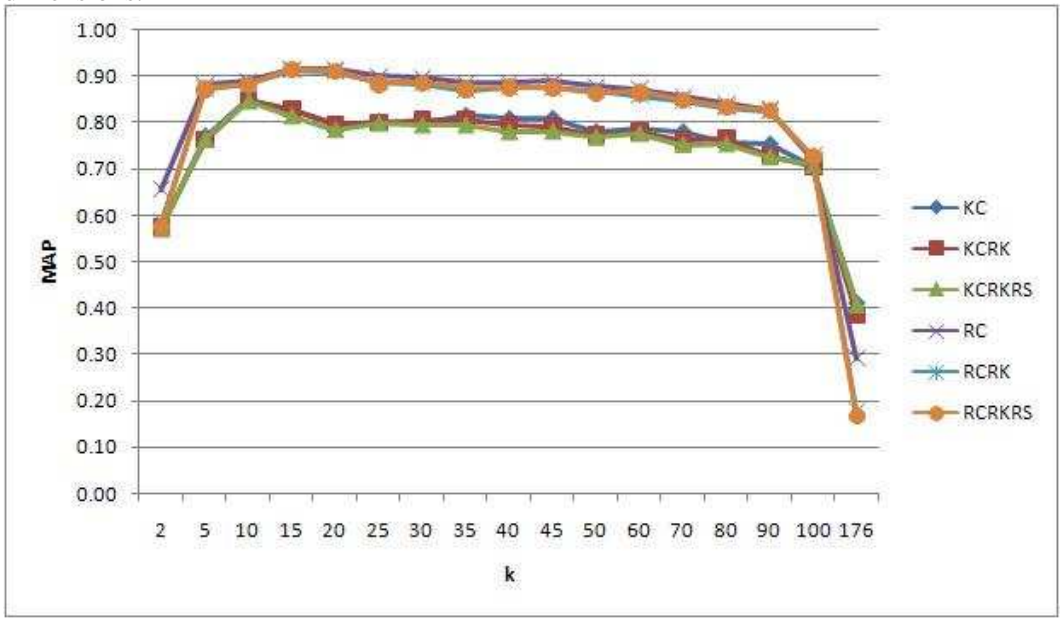

Figure 6.2: Dataset B: MAP performance using the tnc weighting scheme across various dimensions.

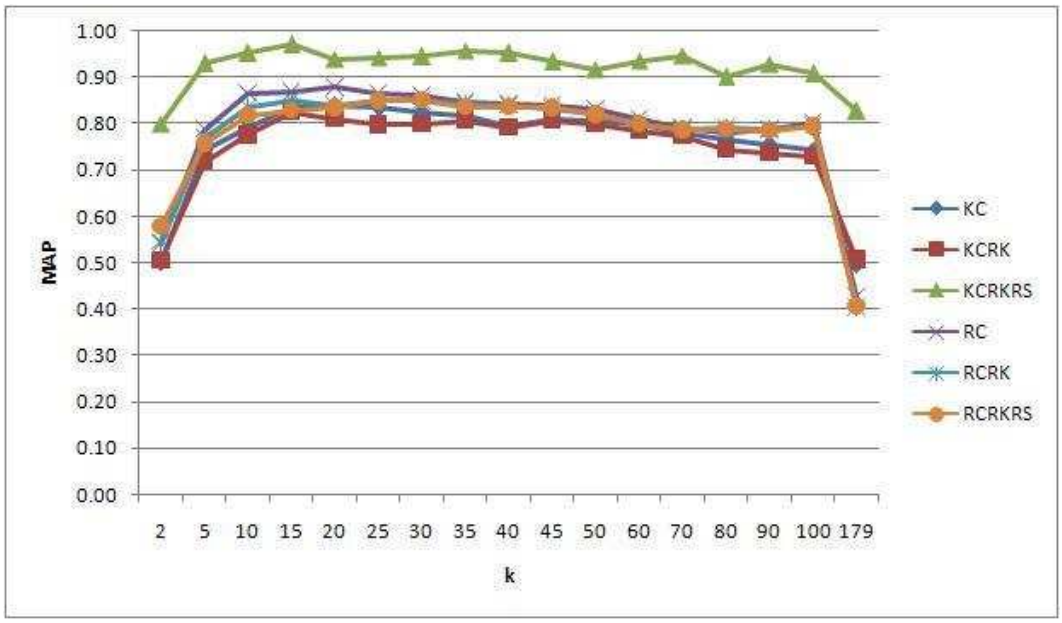

Figure 6.3: Dataset C: MAP performance using the tnc weighting scheme across various dimensions.

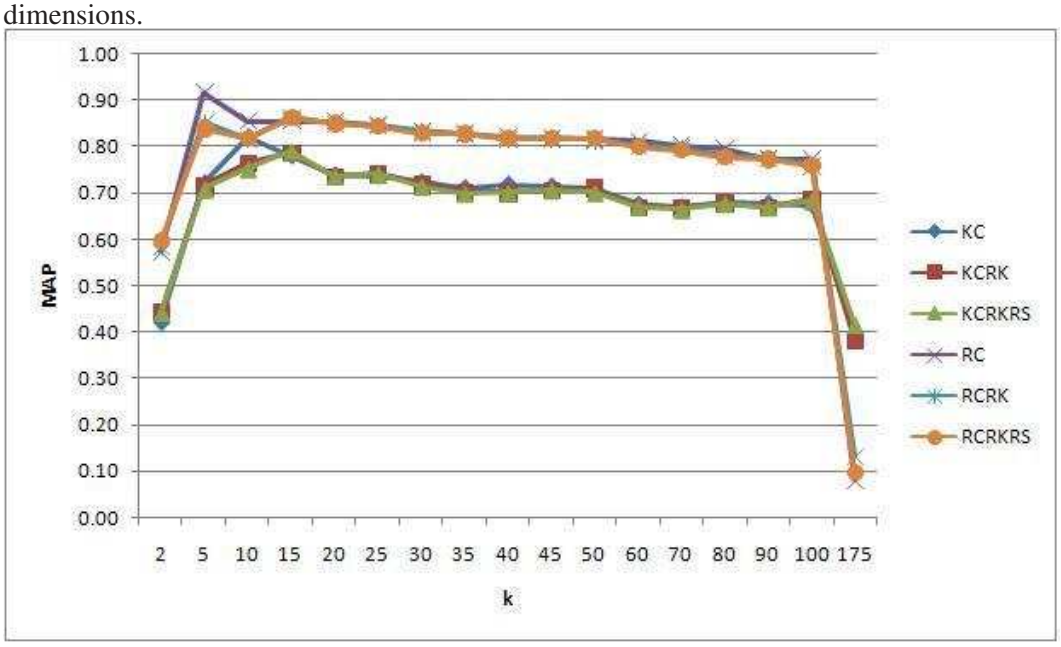

Figure 6.4: Dataset D: MAP performance using the tnc weighting scheme across various dimensions.

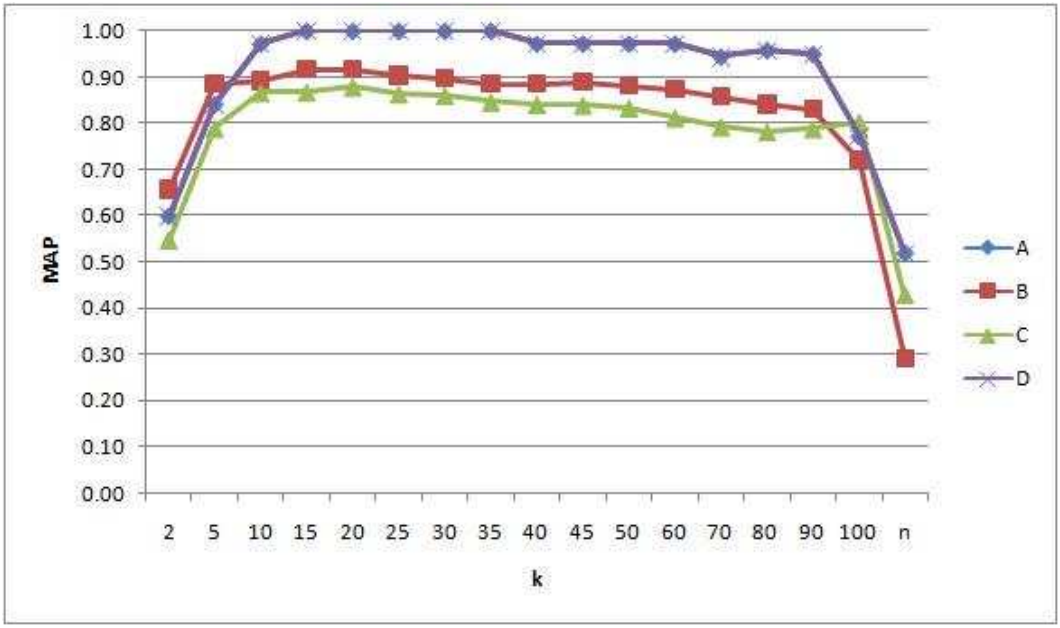

Figure 6.5: Datasets A, B, C, D: MAP performance using the RC sub-dataset and the tnc weighting scheme across various dimensions.

6.6 Investigation into Similarity Values

LSA computes the cosine similarity between a given query vector with all file vectors in the vector space. Our hypothesis is that choice of pre-processing and choice of dimensions impacts on the similarity values between a query and its relevant files. A good system would give high scores to similar file pairs, low scores to non-similar file pairs, and using the terminology and evaluation measures proposed by Hoad and Zobel [53] the greater the separation between the cosine values given to the LPM and HFM the better the performance of the system. Hoad and Zobel's [53] evaluation measures are described in Section 4.10.

In the discussion in Section 6.5 we established that the tnc weighting algorithm gave good results (i.e. high MMAP) and the RC pre-processing parameter was also a good

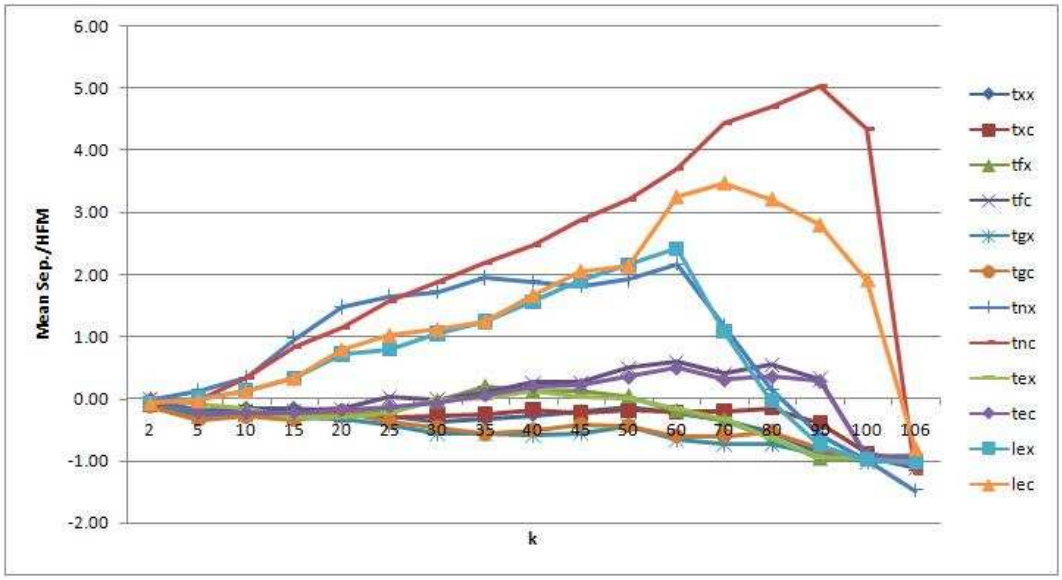

Figure 6.6: Dataset's A sub-dataset RC: Sep./HFM performance using various weighting algorithms and dimensions.

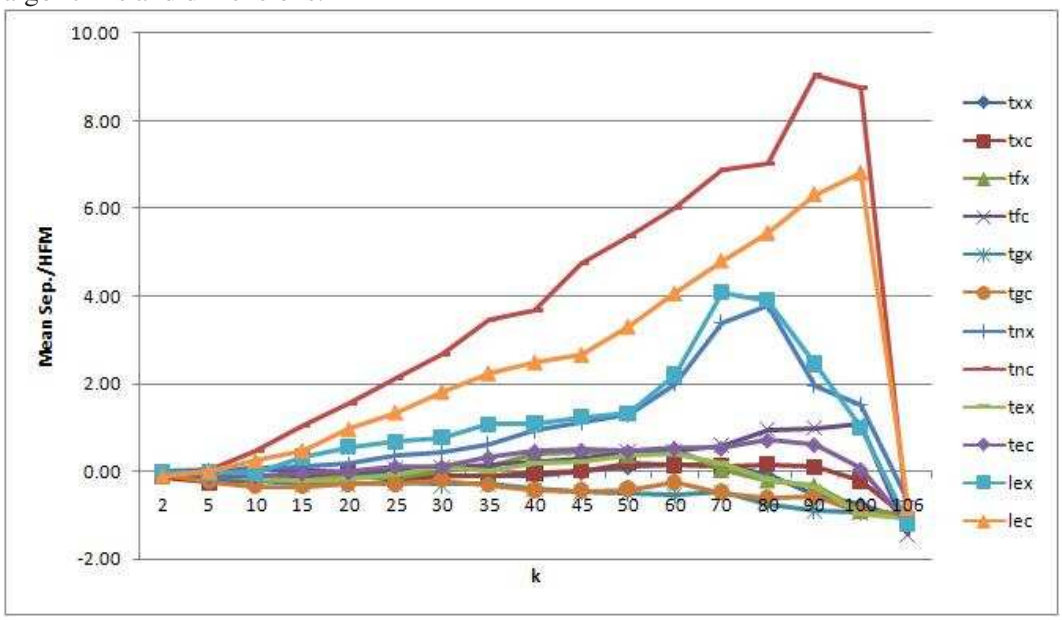

Figure 6.7: Dataset's A sub-dataset KC: Sep./HFM performance using various weighting algorithms and dimensions.

choice. The results also revealed that with regards to achieving high AP, 15 was good choice of dimensions.

Figure 6.5 shows the performance of sub-datasets RC of all datasets when using the tnc weighting algorithm across various dimensions. The chart shows that good choice of dimensions is between 15 and 35, using more dimensions starts to have a negative impact on performance.

Because our aim is to find the best parameter combinations for the task of source-code similarity detection with LSA, we are also concerned with the similarity values (and not simply the positions of relevant files in the retrieved list of files) assigned to similar source-code files when various parameters are applied. This is because we aim to use the findings from the experiments described in this Chapter when developing the LSA based tool for source-code similarity detection (as described in Chapters 8, 9, and 10).

Choice of weighting scheme, pre-processing and dimensionality also influence the similarity values given to files. Take for example dataset A, Figure 6.6 shows the Sep./HFM performance using various weighting algorithms and dimensions on the RC sub-dataset, and Figure 6.7 shows the Sep./HFM performance using various weighting algorithms and dimensions on the KC sub-dataset. Clearly, each Figure illustrates that performance is highly dependent on the choice of weighting scheme, and comparing the two Figures shows that similarity values are also dependent on the choice of pre-processing parameters.

Although best AP results were returned at 15 dimensions we strongly suspect that with

regards to similarity values given to relevant and non-relevant files, 15 dimensions is too few – however, for an information retrieval task where results are ordered in a ranked list and where the similarity value does not really matter, then 15 dimensions are appropriate for the system to retrieve the most relevant files in the top ranked. For the purpose of our task were thresholds will be applied, similarity values are crucial to system performance.

Figures 6.8, 6.9, 6.10, and 6.11 show the mean values of the LPM, HFM, Separation, and Sep./HFM, respectively, over all queries for each dataset's RC sub-dataset. The average Sep./HFM values for all datasets are also displayed in each Figure.

Figure 6.8 shows that on average, values given to the relevant files lowest in the retrieved list are near and above 0.80 when using 2 to 15 dimensions. In order to decide whether 15 dimensions is sufficient, we need to observe the similarity values given to non-relevant files (i.e. the HFM values). Figure 6.9 shows that at 15 dimensions non-relevant files received, on average, very high similarity values, i.e. above 0.70. Separation between relevant and non-relevant files (as shown in Figure 6.10) is very small (0.03 and below) which indicates that many non-relevant files received high similarity values. Figure 6.11 shows that between 2 and 15 dimensions, overall performance measured by Sep./HFM was very low (0.22 and below). These results clearly suggest that with regards to similarity values, more dimensions are needed if the functionality of filtering files above a given threshold will be included in system implementation. At 30 and above dimensions, the average values given to non-relevant files are 0.53 or below (see Figure 6.9), and there appears to be a good amount of separation (see Figure 6.10) between the similarity values given to relevant

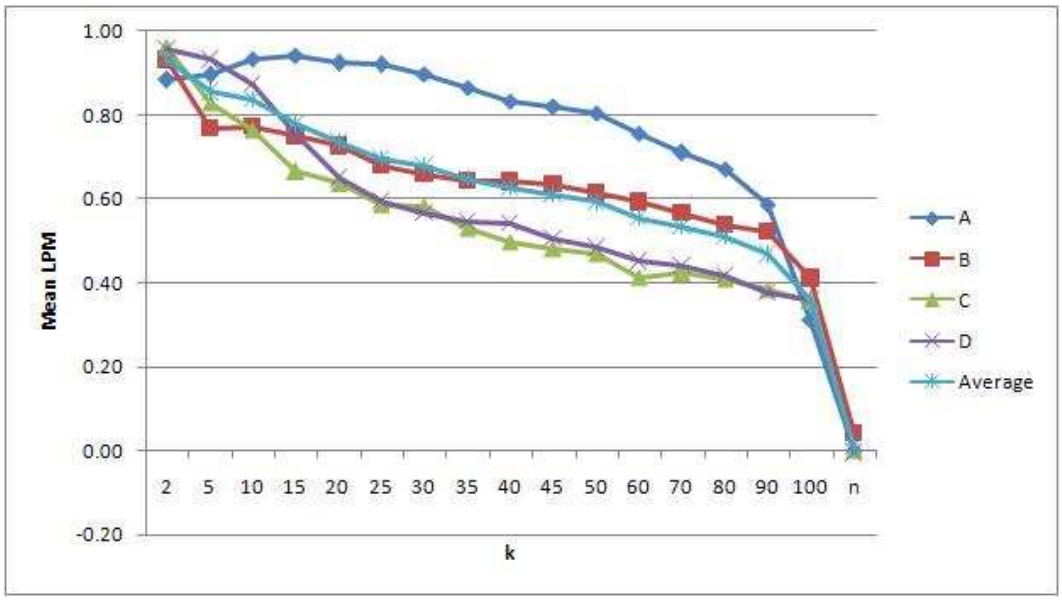

Figure 6.8: Datasets A, B, C, D: Mean LPM using the RC sub-dataset and the tnc weighting scheme across various dimensions.

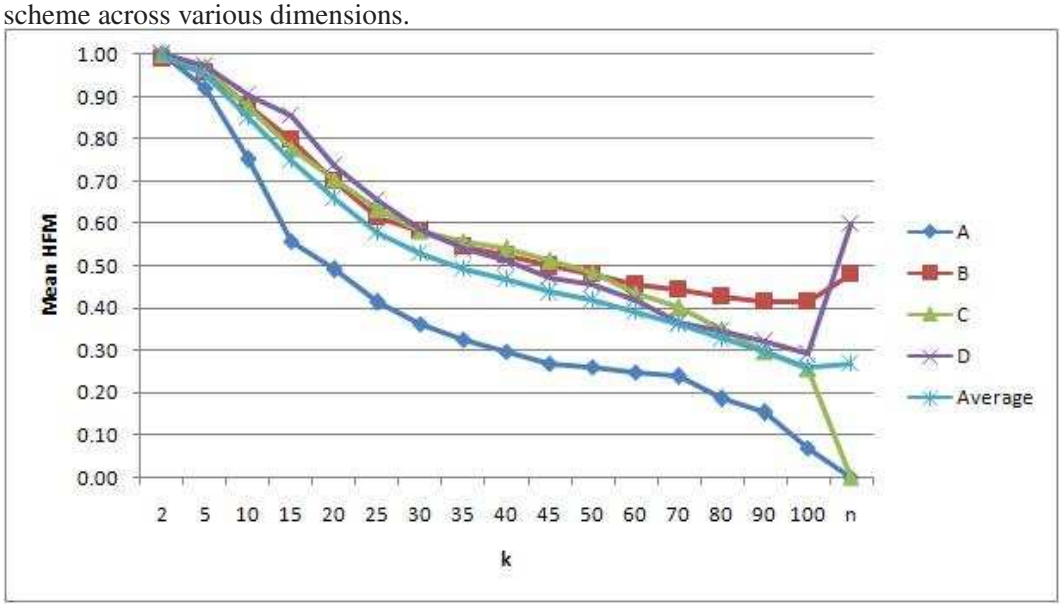

Figure 6.9: Datasets A, B, C, D: Mean HFM using the RC sub-dataset and the tnc weighting scheme across various dimensions.

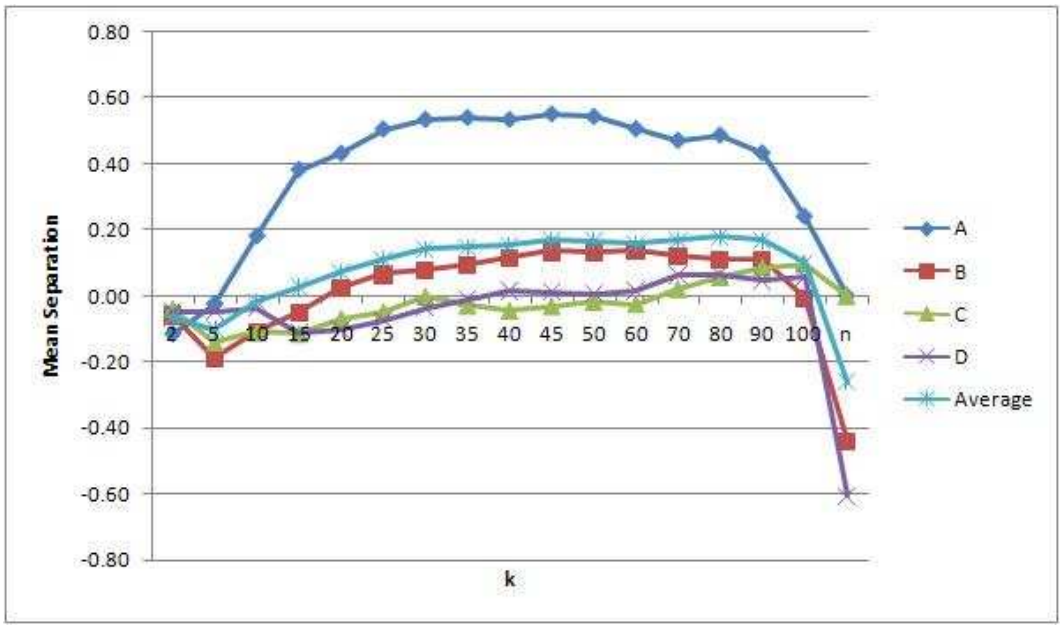

Figure 6.10: Datasets A, B, C, D: Mean Separation using the RC sub-dataset and the tnc weighting scheme across various dimensions.

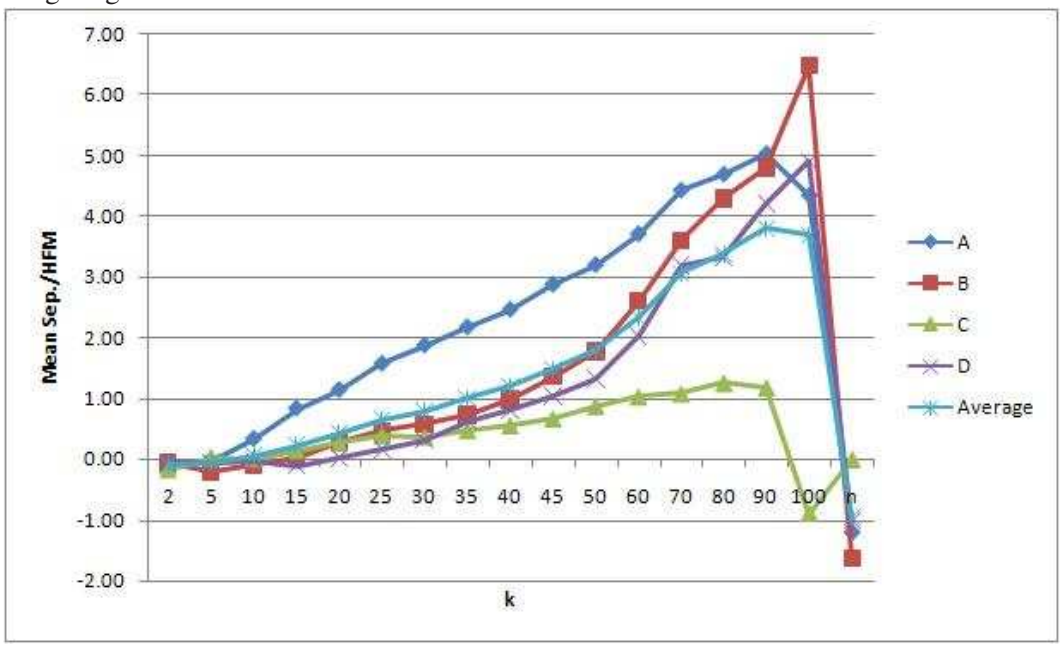

Figure 6.11: Datasets A, B, C, D: Mean Sep./HFM using the RC sub-dataset and the tnc weighting scheme across various dimensions.

and non-relevant files (i.e. average separation at 30 dimensions is 0.14, highest average separation recorded is 0.18).

With regards to good choice of dimensionality, observing the values of separation show that there is not much change in the curve after 30 dimensions. Also, system performance by means of Sep./HFM increases considerably (i.e. by 0.57 points) between 15 and 30 dimensions.

Figure 6.12 shows that, in overall, when term-weighting schemes were combined with document length normalisation the results improved. Weighting scheme names ending with the letter *c* (e.g. txc, tnc) are those which were combined with cosine document length normalisation.

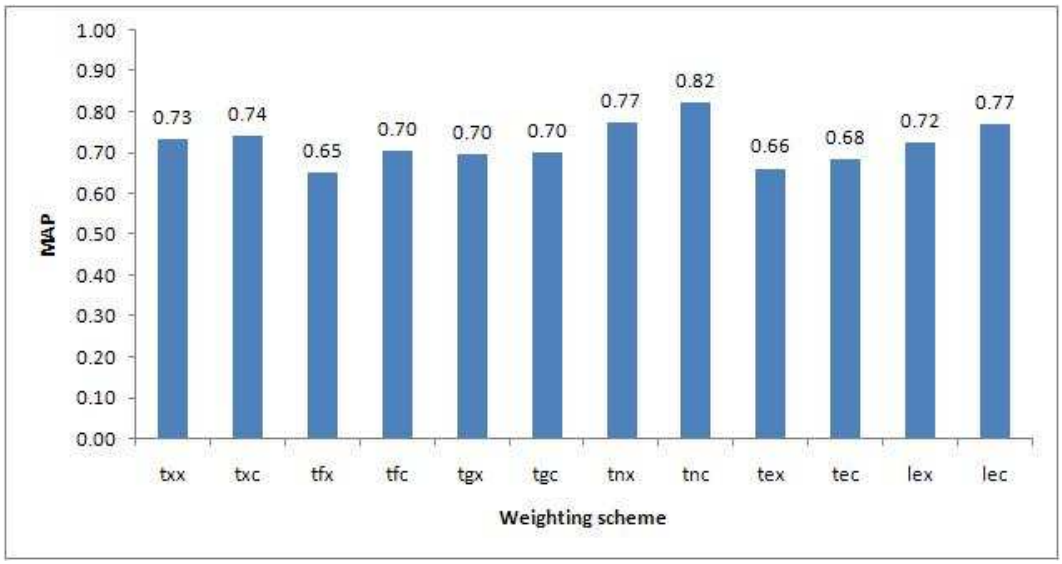

Figure 6.12: Average MAP values for each weighting scheme across all dimensions and datasets.

6.7 Discussion

The results clearly show that choice of parameters influences the effectiveness of source-code similarity detection with LSA. Conducting experiments using various parameters and reaching a conclusion as to which parameters in overall performed best for our task, has proven to be a rather tricky task. Most evaluations of performance in the literature are based on precision, however, in our case, it was necessary to investigate the similarity values assigned to files as these are crucial for the task of source-code similarity detection for which we will be applying LSA.

Our results suggest that pre-processing by removing comments and using the tnc weighting scheme consistently gives good results. Setting k to 15 dimensions is ideal for all datasets, if the aim of the system is to retrieve a ranked list of files sorted in order of similarity to a query, without use of thresholds (i.e. cut-off values) or without depending on the value of the similarity between the query and retrieved files as indicators of similarity.

In the event that threshold values will be used by the system, the results suggest that the number of dimensions must be increased to 30, such that similarity values are more representative of the actual similarity between the queries and files. In addition, our results show that having a low number of dimensions gives relatively high values to non-similar files, and thus retrieving many false positives when using a threshold value as a cutoff.

In overall, the results revealed that removing comments from source-code improves results. The experiment also revealed that when using the tnc weighting scheme, remov-

ing comments and keywords reduced the MMAP value of datasets C and D. Furthermore, removing comments, Java reserved terms and skeleton code all at once caused a major reduction in performance of dataset A, and not much effect on datasets B, and C, and caused some reduction in performance on dataset D (i.e. compare columns RC, RCRK, RCRKRS of Table 6.1, 6.2, 6.3, 6.4).

With regards to which the most effective choice of pre-processing and term-weighting parameters were for our datasets, our results suggest that the tnc weighting scheme and RC pre-processing parameters performed consistently well. On average, optimal number of dimensions appears to be between 10 and 30 depending on the task, (i.e. whether or not thresholds will be used by the system) after 30 dimensions performance begins to deteriorate.

Comparing our results with previous research in the field of textual information retrieval, we found that our results are different – applying the term frequency local term weighting, and normal global weighting algorithms outperformed all other weighting schemes (with or without combining it with the cosine document length normalisation). These results are not consistent with those by Dumais [38] who found that the *normal* global weighting performed significantly less than all other weighting schemes she experimented with. We have experimented with all weighting algorithms Dumais has experimented with and more.

Researchers have tested various weighting schemes and best results were reported when applying the *logarithm* as the local, and the *entropy* as the global weighting scheme [107, 38, 112]. However, Wild *et al.* [143] found that the IDF global weighting outper-

formed all other weighting functions, and no clear indication as to which local weighting function performed best. Our results also show that the log-entropy combination performed well but only when combined with document length normalisation. Dumais [38] has not experimented with document length normalisation algorithms.

6.8 Conclusion

This Chapter described results gathered from conducting experiments to investigate the impact of various parameters on the effectiveness of source-code similarity detection with LSA. Our results show that the effectiveness of LSA for source-code file similarity detection is dependent on choice of parameters. The results also suggest that choice of parameters are interdependent of each other, and also dependent on the corpus and the task that LSA has been applied to.

We evaluated the influence of parameters on LSA performance using a range of evaluation measures that do and do not make use of similarity values. Our results suggest that for tasks which involve giving a query to the system that outputs ranked results (i.e. files similar to the query in order of similarity) then the similarity values are not important when evaluating system performance. However, in a system where thresholds will be used (i.e. when a user gives a query and requires the system to retrieve files where similarity between the query and files are above a given threshold) then similarity values are of great importance to system performance. The results show that choice of dimensionality has a great

impact on these similarity values, and that retrieval systems which make use of thresholds are likely to require approximately 15 more dimensions. These results strongly suggest that choice of LSA dimensionaly is task dependent.

Chapter 7 describes a small experiment conducted to closely observe the behaviour of LSA using pre-processing parameters specifically applicable to source-code files.

Chapter 7

Experiments on Pre-Processing Parameters using a Small Dataset

The aim of this Chapter is to investigate the impact of source-code specific pre-processing parameters on the effectiveness of source-code similarity detection with LSA. We experimented with pre-processing parameters that involve transforming identifiers, removing keywords from source-code, and source-code comments. We will conduct this experiment using the tnc weighting scheme, as this was reported to be the most effective, from the experiments discussed in Chapter 6.

Table 7.1: Source-code specific pre-processing parameters

	Keep or remove comments	**Merge or separate terms containing multiple words**
KCMT	Keep Comments	Merge Terms
KCST	Keep Comments	Separate Terms
RCMT	Remove Comments	Merge Terms
RCST	Remove Comments	Separate Terms

7.1 The Datasets, Queries and Relevant Files

This Chapter describes an initial investigation into the impact of source-code specific parameters on LSA performance. This experiment was carried out on a small dataset consisting of 51 source-code files produced by 17 undergraduate students at the University of Warwick. The students were given three skeleton classes and were advised to use these as a starting point for completing their program.

The TMG software [147] was used to pre-process the files. During pre-processing the following were removed from the files: terms solely composed of numeric characters, syntactical tokens (e.g. semi-colons, colons) and terms consisting of a single letter. In addition, comments were removed from the source-code. This procedure was performed using the *sed* Linux utility.

This dataset was pre-processed four times, each time using a different parameter combination as shown in Table 7.1. This created four sub-datasets; KCMT, KCST, RCMT, and RCST. The Linux shell script *sed* utility was used to strip the comments from the source-code and create the RCMT and RCST datasets.

Table 7.2: Dataset characteristics

Dataset Characteristics	**KCMT**	**KCST**	**RCMT**	**RCST**
Number of documents	51	51	51	51
Number of terms (after the normalization)	682	637	208	183
Average number of indexing terms per document	443.69	564.02	192.94	302.90

Pre-processing by *removing comments* involves deleting all comments from source-code files and thus keeping only the source-code in files. When *keeping comments* in files all source-code comments and source-code are kept in the files.

Merging terms involves transforming each term comprised of multiple words into a single term regardless of the obfuscation method used within that term, e.g. student_name and studentName both become studentname. Hyphenated terms were merged together.

Separating terms involves splitting terms comprising of multiple words into separate terms, e.g. student_name and studentName become two terms – student and name. Hyphenated terms were separated into their constituent terms.

The TMG software [147] was thereafter used to create the term-by-file matrices and to apply the *tnc* term-weighting scheme as described in Section 4.7. Table 7.2 shows the characteristics of the four datasets.

Three query files and their relevant files were identified from the corpus and used for evaluating the performance of LSA on the four sub-datasets; these are shown in Table 7.3. No other similar files existed in the corpus. Relevant files were those files we identified as similar to the query files. The identified pairs contain differences that could be considered

Table 7.3: Queries and their relevant files

Query ID	**Relevant files**	**Total relevant**
Query 1: F2	F2, F5, F8, F11	4
Query 2: F14	F14, F17	2
Query 3: F37	F37, F19, F22, F25, F28, F31, F34, F40, F25	8

Table 7.4: Source-code differences between queries and their relevant files

File ID pairs	**Structural changes**	**Identifier & variable changes**	**Iteration & selection statement changes**	**Math expression changes**	**Introduction of bugs**
F2-F5	√	√	√	√	
F2-F8	√	√	√	√	
F2-F11	√	√	√		
F14-F17	√				
F37-F22		√			
F37-F28		√			
F37-F31		√			
F37-F34		√			
F37-F19	√	√			
F37-F40		√			√
F37-F25		√			

as attempts to hide similarity between the files. The source-code differences between each query and its relevant files are shown in Table 7.4.

In Table 7.4 *Structural changes* are those changes made to the source-code structure which include changes to the position of source-code fragments with or without affecting the overall program functionality. *Identifier and variable changes* include modifying the names, types and position of identifiers, changing modifiers (i.e. private, public, protected, final) in variable declarations, modifying values assigned to variables, and splitting or merging identifier declarations. *Iteration and selection statement modifications* include modifying conditions to become equivalent conditions, modifying statements to become

equivalent statements, modifying the order of selection statements, converting a selection statement to an equivalent one (i.e. an *if* to a *switch* statement) and adding more execution paths. *Mathematical expression modifications* include modifying the order of the operands (e.g. $x < y$ can become $y > x$), modifying mathematical expressions without affecting the output and combining multiple mathematical expressions. *Introduction of bugs* involves making changes to the source-code such that functionality is affected and that the source-code cannot be compiled.

7.2 Experiment Methodology

Once the term-by-file matrices of sub-datasets KCMT, KCST, RCMT and RCST (described in Section 7.1) were created, the tnc weighting scheme was then applied to each of the term-by-file matrices. We have selected to use the tnc term-weighting scheme because it has been shown to work well on our source-code datasets (as discussed in Chapter 6). SVD and dimensionality reduction were performed on the matrix using 10 dimensions. An experiment using the queries (see Table 7.3), and evaluating performance using the AP evaluation measure, revealed that 10 was a sufficient number of dimensions. The reduced term-by-file matrix was then reconstructed by $A_k = U_k S_k V_k^T$.

Three file vectors were selected from the A_k matrix and treated as query vectors. The selected query vectors correspond to the files identified to be used as queries, shown in Table 7.3. The cosine similarity between each query vector and each of the other file vectors in

the vector space were computed.

For each query, a ranked list of files was created with the most relevant positioned at the top of the list. Performance of each query was then evaluated using the evaluation measures discussed in Section 4.10. The results from conducting the experiment are described in Section 7.3.

The main aim of this experiment is to investigate the impact of source-code specific parameters on LSA performance with regards to identifying similar source-code files. In addition to the standard evaluation measures used by the IR community, such as recall and precision, it is important to use evaluation measures that make use of similarity values, as discussed in Chapter 6. These evaluation measures are also explained in Section 4.10.

7.3 Experiment Results

LSA performance was evaluated using a retrieved list consisting of all files returned in order of similarity to the query such that statistics on all relevant files were captured. Hence, when similarity was computed between each query vector and all other files vectors, the full list comprising the 51 files in order of similarity was used. The results are shown in Table 7.5. In addition to the evaluation measures described in Section 4.10, we will use the *P@100%R*, which is the value of precision when 100% recall is achieved. The Table shows precision and AP values at 100% recall. Figure 7.1 shows the value of precision at various points of recall. The discussion in the remaining of this Section is based on Table 7.5.

Table 7.5: Performance Statistics for Queries 1 - 3

KCMT							
Query no.	R-Precision	P@100%R	AP	ESL	LPM%	HFM%	Sep.
1	1.00	1.00	1.00	2.50	86.37	39.85	46.53
2	1.00	1.00	1.00	1.50	98.16	48.08	50.08
3	0.88	0.67	0.96	5.00	20.15	32.01	-11.87
Average	0.96	0.89	0.99	3.00	68.23	39.98	28.25
KCST							
Query no.	R-Precision	P@100%R	AP	ESL	LPM%	HFM%	Sep.
1	1.00	1.00	1.00	2.50	72.25	62.30	9.96
2	1.00	1.00	1.00	1.50	99.06	29.26	69.80
3	0.88	0.53	0.93	5.50	15.03	43.21	-28.18
Average	0.96	0.84	0.98	3.17	62.11	44.92	17.19
RCMT							
Query no.	R-Precision	P@100%R	AP	ESL	LPM%	HFM%	Sep.
1	1.00	1.00	1.00	2.50	97.05	31.29	65.75
2	1.00	1.00	1.00	1.50	99.62	28.82	70.79
3	1.00	1.00	1.00	4.50	67.68	32.72	34.96
Average	1.00	1.00	1.00	2.83	88.11	30.94	57.17
RCST							
Query no.	R-Precision	P@100%R	AP	ESL	LPM%	HFM%	Sep.
1	1.00	1.00	1.00	2.50	85.48	48.77	36.72
2	1.00	1.00	1.00	1.50	99.83	23.06	76.77
3	1.00	1.00	1.00	4.50	74.91	49.21	25.70
Average	1.00	1.00	1.00	2.83	86.74	40.34	46.40

When comparing various systems, the better system would return the highest LPM, lowest HFM, highest separation, and highest HFM./Separation values. Comparing the results of the KCMT and KCST sub-datasets, the average LPM is higher by 6.12% in KCMT which means that on average the lowest relevant files received a higher similarity value with KCMT. In addition separation is greater for KCMT by 11.06% which indicates that there was a better separation between relevant and non-relevant files.

AP was below 1.00 for both KCMT and KCST meaning that irrelevant files were returned amongst the relevant ones. This observation is verified by the value of HFM being higher than that of LPM for Query 3 for both KCMT and KCST. Precision was higher in KCMT which means that relevant files were retrieved higher up the list than in KCST. The average ESL values for KCMT and KCST compared, show that for KCST the user has to go through a longer list of files before reaching all the relevant files. These results suggest that when comments are kept in the source-code, merging terms that are comprised of multiple words appears to improve retrieval performance, for this particular corpus.

Comparing RCMT and RCST, the average LPM is higher for RCMT by 1.40% and separation is higher by 10.77% which suggests that merging terms increases the separation value and hence causes a much greater separation between the values given to the last relevant and first non-relevant file.

Precision and AP were the same for RCMT and RCST. The main difference was in the similarity values given to files. On average, RCST has given lower values to LPM files and higher values to HFM files. Looking at the overall performance by means of Sep./HFM,

results show that removing comments and avoid splitting identifiers (i.e. RCMT) improves system performance.

Overall performance improved when identifiers consisting of multiple words were not separated (i.e. merged in the case where they were separated by obfuscators) regardless of whether comments were removed or kept in the source-code performance improved greatly. The Sep./HFM values were: KCMT 0.71%, KCST 0.38%, RCMT 1.85%, and RCST 1.15%. Therefore, merging terms increased overall performance by 0.33% when comments were kept in the source-code, and by 0.70% when comments were removed from the source-code.

In summary, RCMT outperformed all other pre-processing combinations. From the results we can hypothesize that performance improves when removing comments and merging terms that contain multiple words.

Table 7.5 shows that performance of Query 3 was in overall relatively low, and hence, the behaviour of Query 3 across the datasets was further investigated. Table 7.7 shows the top 20 results in ranked order of similarity to Query 3 when the different sub-datasets are used. Files are ranked in order of their similarity to the query with the most similar positioned at the top of the list. In the *Rel?* column *R* denotes relevant files, and *NR* non-relevant files. The cosine between the query and the file is also shown in the *Sim* column. The *Sim* column also shows the average cosine values of the relevant files (R) and the non-relevant files (NR).

Table 7.6: Part 1: Top 20 Query 3 results in ranked order

KCST				KCMT			
R	FID	Rel?	Sim	R	FID	Rel?	Sim
1	37	R	1.00	1	37	R	1.00
2	28	R	0.95	2	19	R	0.98
3	31	R	0.94	3	28	R	0.97
4	19	R	0.92	4	31	R	0.95
5	22	R	0.90	5	22	R	0.91
6	25	R	0.71	**6**	**40**	**R**	**0.71**
7	24	NR	0.43	7	25	R	0.64
8	**40**	**R**	**0.35**	8	13	NR	32
9	33	NR	0.28	9	24	NR	0.32
10	43	NR	0.24	10	33	NR	0.21
11	12	NR	0.22	11	43	NR	0.21
12	32	NR	0.21	**12**	**34**	**R**	**0.20**
13	30	NR	0.19	13	15	NR	0.15
14	39	NR	0.18	14	30	NR	0.15
15	**34**	**R**	**0.15**	15	14	NR	0.15
16	27	NR	0.14	16	39	NR	0.14
17	3	NR	0.11	17	32	NR	0.13
18	14	NR	0.11	18	27	NR	0.09
19	13	NR	0.09	19	5	NR	0.06
20	46	NR	0.08	20	46	NR	0.06
Average		R	0.74			R	0.79
Average		NR	0.20			NR	0.16

Continued on the next page...

Table 7.7: Part 2: Top 20 Query 3 results in ranked order

RCST				RCMT			
R	FID	Rel?	Sim	R	FID	Rel?	Sim
1	37	R	1.00	1	37	R	1.00
2	31	R	0.99	2	28	R	0.98
3	22	R	0.98	3	22	R	0.98
4	**40**	**R**	**0.96**	4	19	R	0.97
5	28	R	0.96	**5**	**40**	**R**	**0.97**
6	25	R	0.96	6	31	R	0.96
7	**34**	**R**	**0.89**	7	25	R	0.87
8	19	R	0.75	**8**	**34**	**R**	**0.68**
9	7	NR	0.49	9	10	NR	0.33
10	10	NR	0.42	10	24	NR	0.24
11	24	NR	0.30	11	46	NR	0.19
12	46	NR	0.25	12	7	NR	0.17
13	43	NR	0.24	13	43	NR	0.16
14	30	NR	0.20	14	16	NR	0.13
15	38	NR	0.16	15	13	NR	0.10
16	12	NR	0.14	16	30	NR	0.09
17	33	NR	0.12	17	29	NR	0.06
18	23	NR	0.12	18	33	NR	0.05
19	3	NR	0.08	19	3	NR	0.04
20	48	NR	0.08	20	23	NR	0.04
Average		R	0.94			R	0.93
Average		NR	0.22			NR	0.13

Table 7.8: Summary of Spearman rho correlations for Query 3 and its relevant files.

Summary table	F19	F22	F25	F28	F31	F34	F40	Average
F37(KCST)	0.85	0.77	0.74	0.87	0.86	0.26	0.44	0.69
F37(KCMT)	0.93	0.78	0.68	0.90	0.86	0.38	0.67	0.74
F37(RCST)	0.52	0.91	0.85	0.85	0.97	0.70	0.81	0.80
F37(RCMT)	0.89	0.87	0.66	0.88	0.84	0.61	0.86	0.80

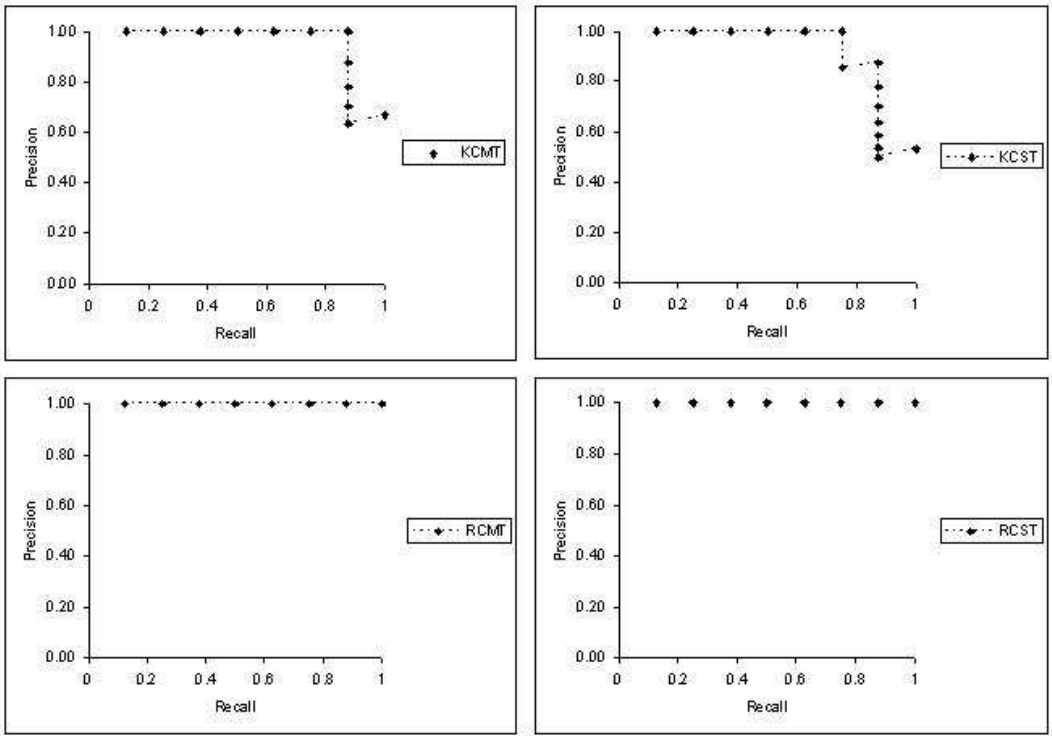

Figure 7.1: Recall-Precision curves for Query 3

Figure 7.1 shows the precision-recall graphs for Query 3. Each graph shows precision at various points of recall. Table 7.7 shows that with regards to the KCST sub-dataset, files F40 and F34 were ranked in 8th and 15th position respectively with irrelevant files preceding them. Merging terms (KCMT) improved the positions of files F40 and F34, however irrelevant files still preceded file F34. KCMT retrieved file F34 in 12th position with four irrelevant files before it. Comparing KCST and KCMT, merging terms increased the average of cosine values given to relevant files by 0.05, and decreased the average similarity value of irrelevant files by 0.04.

Part 2 of Table 7.7 shows that removing comments caused files F34 and F40 to move up the ranked list of files. Comparing RCST and RCMT, the results show that merging terms (RCMT) has slightly decreased the average similarity value of relevant files by 0.01, however the average similarity value of the non-relevant files also decreased by 0.07 which suggests an improvement in performance when terms are merged. RCST and RCMT have retrieved all eight relevant files in the top ranked with no non-relevant files in between them.

In summary the results show that when merging identifiers, the results further improved both when comments were kept in (KCMT) and removed (RCMT) from the source-code. We suspect that since each identifier whether it consists of one, two or more words (separated or not by obfuscators) represents one purpose and meaning in the source-code, separating the terms contained in each identifier increases noise and appears to have a negative impact on LSA performance.

The results also suggest that comments within source-code have a negative impact on

LSA performance, and removing these improves performance for this particular dataset. Next, we investigate this hypothesis by investigating the source-code similarities before and after applying LSA.

7.3.1 LSA performance using the comments of source-code files

Our hypothesis is that keeping comments within source-code files may have a negative impact on LSA performance. To examine this hypothesis we investigate the source-code comment differences between file F37 and its relevant files. We chose to investigate file F37 as it was the one having the largest number of relevant files, as shown in Table 7.3.

All source-code was removed from the files, such that each file only consisted of source-code comments. Comments that contained source-code identifiers that comprised of multiple words, were merged into a single term, because this setting has reported best results in our previous experiments discussed in this Chapter. Thereafter pre-processing as described in Section 7.1, was applied. This produced a 582×51 term-by-file matrix, A, containing only the comments found in the files.

SVD and dimensionality reduction were performed on matrix A, and after experimenting with various dimensionality settings, the dimensionality parameter k, was set to 5. Thereafter, the reduced matrix was reconstructed, creating a term-by-file matrix A_k. Spearman's rho correlations were computed between file F37 and all its relevant files using matrices A, and A_k. Correlations were chosen to investigate the relationship between file F37 and its relevant files before and after applying LSA. Tables 7.9 and 7.10 contain the cor-

Table 7.9: Spearman rho correlations for Query 3 and its relevant files (when only comments are kept in the files) before applying LSA.

tf raw	F19	F22	F25	F28	F31	F34	F37	F40
F19	1.00	0.57	0.45	0.48	0.56	0.41	0.65	0.45
F22	0.57	1.00	0.38	0.58	0.63	0.37	0.62	0.40
F25	0.45	0.38	1.00	0.39	0.38	0.38	0.44	0.42
F28	0.48	0.58	0.39	1.00	0.67	0.40	0.58	0.50
F31	0.56	0.63	0.38	0.67	1.00	0.36	0.66	0.44
F34	0.41	0.37	0.38	0.40	0.36	1.00	0.41	0.40
F37	0.65	0.62	0.44	0.58	0.66	0.41	1.00	0.40
F40	0.45	0.40	0.42	0.50	0.44	0.40	0.40	1.00

Table 7.10: Spearman rho correlations for Query 3 and its relevant files (when only comments are kept in the files) after applying LSA.

tnc k=5	F19	F22	F25	F28	F31	F34	F37	F40
F19	1.00	0.91	0.61	0.87	0.85	0.46	0.97	0.84
F22	0.91	1.00	0.60	0.97	0.98	0.58	0.95	0.79
F25	0.61	0.60	1.00	0.55	0.51	0.80	0.54	0.85
F28	0.87	0.97	0.55	1.00	0.98	0.65	0.92	0.81
F31	0.85	0.98	0.51	0.98	1.00	0.59	0.92	0.74
F34	0.46	0.58	0.80	0.65	0.59	1.00	0.47	0.81
F37	0.97	0.95	0.54	0.92	0.92	0.47	1.00	0.79
F40	0.84	0.79	0.85	0.81	0.74	0.81	0.79	1.00

relations between file F37 and its relevant files before and after LSA is applied, respectively, and Table 7.11 shows the summary of correlations.

Table 7.11 shows that LSA increased the correlations for all but F37-F25 and F37-F34 file pairs, which only increased by 0.06 and 0.09 points respectively. The relatively low

Table 7.11: Summary of Spearman rho correlations for Query 3 (when only comments are kept in the files.)

Summary table		F19	F22	F25	F28	F31	F34	F40	Average
system_a (before LSA)	F37	0.65	0.62	0.44	0.58	0.66	0.41	0.40	0.54
system_b (after LSA)	F37	0.97	0.95	0.54	0.92	0.92	0.47	0.79	0.79
$system\_b - system\_a$		0.32	0.33	0.09	0.34	0.26	0.06	0.39	0.26

correlations between these file pairs, before and after LSA is applied, suggest that the similarity between comments of these file pairs is relatively low. Reflecting back to Table 7.7 we notice that in dataset KCMT files F25 and F34 have again received relatively low similarity values when compared to the similarity values given to the rest of the relevant files. However, this was not the case for dataset RCMT whose files contained no comments – the similarity values between files F37-F25 and F37-F34 have improved considerably[1]. These findings suggest that removing comments from the source-code files can have a positive impact on the retrieval performance of LSA. These results are consistent with our findings described in 6, which suggest that when the tnc weighting scheme is applied, removing comments from the source-code, in overall improves retrieval performance.

7.4 Java Reserved Words and LSA Performance

The reserved words (or keywords) of the Java programming language and the structure of the source-code help one to understand its meaning (i.e. semantics). Our experiments described in Chapter 6, suggest that when appropriate parameters are selected, LSA appears to perform well in identifying similar source-code files regardless of the fact that it treats every file as a bag of words and does not take into consideration the structure of files.

The purpose of this experiment is to investigate the influence of the Java reserved words on LSA performance using a small corpus of source-code files. In this experiment we used

[1]Note that since terms containing multiple words were merged in this experiment, only the results with KCMT and RCMT were compared.

Table 7.12: RCMTRK Performance Statistics for Queries 1 - 3

RCMTRK							
	R-Precision	P@100%R	AP	ESL	LPM%	HFM%	Sep.
Average	1.00	1.00	1.00	2.83	86.72	36.08	50.63

the RCMT sub-dataset (because our previous experiments suggest that these parameters work well on source-code datasets) and removed the Java reserved terms from the dictionary, hence creating a new sub-dataset RCMTRK. This created a 178×51 term-by-file matrix. In total, 30 Java reserved terms were removed from the matrix. The tnc weighting scheme was applied, and after experimentation, dimensionality was set to 10.

The similarity values between the queries and files in the vector space were computed, and performance was evaluated. The results are displayed in Table 7.12. Comparing the results of sub-datasets RCMT with those of RCMTRK, shown in Tables 7.5 and 7.12 respectively, show that RCMTRK received a lower LPM, higher HFM, and a lower separation value, and these are all indicators that RCMTRK has performed worse than RCMT. These results suggest that removing Java reserved words from the corpus can have a negative impact on performance when the corpus is pre-processed using the RCMT settings.

Java reserved words, can be considered as holding information about the structure of files, which provides information about the semantics (i.e. meaning) of files. By removing such information, meaning from files is removed, and this has a negative impact on the performance of LSA. The results gathered from Chapter 6 suggest that whether or not removing Java reserved terms (keywords) has a negative impact on performance is dependent

on the corpus, but also on other parameter choices such as weighting scheme and dimensionality. In overall, the results of both Chapter 6 and this Chapter suggest that when the tnc term weighting scheme is applied and comments are removed from the source-code, it is best to leave the Java reserved terms within the source-code in order to avoid removing too much meaning from files. Furthermore, weighting schemes work by adjusting the values of terms depending on their local and global frequencies, and thus the values of frequently occurring terms (such as Java reserved terms) are managed by the weighting schemes. Frequently occurring Java reserved terms will be adjusted, such that they have relatively lower values in each file, and this suggests that keeping them is not likely to have negative impact on performance, as long as choice of dimensionality is carefully considered. Each dimension captures different levels of meaning about files in the corpus, and thus finding a dimensionality setting that works best for a dataset is important in similarity detection.

7.5 Conclusion

This Chapter follows on from experiments described in Chapter 6, describing experiments conducted with a small dataset to observe the performance of selected pre-processing parameters specific to source-code files. The results from the small experiment revealed that separating identifiers and keeping comments in the source-code increases noise in the data, and decreases LSA performance.

We are aware that the experiment discussed in this Chapter was based on a small dataset

and only three queries, however, the main purpose of the experiment was to use a manageable dataset for conducting experiments and observing the behaviour of LSA when source-code specific pre-processing parameters to a small corpus.

We are also aware that performance of LSA is corpus dependent, and also depends on choice of parameters, and this raises the questions of

- what is the impact of separating identifiers comprising of multiple words on the effectiveness of source-code similarity detection with LSA when using large datasets?, and
- what is the impact of computational efficiency for conducting such a task?

The results described in this Chapter are consistent with those results described in Chapter 6 where four different and larger source-code datasets were used for conducting the experiments. The results from our experiments revealed that removing comments from source-code, merging terms containing multiple words, and keeping Java reserved terms, improves the effectiveness of LSA for the task of source-code similarity detection.

Chapter 8

A Hybrid Model for Integrating LSA with External Tools for Source-Code Similarity Detection

This chapter describes similarity in source-code files and categories of source-code similarity. We propose a model that allows for the application of LSA for source-code plagiarism detection and investigation. The proposed tool, PlaGate, is a hybrid model that allows for the integration of LSA with plagiarism detection tools in order to enhance plagiarism detection. In addition, PlaGate has a facility for investigating the importance of source-code fragments with regards to their importance in characterising similar source-code files for plagiarism, and a graphical output that indicates clusters of suspicious files.

8.1 Introduction

In Chapter 3 we proposed a detailed definition as to what constitutes source-code plagiarism from the perspective of academics who teach programming on computing courses. The proposed definition is mostly concerned with student actions that indicate plagiarism. In this Chapter we go a step further and describe the type of similarity found in student assignments. We also propose a tool for combining LSA and existing plagiarism detection tools.

Once similarity between students' work is detected, the academic compares the detected source-code files for plagiarism. This Chapter describes the functionality behind PlaGate, a tool for detecting similar source-code files, and investigating similar source-code fragments with a view to gathering evidence of plagiarism.

PlaGate is a hybrid model that can be integrated with external plagiarism detection tools. Most recent plagiarism detection tools are string-matching algorithms. The motivation behind PlaGate comes from the way the two algorithms (i.e. string-matching based and LSA) detect similar files.

String-matching tools have been shown to work well for detecting similar files, however, they are not immune to all types of plagiarism attacks [115]. For example, string-matching based systems use parsing based approaches and often source-code files are not parse-able or compilable. This situation causes files to be excluded from the detection comparison. Furthermore, the performance of string-matching based tools are known to suffer

from local confusion [115]. These tools compute the pair-wise similarity between files. PlaGate, is based on a different approach – it is language independent, files do not need to be parsed or compiled to be included in the comparison, and it is not affected by structural changes in the code (i.e. hence local confusion is not a problem).

8.2 Similarity in Source-code Files

Similarity between two files is not based on the entire files, but rather on the source-code fragments they contain. Investigation involves scrutinizing similar source-code fragments and judging whether the similarity between them appears to be suspicious or innocent. If a significant amount of similarity is found then the files under investigation can be considered suspicious.

As part of a survey conducted by Cosma and Joy [26], academics were required to judge the degree of similarity between similar source-code fragments and justify their reasoning. The survey revealed that it is common procedure during the investigation process, while academics compare two similar source-code fragments for plagiarism, to take into consideration factors that could have caused this similarity to occur. Such factors include:

- The assignment requirements — for example students may be required to use a specific data structure (e.g. vectors instead of arrays) to solve a specific programming problem;
- Supporting source-code examples given to students in class — these might include

skeleton code that they could use in their assignment solutions;

- Whether the source-code fragment in question has sufficient variance in solution — that is, whether the particular source-code fragment could have been written differently; and

- The nature of the programming problem and nature of the programming language — for example, some small object-oriented methods are similar, and the number of ways they can be written are limited.

All similarity between files must therefore be carefully investigated to determine whether it occurred innocently or suspiciously (i.e. due to plagiarism). The survey findings revealed that source-code plagiarism can only be proven if the files under investigation contain distinct source-code fragments and hence demonstrate the student's own approach to solving the specific problem. It is source-code fragments of this kind of similarity that could indicate plagiarism [26].

In addition, according to the survey responses, similar source-code fragments under investigation must not be "short, simple, trivial (unimportant), standard, frequently published, or of limited functionality and solutions" because these may not provide evidence for proving plagiarism. Small source-code fragments that are likely to be similar in many solutions can be used to examine further the likelihood of plagiarism, but alone they may not provide sufficient evidence for proving plagiarism. Section 8.3 describes the types of similarity found between files, and the kind of similarity that is considered suspicious.

Furthermore, the survey responses revealed that academics considered the following factors as evidence of similarity between source-code fragments [26].

- Lack of indentation, bad or identical indentation,
- Changed identifiers but same structure and logic the same,
- Program logic is the same,
- Source-code structure is the same,
- Same or same-bad source-code formatting, for example, both source-codes have the same location of white spaces between words,
- Similar source-code comments,
- The same number of lines and each line has the same functionality,
- Shared class variable,
- Similar mistakes in the two source-codes,
- Similarity between unusual lines of code, and
- Lexical, syntactical, grammatical and structural similarities.

8.3 Similarity Categories between Files and Source-Code Fragments

In student assignments two source-code fragments may appear similar, especially when they solve the same task, but they may not be plagiarised. When investigating files for plagiarism some source-code fragments would be used as stronger indicators than others, and from this perspective source-code fragments have varying contributions to the evidence gathering process of plagiarism. This Section presents a criterion for identifying the contribution levels of source-code fragments.

This criterion was developed after considering the findings from a survey discussed in Section 8.2 and in [26]. The survey was conducted to gather an insight into what constitutes source-code plagiarism from the perspective of academics who teach programming on computing subjects. A previous study revealed that similarity values between files and between source-code fragments are a very subjective matter [26], and for this reason the criterion developed consists of categories describing similarity in the form of levels rather than similarity values.

The *contribution levels (CL)* for categorising source-code fragments by means of their contribution toward providing evidence for indicating plagiarism are as follows.

- **Contribution Level 0 – No contribution.** This category includes source-code fragments provided by the academic that appear unmodified within the files under consideration as well as in other files in the corpus. Source-code fragments in this category

will have no contribution toward evidence gathering. Examples include skeleton code provided to students and code fragments presented in lectures or handouts.

- **Contribution Level 1 – Low contribution.** This category includes source-code fragments belonging to the *Contribution Level 0 – No contribution* category but which are modified in each file in which they occur in a similar manner. Examples include template code provided to students to structure their code, and source-code fragments which are coincidentally similar due to the nature of the programming language used. Source-code fragments in this category may only be used as low contribution evidence if they share distinct lexical and structural similarity.

- **Contribution Level 2 – High contribution.** The source-code fragments belonging to this category appear in a similar form only in the files under investigation, and share distinct and important program functionality or algorithmic complexity.

The similarity levels (SL) that can be found between files are as follows:

- **Similarity Level 0 – Innocent.** The files under investigation do not contain any similar source-code fragments, or contain similar source-code fragments belonging to the *Contribution Category 0: No contribution*, and *Contribution Level 1 – Low contribution* categories.

- **Similarity Level 1 – Suspicious.** The files under investigation share similar source-code fragments which characterise them as distinct from the rest of the files in the corpus. The majority of similar source-code fragments found in these files must belong

to the *Contribution Level 2 – High contribution* category although some belonging to *Contribution Level 0 – No contribution* and *Contribution Level 1 – Low contribution* categories may also be present and may contribute to the evidence. Enough evidence must exist to classify the files under investigation in this category.

8.4 PlaGate System Overview

The PlaGate tool aims to enhance the process of plagiarism detection and investigation. PlaGate can be integrated within external plagiarism detection tools with the aim of improving plagiarism detection (i.e. by increasing the number of similar file pairs detected) and to provide a facility for investigating the source-code fragments within the detected files.

The first component of PlaGate, PlaGate's Detection Tool (PGDT), is a tool for detecting similar files. This component can be integrated with external plagiarism detection tools for improving detection performance.

The second component of PlaGate, PlaGate's Query Tool (PGQT), is integrated with PGDT and the external tool for further improving detection performance. PGQT has a visualisation functionality useful for investigating the relative similarity of given source-code files or source-code fragments with other files in the corpus. In PlaGate, the source-code files and source-code fragments are characterised by LSA representations of the meaning of the words used. With regards to investigating source-code fragments, PGQT compares source-code fragments with source-code files to determine their degree of relative similarity

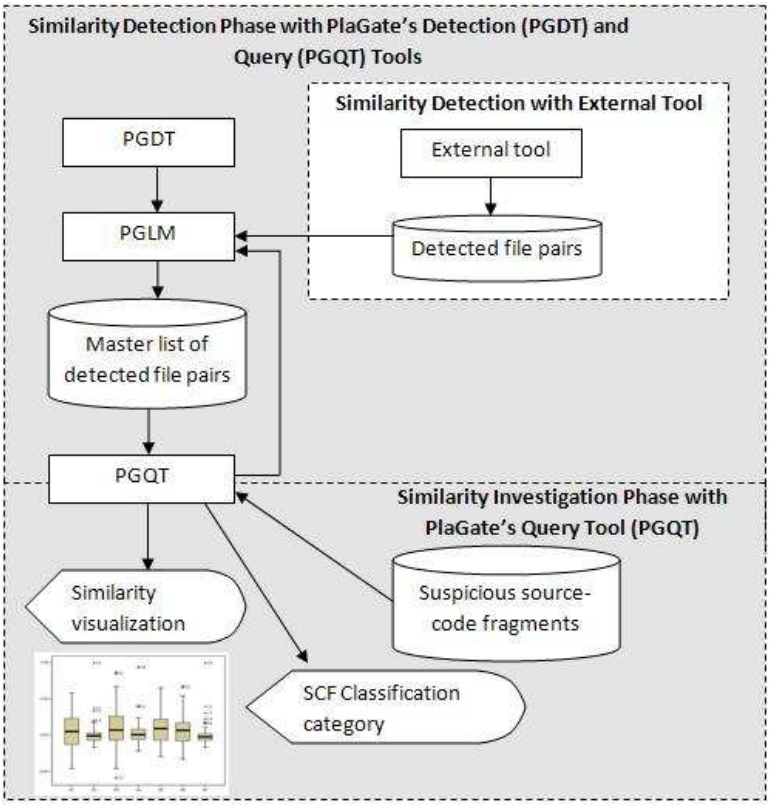

Figure 8.1: PlaGate's detection and investigation functionality integrated with an existing plagiarism detection tool.

to files. Files that are relatively more similar than others, contain distinct (i.e. contribution level 2) source-code fragments that can distinguish these files from the rest of the files in the corpus. The hypothesis is that similarity between the distinct source-code fragments and the files in which they appear will be relatively higher than the files that do not contain the particular source-code fragment, in the LSA space. From this it can be assumed that the relative similarity (i.e. importance) of a source-code fragment across files in a corpus can indicate its contribution level toward indicating plagiarism.

Figure 8.1 illustrates the detection functionality of PlaGate and how it can be integrated

with an external tool.

The similarity detection functionality of PlaGate is described as follows:

1. Similarity detection is performed using the external tool.
2. External tool outputs a list containing the detected file pairs based on a given cutoff value.
3. PlaGate's List Management component (PGLM) stores the detected files in the master list of detected file pairs.
4. Similarity detection is carried out with PGDT and similar file pairs are detected based on a given cutoff value.
5. PGLM stores the detected files in the master list of detected file pairs.
6. PGLM removes file pairs occurring more than once from the master list.
7. PGQT takes as input the first file, F_a, from each file pair stored in the master list.
8. PGQT treats each F_a file as a query and detects file pairs based on a given cutoff value.
9. PGLM updates the master list by including the file pairs detected by PGQT. Steps 5 and 6 are thereafter repeated one more time.

PGQT can also be used for visualising the relative similarity of source-code fragments and files. This procedure is described as follows:

1. PGQT accepts as input a corpus of source-code files[1] and source-code fragments to be investigated.

2. PGQT returns graphical output in the form of boxplots that indicate the contribution levels of source-code fragments.

3. PGQT returns the files most relative to the particular source-code fragment.

4. PGQT returns the category of contribution of the source-code fragment in relation to the files under investigation specified by the user.

PGQT can also be used for visualising the relative similarity between files. This procedure is described as follows:

1. PGQT accepts as input a corpus of source-code files[1], and accepts as queries the source-code files to be investigated (these could be files from the master list or any other file in the corpus selected by the academic).

2. PGQT returns output in the form of boxplots that indicate the relative similarity between the queries and files in the corpus.

3. PGQT returns the relative degree of similarity between files in the corpus.

[1]The source-code file corpus only needs to be input once either at the detection stage or at the investigation stage.

8.4.1 System representation

The following definitions describe the PlaGate components:

- A *file corpus* C is a set of source-code files

  $C = \{F_1, F_2, \ldots, F_n\}$

  where n is the total number of files.

- A *source-code fragment*, denoted by s, is a contiguous string of source-code extracted from a source-code file, F.

- A *source-code file* (also referred to as a file) F is an element of C, and is composed of source-code fragments, such that

  $F = \{s_1, s_2, \ldots, s_p\}$

  where p is the total number of source-code fragments found in F.

- A set of source-code fragments S extracted from file F is

  $S \subseteq F$.

- *File length* is the size of a file F, denoted by l_F, is computed by

  $l_F = \sum_{i=1}^{q} t_i$

  where t_i is the frequency of a unique term in F, and q is the total number of unique terms in F.

- *Source-code fragment length* is the size of a source-code fragment s, denoted by l_s, is computed by

$l_s = \sum_{i=1}^{u} t_i$

where t_i is the frequency of a unique term in s, and u is the total number of unique terms in s.

8.5 The LSA Process in PlaGate

The LSA process in PlaGate is as follows:

1. Initially a corpus of files is pre-processed, by removing from the files terms that were solely composed of numeric characters, syntactical tokens (e.g. semi-colons, colons), terms that occurred in less than two files (i.e. with global frequency less than 2), and terms with length less than 2. Prior experiments, described in Chapters 6 and 7, suggest that removing comments and Java reserved terms, and merging identifiers consisting of multiple words improves retrieval performance, and therefore those pre-processing parameters were employed in the PlaGate tool.

2. A pre-processed corpus of files into an $m \times n$ matrix $A = [a_{ij}]$, in which each row represents a term vector, each column represents a file vector, and each cell a_{ij} of the matrix A contains the frequency at which a term i appears in file j [16]. Each file F, is represented as a vector in the term-by-file matrix.

3. Term-weighting algorithms are then applied to matrix A in order to adjust the importance of terms using local and global weights and document length normalization. Our previous experiments conducted suggest that the LSA performs well for the task

of source-code similarity detection when the term frequency local weight, normal global weight and cosine document length normalisation were applied to our corpora. The tnc term weighting Function is integrated into PlaGate.

4. Singular Value Decomposition and dimensionality reduction are then performed on the weighted matrix A. Our experiments described in Chapter 6 revealed that for our datasets, setting k to 30 dimensions reported good results. Because we will test PlaGate using the same datasets described in Chapter 6, we have set k to 30 dimensions.

8.5.1 PGQT: Query processing in PlaGate

After computing LSA, an input file or source-code fragment (that needs to be investigated) is transformed into a query vector and projected into the reduced k-dimensional space. Given a query vector q, whose non-zero elements contain the weighted term frequency values of the terms, the query vector can be projected to the k-dimensional using Function 4.1 found in Section 4.3. Once projected, a file F, or source-code fragment s, is represented as a query vector Q.

The similarities between the projected query Q and all other source-code files in the corpus are computed using the cosine similarity. We have selected to implement the cosine measure in PlaGate, because, as already mentioned in Chapter 6, it is the most popular measure for computing the similarity between vectors and has been shown to produce good results. The cosine similarity involves computing the dot product between two vectors (e.g. the query vector and file vector) and dividing it by the magnitudes of the two vectors, as

described in Equation 4.2. This ratio gives the cosine angle between the vectors, and the closer the cosine of the angle is to +1.0 the higher the similarity between the query and file vector. Therefore, in the $m \times n$ term-by-file matrix A the cosine similarity between the query vector and the n file vectors is computed using Function 4.2 found in Section 4.3. Hence the output is a $1 \times n$ vector SV in which each element sv_i, contains the similarity value between the query and a file in the corpus,

$sv = sim(Q, F), \in [-1.0, +1.0]$

8.5.2 Source-code fragment classification in PlaGate

Common terms can exist accidentally in source-code files and due to the factors discussed in Section 8.2, this can cause relatively high similarity values between files in an LSA based system. For this reason criteria have been devised for classifying the source-code fragments into categories.

Each source-code fragment can be classified into a contribution level category based on the similarity value between the source-code fragment (represented as a query vector) and selected files under investigation (each file represented as a single file vector). This similarity is the cosine between the query vector and the file vector as discussed in Section 8.5.1. The classification of similarity values is performed as follows.

1. PlaGate retrieves the $sim(Q, F)$ similarity value between a query Q and the selected file F.

2. Based on the value of $sim(Q, F)$ PlaGate returns the classification category as contribution level 2 (if $sim(Q, F) \geq \phi$), otherwise contribution level 1.

The value of ϕ depends on the corpus, and setting ϕ to 0.80 is suitable for our datasets. This will be explained in detail in Chapter 10.

8.6 PGDT: Detection of Similar File Pairs in PlaGate

This Section describes the process of detecting pairs of similar files using the PlaGate detection tool. Treating every file as a new query and searching for similar files that match the query would not be computationally efficient or feasible for large collections. This is because each file must be queried against the entire collection. For this reason, instead of the cosine similarity algorithm described in Section 8.5.1, Function 8.1 will be used for computing the similarity between all files in the collection. After applying SVD and dimensionality reduction to the term-by-file matrix, A, the file-by-file matrix can be approximated by

$$(V_k \Sigma_k)(V_k \Sigma_k)^T. \tag{8.1}$$

This means that element i, j of the matrix represents the similarity between files i and j in the collection.

8.7 Visualisation of Relative Similarity in PlaGate

The output from PlaGate is visualised using boxplots. Boxplots (also called box-and-whisker plots) are graphical display methods for visualising summary statistics. The boxplot was originated by Tukey, who called it *schematic plot* [135].

To construct a boxplot, a SV vector (see Section 8.4.1) is initially ordered from smallest to largest $sv_1, \ldots, sv_n$ and then the median, upper quartile, lower quartile and interquartile range values are located. The median value is the middle value in a dataset. The *lower-quartile* (LQ) is the median value of the data that lie at or below the median. The *upper-quartile* (UQ) is the median value of the data that lie at or above the median. The box contains 50% of the data. The *inter-quartile range (IQR)* is the difference between the UQ and the LQ. The IQR is the range of the middle 50% of the data, and eliminates the influence of outliers because the highest and lowest quarters are removed. The *range* is the difference between the maximum and minimum values in the data.

A *mild outlier* is a value that is more than 1.5 times the IQR and an *extreme outlier* is a value that is more than 3 times the IQR. In the box-and-whisker plot the *mild outliers* are marked with a small circle (o) and the *extreme outliers* are marked with an asterisk (*).

Mild and extreme upper limit outliers are calculated as $ULmo = UQ - 1.5 \times (IQR)$ and $ULeo = UQ - 3.0 \times (IQR)$ respectively, and mild and extreme lower limit outliers are calculated as $LLmo = LQ - 1.5 \times (IQR)$ and $LLeo = LQ - 3.0 \times (IQR)$ respectively. *Adjacent values* are the highest and lowest values in the dataset within the limits, i.e.,

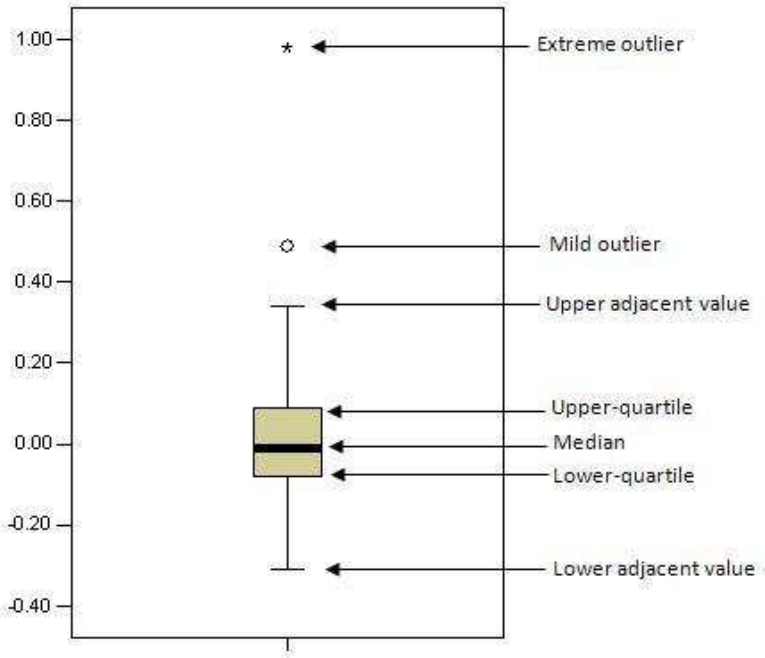

Figure 8.2: An example boxplot chart.

smaller than the lower limit and larger than the upper limit. If no outliers exist the adjacent values are the minimum and maximum data points. The whiskers extend from the LQ to the lowest adjacent value and from the upper-quartile to the highest adjacent value.

Using boxplots, the data output from PlaGate can be visualised and quickly interpreted. Boxplots help the user to identify clustering type features and enable them to discriminate quickly between suspicious and not suspicious files.

8.8 Design of Functional Components in PlaGate

The components of PlaGate in Figure 8.3 are described as follows:

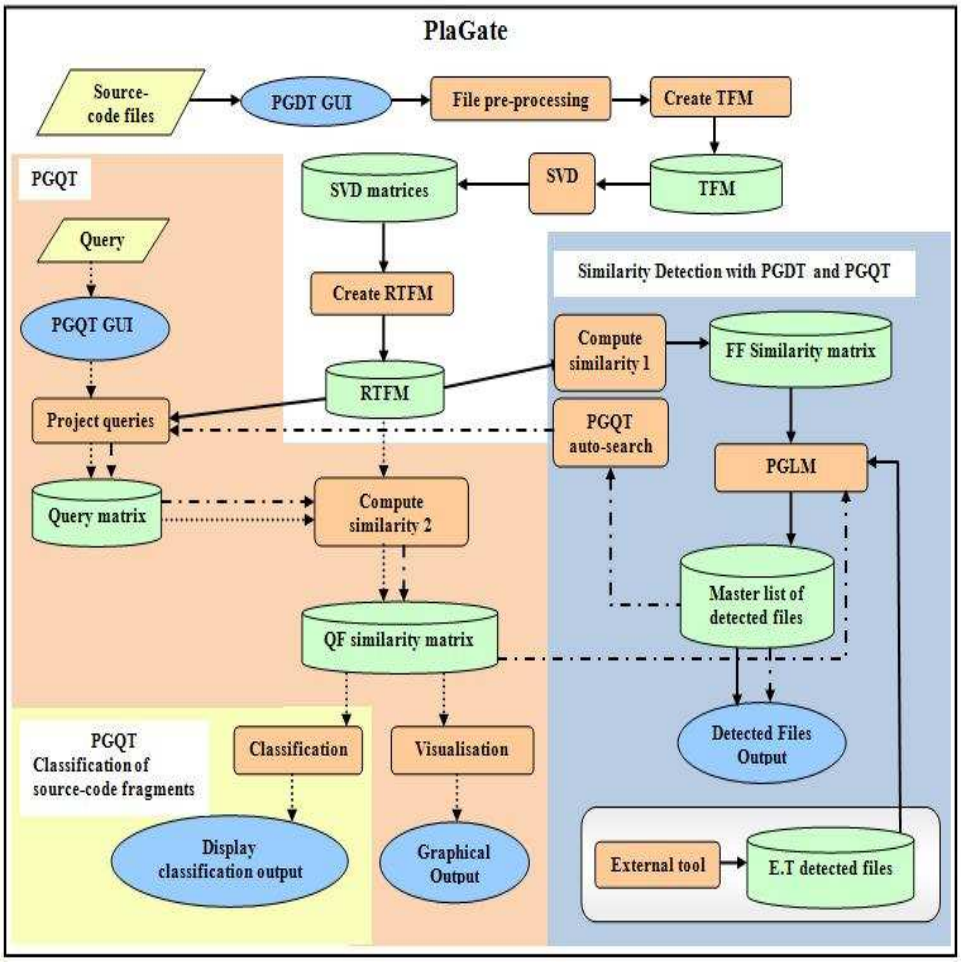

Figure 8.3: Design of Functional Components of the PlaGate Plagiarism Detection and Investigation System.

- *Source-code files:* these are the source-code files that comprise the corpus, stored on the users system.
- *PGDT GUI:* takes as input a corpus of source-code files.
- *File pre-processing:* this Function pre-processes the corpus.
- *Create TFM:* stores the pre-processed files into a term-by-file matrix (TFM). Term weighting and cosine document length normalisation algorithms are applied to the term-by-file matrix.
- *TFM:* the weighted term-by-file matrix.
- *SVD:* performs Singular Value Decomposition on the term-by-file matrix.
- *SVD matrices:* this database contains the term-by-dimension, file-by-dimension and singular values matrices generated after performing SVD.
- *Create RTFM:* this component computes the reduced term-by-file matrix.
- *RTFM:* the reduced term-by-file matrix created after performing SVD on TFM.
- *Compute similarity 1:* computes the pair-wise similarity values of file, using the reduced file-by-dimension and dimension-by-dimension matrices created from SVD.
- *FF similarity matrix:* holds the file-by-file matrix created from computing the Compute Similarity 1 Function.

- *PGLM:* is PlaGate's List Management component and takes as input the detected file pairs from PGDT, PGQT, and the external tool and creates a master list of detected file pairs.

- *PGQT auto-search:* takes as input the first file from each file pair detected by PGDT and the external tool and treats it as a query input into the PGQT tool.

- *Master list of detected file pairs:* contains all detected file pairs, i.e. those file pairs with similarity above a given cutoff value.

- *Query:* this is a source-code file or source-code fragment under investigation.

- *PGQT GUI:* takes as input one or more queries and a cutoff value by the user.

- *Project queries:* this process involves the projection of queries to the existing LSA space.

- *Query Matrix:* this is a matrix holding the projected query vectors.

- *Compute similarity 2:* This Function computes the cosine distance between each projected query vector and all file vectors in the RTFM.

- *QF similarity matrix:* this repository holds the cosine similarity values between the query and each file of the RTFM.

- *Classification:* this Function takes the values from the QF similarity matrix and classifies them into categories.

- *Display classification output:* this displays the classification category of each query (i.e. given source-code fragment).
- *Visualisation:* this component takes as input the vectors from the QF similarity matrix and for each vector creates a boxplot as a visualisation technique.
- *Graphical output:* this is a visual output containing box plots created by the Visualisation Function.
- *External Tool:* this is a plagiarism detection tool external to the PlaGate system.
- *E.T detected files:* these are the similar file pairs detected by the external tool.
- *Detected Files Output:* output showing the detected file pairs.

8.9 Conclusion

In this Chapter we discussed similarity in source-code files, and similarity between files from the perspective that suspicious source-code files can be identified by the distinct source-code fragments they share.

We proposed a tool, PlaGate, for source-code plagiarism detection and investigation with LSA, which can be integrated with external plagiarism detection tools.

Chapter 9 is concerned with evaluating the performance of PlaGate against two external tools, i.e. Sherlock and JPlag.

Chapter 9

Enhancing Plagiarism Detection with PlaGate

This Chapter is concerned with evaluating the similarity detection performance of PlaGate. The experiments described in this Chapter were conducted using four Java source-code datasets. The performance of PlaGate when applied alone on the source-code datasets, and when integrated with two external plagiarism detection tools, JPlag and Sherlock was evaluated.

9.1 Experiment Methodology

The experiment is concerned with evaluating the performance of PlaGate against two external plagiarism detection tools, Sherlock and JPlag. Detection performance when the tools

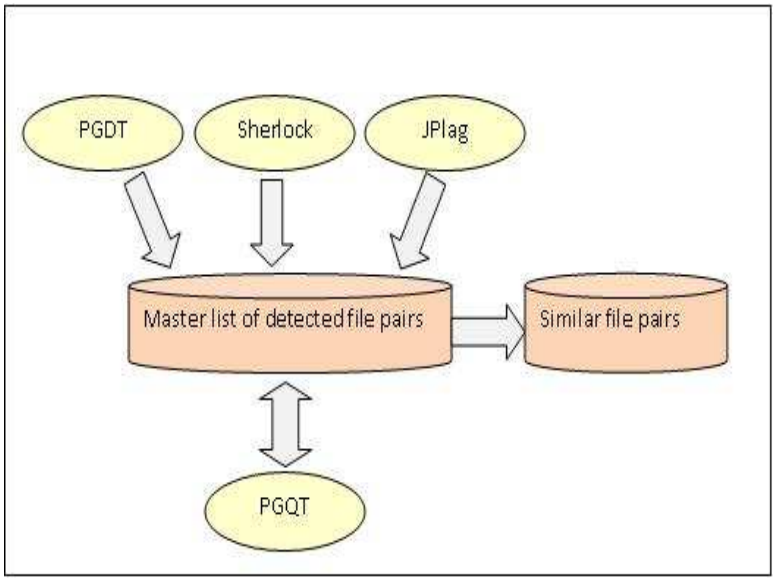

Figure 9.1: Methodology for creating a list of similar file pairs.

function alone and when integrated with PlaGate was evaluated. The evaluation measures recall and precision that are described in Section 4.10 were adapted for the task of evaluating similarity detection, as described in Section 9.2.

In order to evaluate the detection performance of tools using the evaluation measures discussed in Section 9.2, the total number of similar file pairs in each corpus must be known. Figure 9.1 illustrates the process of creating the list of similar file pairs.

Initially, tools PGDT, Sherlock, and JPlag were separately applied on each corpus in order to create a master list of detected file pairs for each corpus. For each corpus, three lists of detected file pairs (based on a given cutoff value as discussed in Section 9.2) were created, where each list corresponds to a tool output. These three lists were merged into a master list. Other known similar file pairs not identified by the tools were also included

in the master list of detected file pairs. Thereafter, PGQT was applied to every F_a file in the list (that is the first file in a pair of files) and file pairs with similarity value above ϕ_p were retrieved and added to the master list of detected file pairs. Academics scrutinised the detected file pairs contained in the master list and identified the similar file pairs. The final outcome consisted of four lists, i.e. one for each corpus, containing the grouped ID numbers of similar files.

After constructing the list of similar file pairs, the detection performance of each tool was evaluated using the evaluation measures discussed in Section 9.2. The detection results returned by each tool were based on a given cutoff value, this is also discussed in Section 9.2. In addition, detection performance is evaluated when PGQT is applied to a) all detected file pairs in the list, and b) only to those file pairs judged as similar by the academics. Results gathered from the second case will be marked with *PGQT v2* in the description of results.

Detection tools are expected to detect similar file pairs, and it would not be reasonable to penalise a system for detecting *similar* file pairs that are not suspicious. As described in Section 8.2 similarity found between files may or may not be suspicious. The transition in decision from similar to suspicious is one that the academic should make.

9.2 Performance Evaluation Measures for Plagiarism Detection

This Section describes evaluation of the comparative performances of PGDT, PGQT, JPlag and Sherlock. The similarity values provided by these three tools are not directly comparable since they use a different measure of similarity.

Two standard and most frequently used measures in information retrieval system evaluations are *recall* and *precision*, and these have been adapted to evaluate the performance of plagiarism detection.

The similarity for two files $sim(F_a, F_b)$ is computed using a similarity measure. Based on a threshold ϕ, the file pairs with $sim(F_a, F_b) \geq \phi$ are detected. For the purposes of evaluation the following terms are defined:

- *Similar* file pairs: each similar file pair, s, contains files that have been evaluated by human graders as similar. A set of similar file pairs is denoted by $S = \{s_1, s_2, \ldots, s_x\}$, where the total number of known similar file pairs in a set (i.e. corpus) is $x = |S|$.

- *Detected* file pairs: are those pairs that have been retrieved with $sim(F_a, F_b) \geq \phi$. The total number of detected file pairs is $|DF|$. The total number of similar file pairs detected is denoted by $|SD|$, and the total number of non-similar file pairs detected is denoted by $|NS|$. A set of detected file pairs is denoted by $DF = SD \cup NS = \{sd_1, sd_2, \ldots, sd_x\} \cup \{ns_1, ns_2, \ldots, ns_y\}$, where $SD \subseteq S$.

- PlaGate's cutoff value, ϕ_p, typically falls in the range $0.70 \leq \phi_p < 1.00$. Any file

pairs with $sim(F_a, F_b) \geq \phi_p$ are *detected*. The cutoff values for PGDT were selected experimentally. For each dataset, detection was performed using cutoff values $\phi_p = 0.70$ and $\phi_p = 0.80$. The cutoff value selected for each dataset was the one that detected the most similar file pairs and fewest false positives.

- JPlag's cutoff value is ϕ_j, where $0 < \phi_j \leq 100$, ϕ_j is set to the lowest $sim(F_a, F_b)$ given to a similar file pair sd_i detected by JPlag.

- Sherlock's cutoff value is ϕ_s, and is set to the top N detected file pairs. Sherlock displays a long list of detected file pairs sorted in descending order of similarity (i.e. from maximum to minimum). With regards to selecting Sherlock's cutoff value, each corpus was separately fed into Sherlock and each file pair in the returned list of detected file pairs was judged as similar or non-similar. The cutoff value is set to position N in the list of detected file pairs where the number of non-similar file pairs begins to increase. In computing Sherlock's precision $N = |DF|$.

Recall, denoted by R, is the proportion of similar file pairs that are detected based on the cutoff value, ϕ. Recall is 1.00 when all similar file pairs are detected.

$$Recall = \frac{|SD|}{|S|} = \frac{number\_of\_similar\_file\_pairs\_detected}{total\_number\_of\_similar\_file\_pairs} \tag{9.1}$$

where $R \in [0, 1]$.

Precision, denoted by P, is the proportion of similar file pairs that have been detected in the list of files pairs detected. Precision is 1.00 when every file pair detected is *similar*.

$$Precision = \frac{|SD|}{|DF|} = \frac{number\_of\_similar\_file\_pairs\_detected}{total\_number\_of\_file\_pairs\_detected} \tag{9.2}$$

where $P \in [0, 1]$.

The overall performance of each tool will be evaluated by combining the precision and recall evaluation measures. It is reasonable to penalise a system if it fails to detect similar file pairs, rather than when it detects false positives (i.e. non-similar), from the viewpoint that it increases the workload for the final judgment by the academic [115]. We consider it worthwhile to view some false positive (i.e. not-suspicious) file pairs if it means detecting some true positive (i.e. similar) file pairs in order to increase the number of plagiarism cases detected. For this reason we have assigned more weight to recall, hence the factor of 2 in equation 9.3. As a single measure for evaluating the performance of tools for plagiarism detection, the weighted sum of precision and recall will be computed by:

$$F = (Precision + 2 \times Recall)/3 \tag{9.3}$$

where $F \in [0, 1]$. The closer the value of F is to 1.00 the better the detection performance of the tool.

Table 9.1: The Datasets

	A	B	C	D
Number of files	106	176	179	175
Number of terms	537	640	348	459
Number of similar file pairs	6	48	51	79
Number of files excluded from comparison by JPlag	13	7	8	7

9.3 The Datasets

Four corpora comprising Java source-code files created by students were used for conducting the experiments. These were real corpora produced by undergraduate students on a Computer Science course at the University of Warwick. Students were given simple skeleton code to use as a starting point for writing their programs. Table 9.1 shows the characteristics of each dataset.

In Table 9.1, *Number of files* is the total number of files in a corpus. *Number of terms* is the number of terms found in an entire source-code corpus after pre-processing is performed as discussed in Section 8.5. *Number of similar file pairs* is the total number of file pairs that were detected in a corpus and judged as similar by academics.

JPlag reads and parses the files prior to comparing them. JPlag excludes from the comparison process files that cannot be read or do not parse successfully. In Table 9.1, the last row indicates the number of files excluded from comparison by JPlag. PlaGate and Sherlock do not exclude any files from the comparison process.

9.4 Experiment Results

Tables 9.2, 9.3, 9.4, and 9.5 show the results, for datasets A, B, C, and D, respectively, gathered from evaluating the detection performance of PGDT, Sherlock and JPlag on their own and when Sherlock and JPlag are integrated with PGDT and PGQT. The results are also illustrated in Figures 9.2, 9.3, 9.4, and 9.5. The discussion that follows corresponds to those Tables and Figures.

The results for dataset A show that PGDT outperformed Sherlock by means of precision, however Sherlock outperformed PGDT by means of recall. When Sherlock was integrated with PGDT (or PGDT and PGQT) the value of recall reached 1.00 which means that all similar file pairs were detected, however precision was lower due to the false positives detected by Sherlock. JPlag reported lower recall and precision values than PGDT, and recall and precision values both improved when JPlag was combined with PGDT and PGQT.

The F results for dataset B, show that Sherlock and JPlag both outperformed PGDT. JPlag's precision reached 1.00 meaning that no false positives were detected, however, JPlag performed poorly by means of recall. Sherlock returned high recall and precision values. Integrating the the external tools with PGDT and PGQT improved recall, but lowered precision.

With regards to dataset C, PGDT and JPlag both reported perfect precision, i.e. no false positives, however their recall was very low. Sherlock, had better recall than PGDT

and JPlag, however, it reported lower precision. Similarly, as with dataset's A and B, integrating the plagiarism detection tools with PGDT and PGQT improved recall, but lowered precision, compared to using the tools alone. The F results gathered from experiments performed on dataset C, show that Sherlock and JPlag outperformed PGDT. Integrating PGDT with Sherlock and JPlag increased detection performance compared to when using the external tools alone.

Finally, the F results gathered from experiments performed on dataset D, show that Sherlock and JPlag outperformed PGDT compared to using the tools alone. Performance improved when Sherlock was combined with PGDT, and when Sherlock was combined with PGDT and PGQT. JPlag's performance decreased when combined with PGDT and when integrated with PGDT and PGQT (or PGQTv2), compared to when using JPlag alone. A closer look at the values in Table 9.5 revealed that when combining JPlag with PGDT recall increased slightly, but precision decreased considerably, which indicates that many false positives were detected by PlaGate for this particular dataset.

With regards to the results gathered from all datasets, when PGQTv2 was used, precision increased than when using PGQT.

Table 9.6 holds the data on how recall and precision performance increases or decreases when PGDT, PGQT, and external plagiarism detection tools are integrated. Figure 9.6 shows the average recall values and average precision values over all datasets; and Figure 9.7 shows the average performance F over all datasets.

Table 9.2: Performance of tools on dataset A

	Recall	Precision	F
$PGDT\ \phi_p = 0.70$	0.67	0.80	0.71
$Sherlock$	0.83	0.25	0.64
$PGDT \cup Sherlock@20$	1.00	0.27	0.76
$PGDT \cup Sherlock@20 \cup PGQT$	1.00	0.23	0.74
$PGDT \cup Sherlock@20 \cup PGQTv2$	1.00	0.29	0.76
$JPlag\ \phi_j = 54.8$	0.50	0.75	0.58
$PGDT \cup JPlag$	0.67	0.67	0.67
$PGDT \cup JPlag \cup PGQT$	0.83	0.71	0.79
$PGDT \cup JPlag \cup PGQTv2$	0.83	0.83	0.83

Table 9.3: Performance of tools on dataset B

	Recall	Precision	F
$PGDT\ \phi_p = 0.80$	0.38	0.75	0.50
$Sherlock@40$	0.75	0.90	0.80
$PGDT \cup Sherlock@40$	0.83	0.83	0.83
$PGDT \cup Sherlock@40 \cup PGQT$	0.96	0.78	0.90
$PGDT \cup Sherlock@40 \cup PGQTv2$	0.96	0.82	0.91
$JPlag\ \phi_j = 99.2$	0.42	1.00	0.61
$PGDT \cup JPlag$	0.46	0.79	0.57
$PGDT \cup JPlag \cup PGQT$	0.54	0.79	0.62
$PGDT \cup JPlag \cup PGQTv2$	0.54	0.79	0.62

Table 9.4: Performance of tools on dataset C

	Recall	Precision	F
$PGDT\ \phi_p = 0.70$	0.23	1.00	0.49
$Sherlock\ \phi_s = 30$	0.51	0.87	0.63
$PGDT \cup Sherlock@30$	0.65	0.89	0.73
$PGDT \cup Sherlock@30 \cup PGQT$	0.65	0.89	0.73
$PGDT \cup Sherlock@30 \cup PGQTv2$	0.65	0.94	0.75
$JPlag\ \phi_j = 91.6$	0.37	1.00	0.58
$PGDT \cup JPlag$	0.57	1.00	0.71
$PGDT \cup JPlag \cup PGQT$	0.71	0.88	0.76
$PGDT \cup JPlag \cup PGQTv2$	0.71	1.00	0.80

Table 9.5: Performance of tools on dataset D

	Recall	Precision	F
$PGDT\ \phi_p = 0.70$	0.28	0.76	0.44
$Sherlock\ \phi_s = 50$	0.57	0.90	0.68
$PGDT \cup Sherlock@50$	0.70	0.82	0.74
$PGDT \cup Sherlock@50 \cup PGQT$	0.82	0.76	0.80
$PGDT \cup Sherlock@50 \cup PGQTv2$	0.82	0.90	0.85
$JPlag\ \phi_j = 100.0$	0.25	1.00	0.50
$PGDT \cup JPlag$	0.28	0.52	0.36
$PGDT \cup JPlag \cup PGQT$	0.30	0.55	0.38
$PGDT \cup JPlag \cup PGQTv2$	0.30	0.55	0.38

Table 9.6: Average change in performance of all datasets, when integrating PlaGate with plagiarism detection tools

	Increase/decrease of average recall	Increase/decrease of average precision
$PGDT \cup Sherlock$ vs Sherlock	0.13	-0.08
$PGDT \cup Sherlock$ vs PGDT	0.42	0.06
$PGDT \cup Sherlock \cup PGQT$ vs Sherlock	0.25	-0.14
$PGDT \cup Sherlock \cup PGQT$ vs PGDT	0.54	0.00
$PGDT \cup Sherlock \cup PGQTv2$ vs Sherlock	0.25	0.00
$PGDT \cup Sherlock \cup PGQTv2$ vs PGDT	0.54	0.14
$PGDT \cup JPlag$ vs JPlag	0.03	-0.48
$PGDT \cup JPlag$ vs PGDT	0.00	-0.24
$PGDT \cup JPlag \cup PGQT$ vs JPlag	0.05	-0.45
$PGDT \cup JPlag \cup PGQT$ vs PGDT	0.02	-0.21
$PGDT \cup JPlag \cup PGQTv2$ vs JPlag	0.05	-0.45
$PGDT \cup JPlag \cup PGQTv2$ vs PGDT	0.02	-0.21

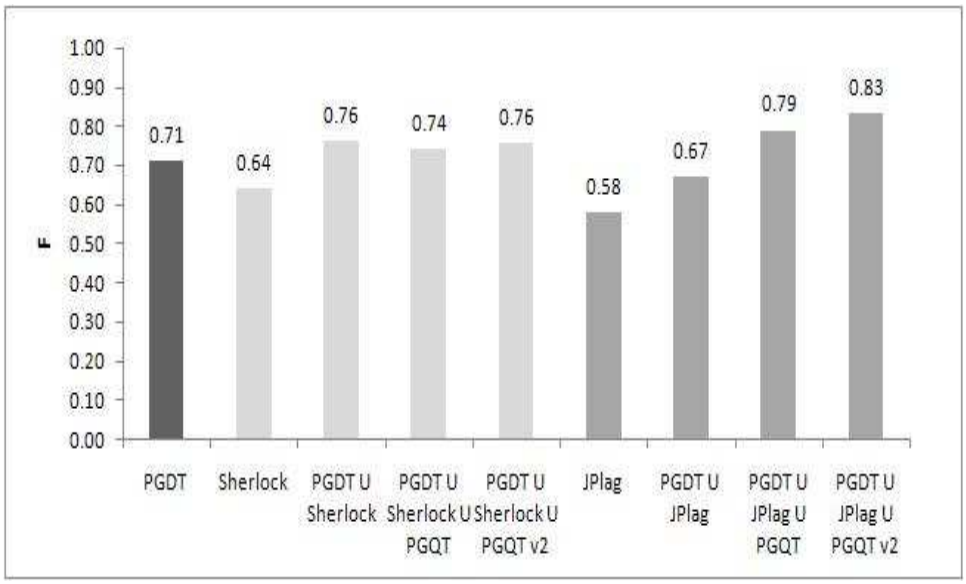

Figure 9.2: Dataset A evaluation. Cutoff values are PGDT $\phi_p = 0.70$, Sherlock $\phi_s = 20$, and JPlag $\phi_j = 54.8$.

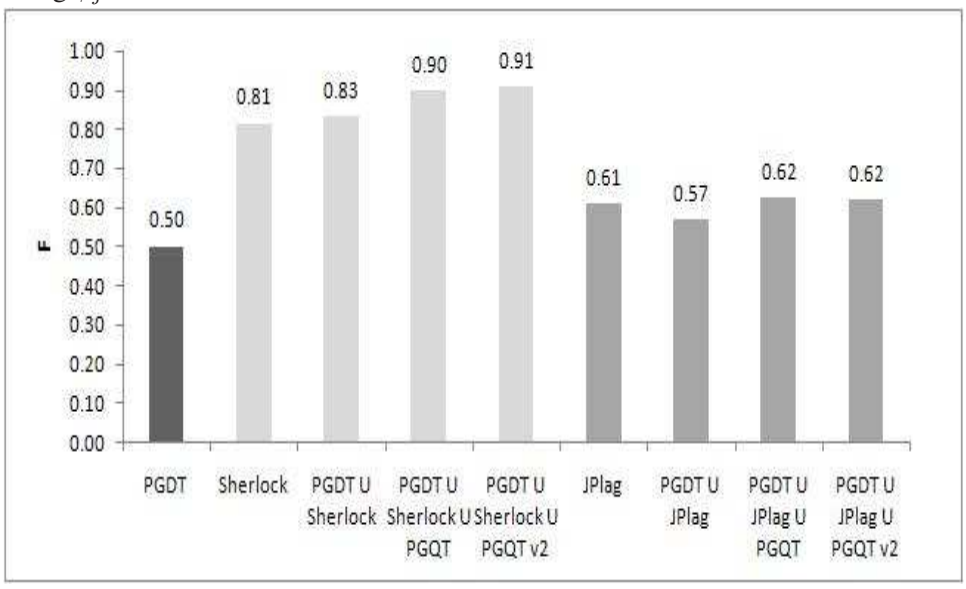

Figure 9.3: Dataset B evaluation. Cutoff values are PGDT $\phi_p = 0.80$, Sherlock $\phi_s = 40$, and JPlag $\phi_j = 99.2$.

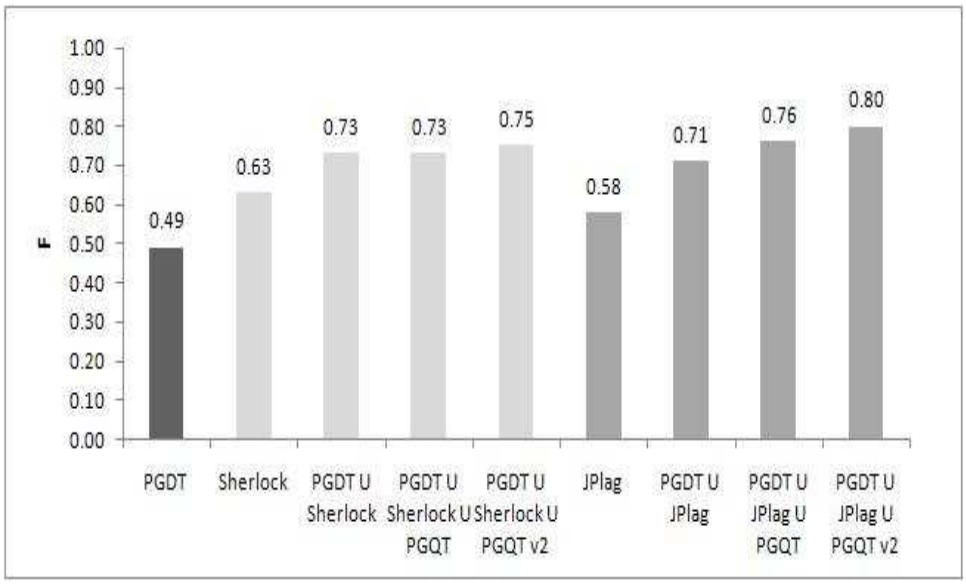

Figure 9.4: Dataset C evaluation. Cutoff values are PGDT $\phi_p = 0.70$, Sherlock $\phi_s = 30$, and JPlag $\phi_j = 91.6$.

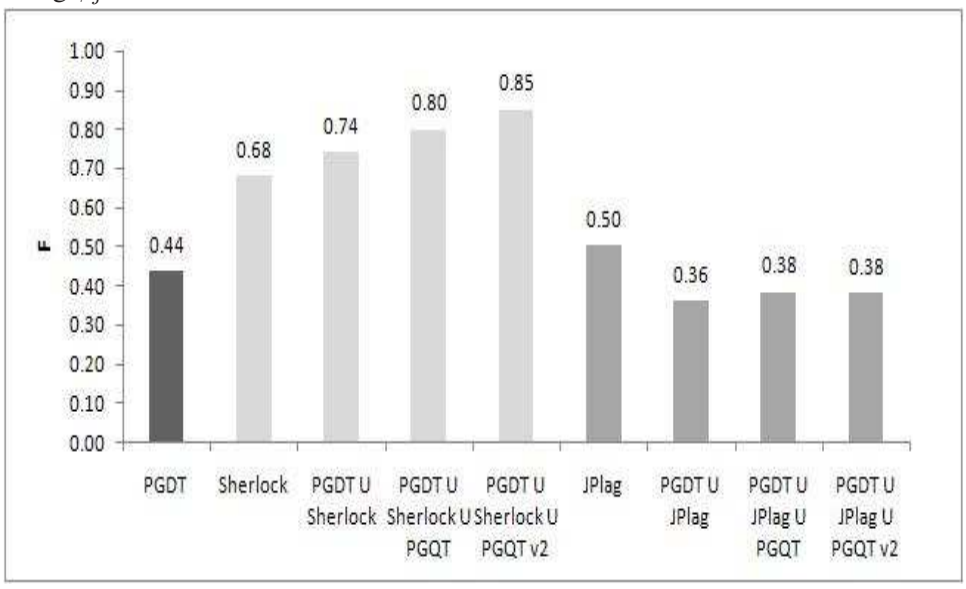

Figure 9.5: Dataset D evaluation. Cutoff values are PGDT $\phi_p = 0.70$, Sherlock $\phi_s = 50$, and JPlag $\phi_j = 100.0$.

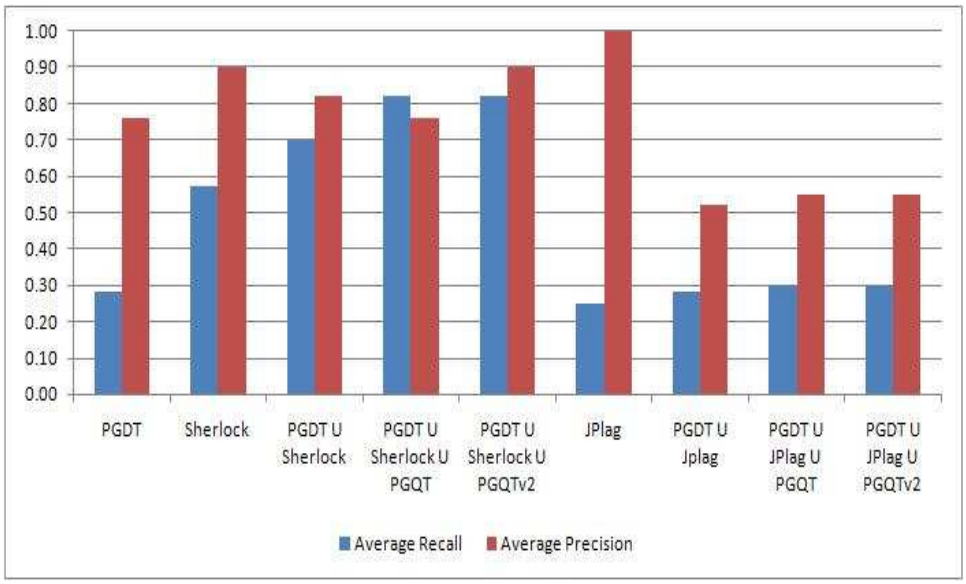

Figure 9.6: Average recall vs average precision performance of datasets A, B, C, and D.

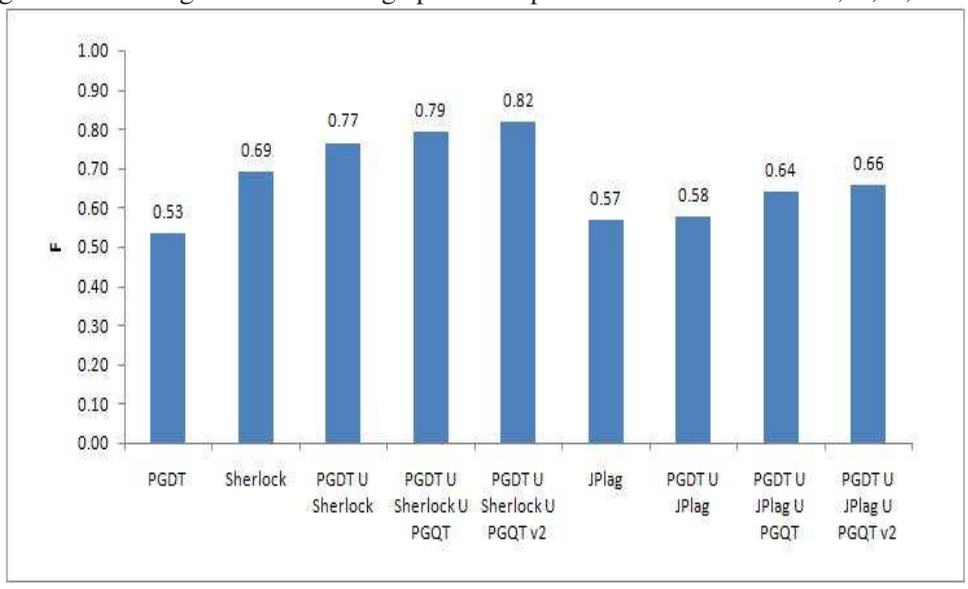

Figure 9.7: Average F performance of datasets A, B, C, and D.

Following is a summary of the results based on Tables 9.6, and Figures 9.6 and 9.7.

- Integrating external tools with PGDT increases the number of similar file pairs detected, i.e. improves recall. However, this integration also increases the number of non-similar file pairs detected (i.e. those non-similar file pairs which received a similarity value above the given cutoff value), and this caused a decrease in precision. Integrating Sherlock with PGDT, improves recall by 0.13 than when using Sherlock alone, and when PGDT is integrated with JPlag performance is improved by 0.03, however, precision suffers a decrease by -0.08 when Sherlock is concerned, and by -0.48 when JPlag is integrated with PGDT.

- When external tools are integrated with both PGDT and PGQT there is a further increase in recall that when external tools are used alone, i.e. an increase of 0.25 when Sherlock is integrated with PGDT and PGQT, and an increase of 0.05 when JPlag is concerned. However, there is also a further decrease in precision.

- There is slight improvement in overall F performance when PGQTv2 is applied rather that PGQT. PGQTv2 is applied only on files in file pairs judged as similar by the academics, and results show that precision is improved without compromising recall.

- With regards to dataset D, integrating PGDT and PGQT (or PGQT v2) with JPlag decreased overall performance by 0.12. Although the total number of similar file pairs detected was increased when JPlag was integrated with PGDT and PGQT or PGQTv2 (i.e. recall increased by 0.05), many non-similar file pairs were detected

which caused a large decrease in precision (i.e. from 1.00 to 0.45).

- Average results, shown in Figure 9.7, suggest that integrating PGDT and PGQTv2 with external tools improves overall detection performance.

Similarity often occurs in groups containing more than two files. Sherlock and JPlag often failed to detect some of the files from groups containing similar files. JPlag failed to parse some files that were similar to other files in a corpus (and they were excluded from the comparison process), and also suffered from local confusion which resulted in failing to detect similar file pairs [115]. Local confusion occurs when source-code segments shorter than the minimum-match-length parameter have been shuffled in the files. String matching algorithms tend to suffer from local confusion, which also appears to be the reason why Sherlock missed detecting similar file pairs. Sherlock often failed to detect similar file pairs because they were detected as having lower similarity than non-similar files (i.e. retrieved further down the list and lost among many false positives) mainly due to local confusion.

PlaGate's PGDT and PGQT do not suffer from local confusion because they are not based on detecting files based on structural similarity, hence they cannot be tricked by code shuffling. Furthermore, unlike JPlag, files do not need to parse or compile to be included in the comparison process. One issue with PlaGate, is that it detects more false positives than the external tools we experimented with, and hence the following question arises:

- How can we modify PlaGate such that less false positives are detected?

Overall results suggest that integrating PlaGate's PGDT and PGQT components with

external plagiarism detection tools improves overall detection performance.

9.5 Conclusion

This Chapter proposes a novel approach based on LSA for enhancing the plagiarism detection process. A tool PlaGate, has been developed that can function alone or be integrated with current plagiarism detection tools for detecting plagiarism in source-code corpora. The findings revealed that PlaGate can complement external plagiarism detection tools by detecting similar source-code files missed by them.

This integration resulted in improved recall at the cost of precision, i.e. more true positives but also more false positives in the list of detected files. In the context of source-code plagiarism detection, string-matching based detection systems (such as those discussed in Section 2.5.2) have shown to detect fewer false positives than an LSA based system. Overall performance was improved when PlaGate's PGDT and PGQT tools were integrated with the external tools JPlag and Sherlock.

Chapter 10

PlaGate: Source-Code Fragment Investigation and Similarity Visualisation

This Chapter is concerned with evaluating the performance of the investigation component of the PlaGate tool. This component of PlaGate computes the relative importance of source-code fragments within all the files in a corpus and indicates the source-code fragments that are likely to have been plagiarised. Evaluation of results is conducted by comparing PlaGate's results to human judgements.

10.1 Experiment Methodology

This Chapter applies the PGQT tool as a technique for detecting groups of similar files and investigating the similar source-code fragments within them. We evaluate PlaGate's PGQT tool to investigating the similarity between source-code files with regards to gathering evidence of plagiarism. The experiment described in this Chapter follows on from the experiment discussed in Chapter 9.

From the file pairs detected in datasets A and B in Chapter 9, four pairs of suspicious files were selected for investigation using the PlaGate tool. The file pairs containing the source-code fragments for investigation are, F113-F77, F75-F28, F82-F9, and F86-F32. The first file from each file pair (i.e. F113, F75, F82, and F86) was decomposed into source-code fragments at the level of Java methods. This resulted in thirty-four pairs of similar source-code fragments each consisting of one Java method, and these were treated as queries to be input into PGQT.

For evaluation purposes, academics with experience in identifying plagiarism graded each pair of source-code files and source-code fragments based on the criteria discussed in Section 8.3. Boxplots were created for each set of source-code fragments and arranged in groups, each containing boxplots corresponding to source-code fragments extracted from a single file.

Sections 10.2.1 and 10.2.2 discuss the visualisation output from detecting similar file pairs in datasets A and B. Suspicious and innocent file pairs have been selected as examples

for investigating the visualisation component of PlaGate's PGQT tool.

Section 10.3 discusses the visualisation output from investigating file similarity using source-code fragments from the suspicious file pairs in datasets A and B.

10.1.1 Visualisation output notes

As discussed in Section 8.5, when PGQT is used for plagiarism detection, a selected file F is treated as a query, it is expected that the particular file selected will be retrieved as having the highest similarity value with itself, and shown as an extreme outlier in the relevant boxplot. Therefore, if file F_1 is treated as a query, the file most similar to that will be the file itself (i.e., F_1). In the evaluation this similarity will be excluded from comparison. Boxplots were created for each of the files under investigation.

When PGQT is used for plagiarism investigation, the similarity value between a source-code fragment belonging to the contribution level 2 category (described in Section 8.3), and the file F from which it originated, is expected to be relatively high compared to the remaining files in the corpus. In addition, if the particular source-code fragment is distinct, and therefore important in characterising other files in the corpus, then these files will also be retrieved indicated by a relatively high similarity value.

With regards to source-code fragments belonging to contribution level 1, described in Section 8.3, because these are not particularly important in any specific files, it is usually not expected that they will retrieve the files they originated from. This is especially the case

if the files contain other source-code fragments that are more important in distinguishing the file from the rest of the files in the corpus.

The numbers shown in the boxplots correspond to the file identifier numbers, such that 9 corresponds to file 9, which is denoted as F9 in the analysis of the results.

10.2 Detection of Similar Source-Code Files using PGQT

This Section is a discussion of how the PGQT investigation tool can be used for detecting files that are similar to a file that is given as a query.

10.2.1 Dataset A

From dataset A, files F82, F75, F86, F45 and F73 were used as queries in PGQT. Figure 10.1 shows the output from PGQT which shows the relative similarity between these files and the rest of the files in the corpus.

Table 10.1 compares the similarity values by PGQT, and the human graders. Column PGQT holds the $sim(F_a, F_b)$ cosine similarity values for each pair containing files $F_a, F_b \in C$. Columns H1 SL and H2 SL show the similarity levels (SL) provided by human grader 1 (H1) and human grader 2 (H2) respectively.

Files F82, F75 and F86 were graded at similarity level 1 (suspicious) by the human graders. Their corresponding boxplots show that these files also received a high similarity value with their matching similar files. The files under investigation and their matching

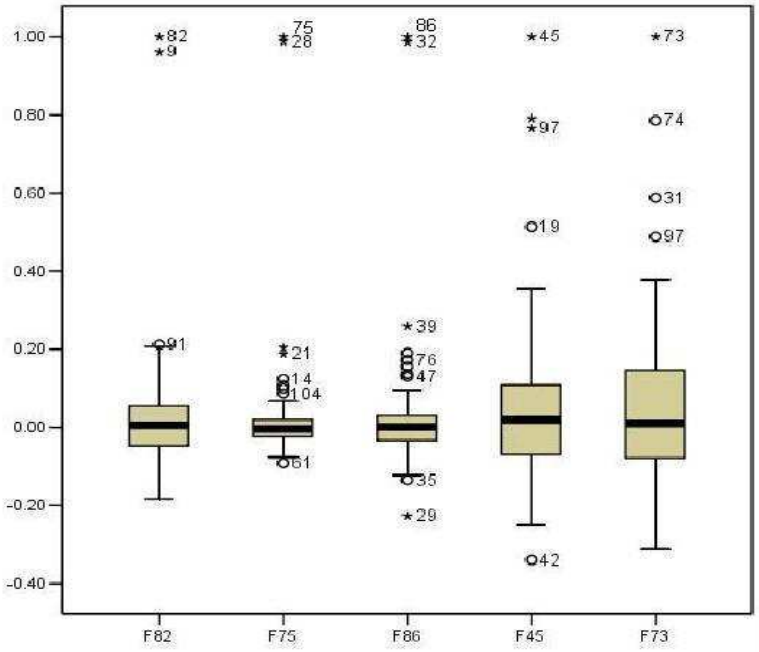

Figure 10.1: Dataset A: Similar files under investigation are F82-F9, F75-F28, F86-F32, F45-F97, F73-F39. Outliers $sim_i \geq 0.80$: F9-9,82; F75-75,28; F86-86,32.

Table 10.1: Dataset A file pairs and their similarity values

Files under investigation in dataset A, n=106			
File Pairs	PGQT	H1 LS	H2 LS
F9-F82	0.96	1	1
F75-F28	0.99	1	1
F86-F32	0.99	1	1
F45-F97	0.77	0	0
F73-F39	0.37	0	0

files were returned as extreme outliers with a large separation from the rest of the files in the corpus. These are indicators that files F82, F75 and F86 are suspicious. Table 10.2 contains the top five highest outliers for each of the files of dataset A.

Files F45 and F73 were graded at similarity level 0 (innocent) by the human graders. The boxplots in Figure 10.1 and Table 10.2 show that files F45 and F73 were returned in

Table 10.2: Dataset A files - Five highest outliers and their values

	F82	Value	F75	Value	F86	Value	F45	Value	F73	Value
1	F82	1.00	F75	1.00	F86	1.00	F45	1.00	F73	1.00
2	F9	0.96	F28	0.99	F32	0.99	F11	0.79	F74	0.79
3	F84	0.21	F12	0.20	F39	0.26	F97	0.77	F31	0.59
4	F4	0.21	F21	0.19	F102	0.19	F19	0.51	F97	0.49
5	F91	0.21	F14	0.12	F76	0.17	F20	0.35	F61	0.38

the top rank but without any other files receiving relatively close similarity values to them, because they were used as the query files. In addition, without taking into consideration outlier values, the data for F45 and F73 are spread out on a relatively large scale when compared to the boxplots corresponding to files F82, F75 and F86, and without much separation between the files under investigation and the remaining of the files in the corpus. These are indicators that files F45 and F73 are innocent.

10.2.2 Dataset B

From dataset B, files F173, F113, and F162 were used as queries in PGQT. Figure 10.2 shows the output from PGQT, illustrating the relative similarity between these files and the rest of the files in the corpus. Table 10.4 contains the top five highest outliers for each of the files of dataset B.

Table 10.3: Dataset B File pairs and their similarity values

Files under investigation in Dataset B, n=176			
File Pairs	PlaGate	H1 LS	H2 LS
F173-F142	1.00	1	1
F113-F77	0.64	1	1
F162-F20	0.38	1	1
F162-F171	0.50	1	1

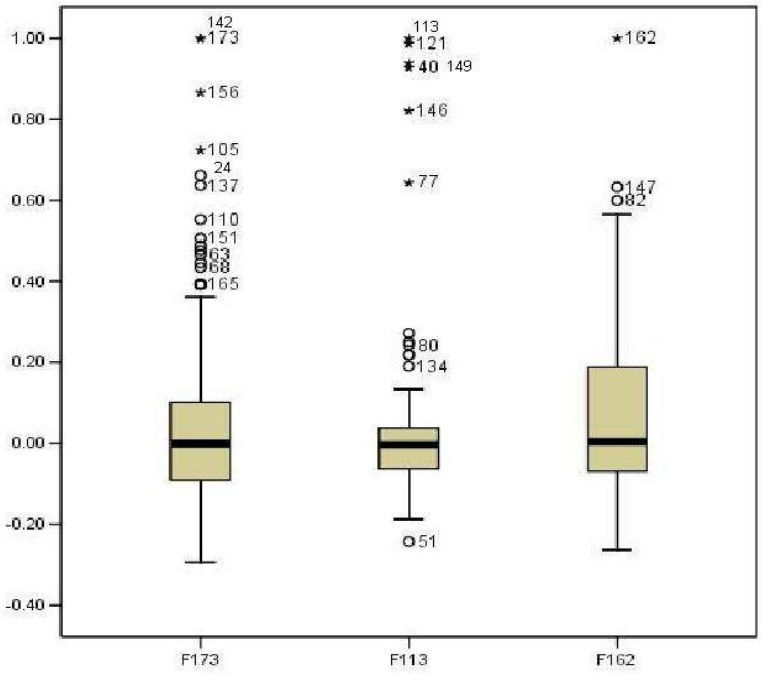

Figure 10.2: Dataset B: Similar files under investigation are F173-F142,F113-F77, F162-F20-F171. Outliers $sim_i \geq 0.80$: F173-173,142,156; F113-113,121,149,40,146

Table 10.4 shows that PlaGate detected file F142 as highly similar to file F173 and thus file pair F173-F142 has received high similarity values. Both JPlag and PlaGate have also detected this file pair as suspicious. The boxplot for F173 shows that the similarity values are spread out with no great separation between files, which indicates that files F142 and F173 are relatively not that unique in relation to the remaining files in the corpus. However it is worth investigating those two files because they were detected as similar and there is a bit of separation between them and the rest of the files. Comparing the source-code within files F142 and F173 revealed that both of these files are relatively short, mainly containing skeleton code provided by academics, but they also share some distinct source-code fragments (not shared by other files in the corpus) that are considered as belonging to the

contribution level 2 category. The boxplot observation matches the academic judjements. The similar source-code fragments (at granularity level of Java methods) found in these files are small, consisting of 3 to 4 small lines of code each and therefore investigation into each of these using the PGQT investigation tool is not necessary. Files F173-F156 share some similarity but it is clear from observing these files that the similarity was trivial and unintentional, due to the programming language used and the problem been solved.

Table 10.3 shows that the human graders consider files F113 and F77 to be suspicious (similarity level 1). The boxplot corresponding to file F113 shows that, excluding outliers, the values are spread on a relatively small scale, the IQR is small, and there exists a large separation between the suspicious and non-suspicious files. These are indicators that the files are suspicious, and an investigation into the similarity between F113 and all extreme outliers, i.e. F113, F121, F149, F40, F146, and F77, revealed that all those files are very similar to file F113 and all contain matching similar source-code fragments.

Table 10.4: Dataset B files - Five highest outliers and their values

	F173	Value	F113	Value	F162	Value
1	F173	1.00	F113	1.00	F162	1.00
2	F142	1.00	F121	0.99	F147	0.63
3	F156	0.87	F149	0.94	F82	0.60
4	F105	0.72	F40	0.93	F163	0.56
5	F24	0.66	F146	0.82	F80	0.56

Regarding the group containing suspicious files F162, F20, and F171, PlaGate has failed to retrieve these, however, JPlag has detected these suspicious files.

Below is the source-code for files F113 and F121, extracted from corpus B. This file pair

was detected by PlaGate and failed to get detected by JPlag and Sherlock. We have added extra comments within the files to indicate the source-code fragments and their contribution level category that will be used passed into PlaGate for investigation (as described in Section 10.3 and the results are also illustrated in Figure 10.6).

File F113:

```
package student;

import wrabble.*;

public class Board implements IBoard
{
TileBagboard tileboard;
private Tile boardArray[][];
private Multiplier multipliers[][];

public Board()
{
setBoardSize(15);
}

%-----------------------------------------------
Method clearBoard() is a contribution level 1
source-code fragment
%-----------------------------------------------

public void clearBoard()
{
for(int i =0;i <boardArray.length;i++)
{
for(int n =0;n <boardArray.length;n++)
boardArray[i][n] = null;
}
}

public Tile getTile(int x,int y)
{
return boardArray[x][y];
}

public boolean setTile(int x,int y,Tile tile)
{
boardArray[x][y] = tile;
return true;
}
```

```
public Multiplier getMultiplier(int x,int y)
{
return multipliers[x][y];
}

public void setMultiplier(int x,int y,Multiplier multiplier)
{
multipliers[x][y] = multiplier;
}

public int getBoardSize()
{
return boardArray.length;
}

public void setBoardSize(int size)
{
boardArray = new Tile[size][size];
multipliers = new Multiplier[size][size];
}

%-----------------------------------------------
Method getWords() is a contribution level 2
source-code fragment
%-----------------------------------------------
public String[] getWords()
{
tileboard = new TileBagboard(1);
String string = "";

for(int i =0;i <getBoardSize();i++)
{
for(int n =0;n <getBoardSize();n++)
{
Tile tile = getTile(i,n);
if(tile != null)
{
string = string + tile.getLetter();
}
else
{
```

```
if(string.length() >= 2)
{
string=string.trim().toLowerCase();
tileboard.addElements(string);
}
string = "";
}
}
}

for(int a =0; a < getBoardSize(); a++)
{
for(int b = 0; b < getBoardSize(); b++)
{
Tile tile1 = getTile(b, a);
if(tile1 != null)
{
string = string + tile1.getLetter();
}
else
{
if(string.length() >= 2)
{
string=string.trim().toLowerCase();
tileboard.addElements(string);
}
string = "";
}
}
return tileboard.StringArray();
}
}

class TileBagboard
{
String wordboardlist1[];
int noofElements;

public TileBagboard(int capacity)

{
wordboardlist1 = new String[capacity];
```

```
noofElements = 0;
}

%-----------------------------------------------
Method addElements() is a contribution level 2
source-code fragment
%-----------------------------------------------

public void addElements(String str)
{
if (noofElements == wordboardlist1.length)
{
String[] wordboardlist2 = new String[wordboardlist1.length*2];

for (int i=0; i<wordboardlist1.length; i++)
{
wordboardlist2[i]=wordboardlist1[i];
}

wordboardlist1=wordboardlist2;
}
wordboardlist1[noofElements] = str;
noofElements++;
}


public String elementAt(int i)
{
return wordboardlist1[i];
}

public int size()
{
return noofElements;
}

%-----------------------------------------------
Method StringArray() is a contribution level 2
source-code fragment
%-----------------------------------------------

public String[] StringArray()
```

```
{
String[] newArray = new String[noofElements];

for(int i =0;i <noofElements;i++)
{
newArray[i]=elementAt(i);
}
return newArray;
}
}
```

File F121

```
package student;

import wrabble.*;

public class Board implements IBoard
{
TileBagboard tileboard;
private Tile tilearray[][];
private Multiplier multiply[][];
int Boardwidth=15;

public Board()
{
setBoardSize(Boardwidth);
}

%-----------------------------------------------
Method clearBoard() is a contribution level 1
source-code fragment
%-----------------------------------------------

public void clearBoard()
{
for(int y =0;y <tilearray.length;y++)
{
for(int x =0;x <tilearray.length;x++)
tilearray[y][x] = null;
}
}

public wrabble.Tile getTile(int x,int y)
{
if ( (x > 0 &&  x<15) && ( y >0 && y < 15)){
if(tilearray[x][y]!=null)
return tilearray[x][y];
else
return null;
}
else
return null;
```

```
}

public boolean setTile(int x,int y, wrabble.Tile tile)
{
if ( ( x >-0 && x <15 ) && ( y > -0 && y <15)) {
tilearray[x][y] = tile;
return true;
}
else
return false;
}

public Multiplier getMultiplier(int x,int y)
{
return multiply[x][y];
}

public void setMultiplier(int x,int y,wrabble.Multiplier multiplier)
{
multiply[x][y] = multiplier;
}

public int getBoardSize()
{
return tilearray.length;
}

public void setBoardSize(int size)
{
tilearray = new Tile[size][size];
multiply = new Multiplier[size][size];
}

%-----------------------------------------------
Method getWords() is a contribution level 2
source-code fragment
%-----------------------------------------------

public String[] getWords()
{
tileboard = new TileBagboard(1);
String string = "";
```

```
for(int i =0;i <getBoardSize();i++)
{
for(int j =0;j < getBoardSize();j++)
{
Tile tile = getTile(i,j);
if(tile != null )
{
string = string + tile.getLetter();
}
else
{
if(string.length() >= 2)
{
string=string.trim().toLowerCase();
tileboard.addElements(string);
}
string = "";
}
}
}

for(int a =0; a < getBoardSize(); a++)
{
for(int b = 0; b < getBoardSize(); b++)
{
Tile tile1 = getTile(b, a);
if(tile1 != null)
{
string = string + tile1.getLetter();
}
else
{
if(string.length() >= 2)
{
string=string.trim().toLowerCase();
tileboard.addElements(string);
}
string = "";
}
}
}
return tileboard.StringArray();
```

```
}
}

class TileBagboard
{
String wordboardlist1[] = null;
int Elements;

public TileBagboard(int cap)
{
wordboardlist1 = new String[cap];
Elements = 0;
}

%-----------------------------------------------
Method addElements() is a contribution level 2
source-code fragment
%-----------------------------------------------

public void addElements(String str)
{
if (Elements == wordboardlist1.length)
{
String[] wordboardlist2 = new String[wordboardlist1.length*2];
for (int i=0; i<wordboardlist1.length; i++)
{
wordboardlist2[i]=wordboardlist1[i];
}

wordboardlist1=wordboardlist2;
}
wordboardlist1[Elements] = str;
Elements++;
}

public String elementAt(int i)
{
return wordboardlist1[i];
}

public int size()
{
```

```
return Elements;
}

%-----------------------------------------------
Method StringArray() is a contribution level 2
source-code fragment
%-----------------------------------------------

public String[] StringArray()
{
String[] newarray = new String[Elements];
for(int i =0;i <Elements;i++)
{
newarray[i]=elementAt(i);
}
return newarray;
}
}
```

10.3 Datasets A and B: Investigation of Source-code Fragments using PGQT

After the similarity detection stage, similar source-code fragments in files F82, F75 and F86 of dataset A, and file F113 of dataset B were investigated. From these files, source-code fragments from both contribution level 1 (low contribution) and contribution level 2 (high contribution) categories were selected as described in the methodology in Section 10.1. Each individual boxplot was placed in one of three sets of boxplot graphics, corresponding to the file it originated from. The boxplots of source-code fragments corresponding to files F82, F75, F86 and F113 are shown in Figures 10.3, 10.4, 10.5, and 10.6 respectively. Table 10.5 shows the contribution levels assigned to the source-code fragments by the human graders.

In the case of the source-code fragments extracted from file F82 (Figure 10.3), the boxplots corresponding to Q1, Q2, Q9 and Q10 all received very high similarity values with both files F82 and F9. Source-code fragments Q1, Q2, Q9 and Q10 were classified as belonging to the contribution level 2 category by the human graders (Table 10.5). The boxplot of Q3 has many mild outliers but there is no great separation between the files in the corpus, and no extreme outliers exist in the dataset. These are strong indicators that source-code fragment Q3 is not particularly important within files F9 and F82 or any other files in the corpus. Q5 has one mild outlier and no great separation exists between files. Excluding outliers, the data for the contribution level 1 source-code fragments Q3, Q4, Q5, Q6, Q7,

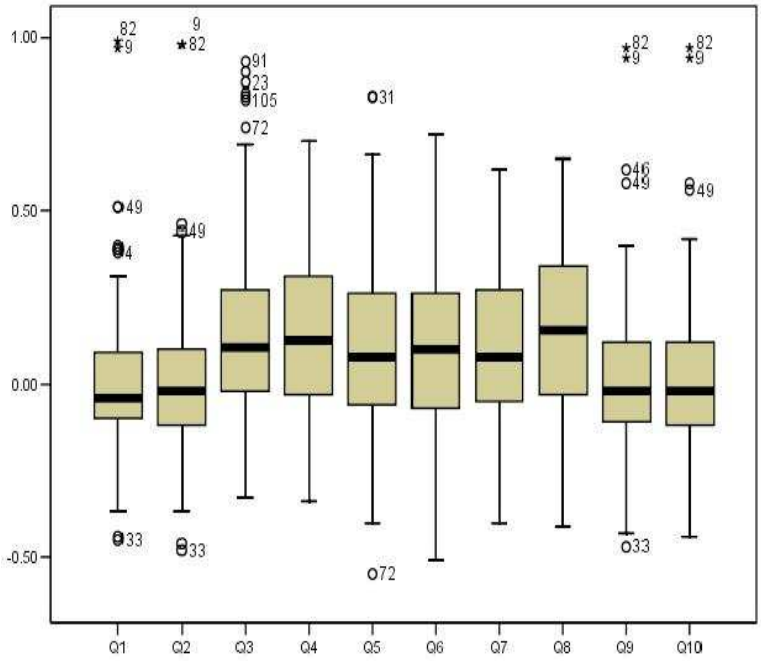

Figure 10.3: F82 boxplot: Similar files under investigation are F82 and F9. Outliers $sim_i \geq 0.80$: Q1 - 82, 9; Q2 - 9, 82; Q3 - 91, 54, 23, 26, 104, 105, 94; Q5 - 22, 31; Q9 - 9, 82; Q10 - 9,82.

and Q8 is spread on a relatively wider scale than data of contribution level 2 source-code fragments. The boxplots corresponding to contribution level 1 source-code fragments do not provide any indications that they are important in any of the files. As shown in Table 10.5, the observations from boxplots match the academic judgements.

Figure 10.4 shows the data for the source-code fragments corresponding to file F75. The boxplots show that Q2, Q4, and Q7 received very high similarity values with files F75 and F28. Table 10.5 shows that source-code fragments Q2, Q4, and Q7 were classified at the contribution level 2 category. The remaining source-code fragments – Q1, Q3, Q5 and Q6 – were graded at contribution level 1 by the human graders. Source-code fragment Q6

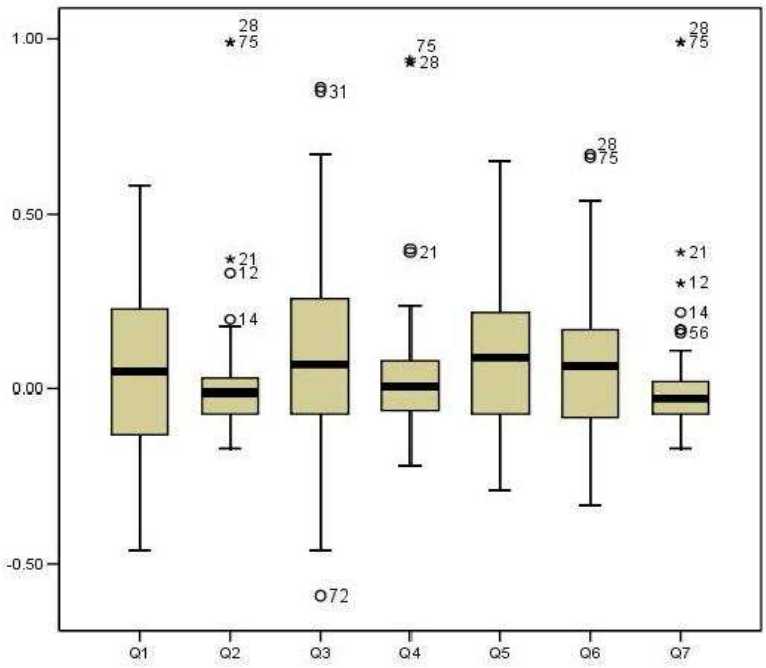

Figure 10.4: F75 boxplot: Similar files under investigation are F75 and F28. Outliers $sim_i \geq 0.80$: Q2 - 75, 28; Q3 - 22, 31; Q4 - 75, 28; Q7 - 75, 28.

has two mild outliers, files F75 and F28, which suggests that source-code fragment Q6 is relatively more similar to those two files than others in the corpus, however there is no great separation between the files in the corpus and hence it can not be considered as particularly important fragment in files F75 and F28.

The boxplots corresponding to source-code fragments extracted from file F86 shown in Figure 10.5 and from file F113 shown in Figure 10.6 follow a similar pattern. In Figure 10.6, the boxplots corresponding to source-code fragments Q1, Q2, and Q4, show files F113 and F77 as extreme outliers with a good separation from the rest of the files. In addition, the remaining files presented as extreme outliers in Q1, Q2 and Q4 are very similar to files F113 and F77 and all contain matching similar source-code fragments to Q1, Q2, and Q4.

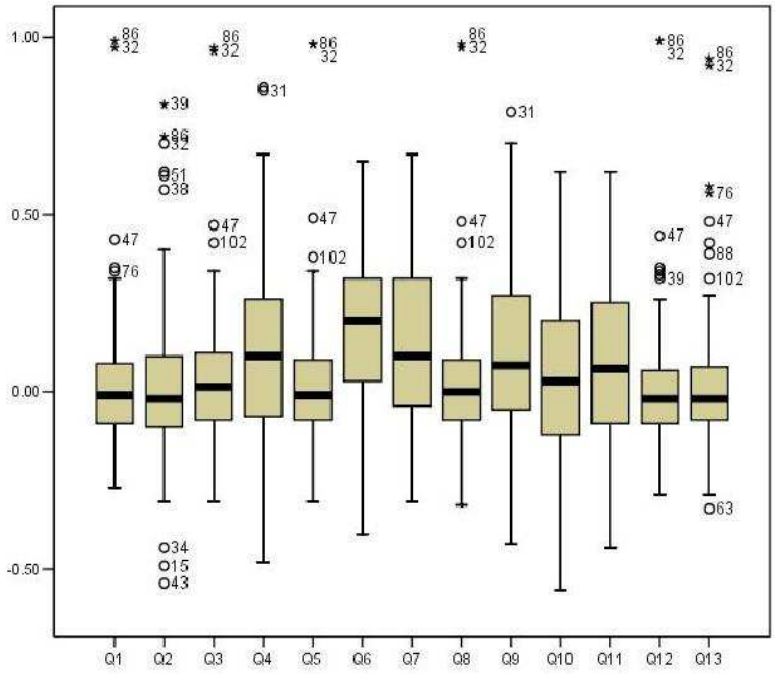

Figure 10.5: F86 boxplot: Similar files under investigation are F86 and F32. Outliers $sim_i \geq 0.80$: Q1 - 86,32; Q2 - 39; Q3 - 86, 32; Q4 - 86, 85; Q5 - 86, 32; Q8 - 86, 32; Q12 - 86, 32; Q13 - 86, 32.

During the detection process JPlag only identified files F113 and F77 as similar and failed to parse the remaining files in this group of similar files. Sherlock performed better than JPlag and detected three files. PGDT outperformed JPlag and Sherlock and detected five of those files. By using PGQT to investigate similar source-code fragments seven similar files were detected and scrutinising these revealed that they all shared suspicious source-code fragments. The suspicious source-code fragments are Q1, Q2, and Q4 shown in Figure 10.6.

In summary, the boxplots follow either *Pattern 1* or *Pattern 2*. These are discussed as follows:

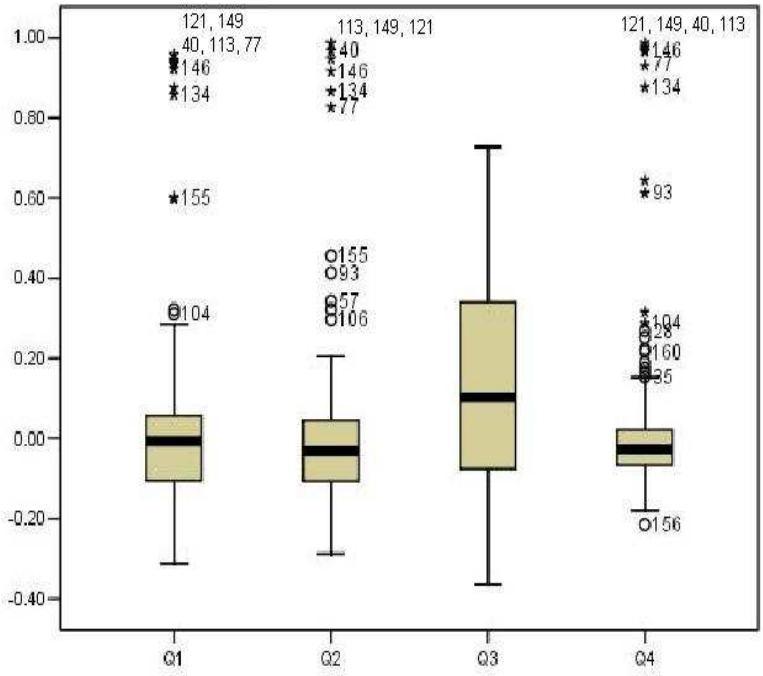

Figure 10.6: F113 boxplot: Similar files under investigation are F113 and F77. Outliers with $sim_i \geq 0.80$: Q1 - 121, 149, 40, 113, 146, 134, 77; Q2 - 121, 149, 40, 113, 146, 134, 77; Q4 - 121, 149, 40, 113, 146, 134, 77. Q1, Q2 and Q4 are CL2 SCFs, Q3 is a CL1 SCF.

- Pattern 1: Source-code fragments belonging to contribution level 1 category
  - The values in boxplots corresponding to source-code fragments categorized as contribution level 1 by academics are spread out on the scale without any files having great separation than others.
  - The IQR of box plots corresponding to contribution level 1 source-code fragments is relatively higher than those corresponding to contribution level 2 source-code fragments.
- Pattern 2: Source-code fragments belonging to contribution level 2 category

- The boxplots for the contribution level 2 source-code fragments show the suspicious files marked as extreme outliers with a good separation from the rest of the files in the corpus.
- The upper-quartile values for the source-code fragments belonging to the contribution level 2 category are either equal or below the median of the source-code fragments falling into the contribution level 1 category. That is, over three-quarters of the values in the contribution level 2 datasets are lower than the median values of the contribution level 1 datasets.
- Some of the boxplots of source-code fragments belonging to the contribution level 1 category have whiskers extending to high on the scale and/or mild outliers. This suggests that in some cases the single pair-wise similarity values between source-code fragments and files may not be sufficient indicators of the source-code fragments' contribution towards indicating plagiarism.
- Excluding the outliers, and therefore considering the IQR and difference between maximum and minimum adjacent values, data for source-code fragments belonging to the contribution level 2 category are spread out on a smaller scale compared to the data for contribution level 1 source-code fragments.

10.3.1 Comparing PGQT's results against human judgements

This Section describes and explores hypotheses formed from analyzing the output of PGQT in Section 10.3. Table 10.5 shows the source-code fragments and their similarity values to

files. Column $sim_1 = sim(Q, F_a)$ and $sim_2 = sim(Q, F_b)$ are the cosine similarity values between each source-code fragment Q and files F_a and F_b where F_a corresponds to the first file in a pair of files, and F_b corresponds to the second file, and $F_a, F_b \in C$. Column m_sim is the average of sim_1 and sim_2, column *PGQT CL* shows the contribution level values based on the m_sim values (contribution levels are described in Section 8.5.2), column IQR shows the IQR values computed from the boxplot as described in Section 8.7, and *Human CL* is the contribution level provided by the academics based on the criterion in Section 8.3.

Table 10.5 shows the source-code fragments and their similarity values to files, and is described in Section 10.3.1.

Observing the similarity values provided by LSA and comparing those to the similarity levels provided by the human graders, revealed a pattern – classifying the similarity values provided by LSA (thus creating LSA CL values) to reflect the categories used by human graders (i.e. CL1 or CL2), revealed that LSA values matched those of humans when setting threshold value ϕ (as described in Section 8.5.2) to 0.80. More detail on the relevant experiments is described later on in this Chapter. Thus, in the discussion that follows, we will set the value of ϕ to 0.80.

The hypotheses that will be tested are as follows:

Hypothesis 1:

- H_0: There is no correlation between the PGQT similarity values (i.e., sim_1, sim_2, m_sim)

Table 10.5: Similarity values for all source-code fragments and their suspicious file pairs

Case	File/SCF name	sim_1	sim_2	m_sim	PGQT CL	IQR	Human CL
1	$F113/Q1^*$	0.96	0.88	0.92	2	0.16	2
2	$F113/Q2^*$	0.99	0.83	0.91	2	0.15	2
3	$F113/Q3^*$	0.23	0.03	0.13	1	0.42	1
4	$F113/Q4^*$	0.97	0.93	0.95	2	0.09	2
5	F75/Q1	0.43	0.45	0.44	1	0.36	1
6	F75/Q2	0.99	0.99	0.99	2	0.10	2
7	F75/Q3	0.26	0.29	0.28	1	0.34	1
8	F75/Q4	0.94	0.93	0.94	2	0.14	2
9	F75/Q5	0.63	0.65	0.64	1	0.29	1
10	F75/Q6	0.66	0.67	0.67	1	0.26	1
11	F75/Q7	0.99	0.99	0.99	2	0.17	2
12	F82/Q1	0.99	0.97	0.98	2	0.20	2
13	F82/Q2	0.98	0.98	0.98	2	0.23	2
14	F82/Q3	0.33	0.33	0.33	1	0.30	1
15	F82/Q4	0.47	0.35	0.41	1	0.35	1
16	F82/Q5	0.40	0.29	0.35	1	0.33	1
17	F82/Q6	0.57	0.48	0.53	1	0.34	1
18	F82/Q7	0.39	0.33	0.36	1	0.32	1
19	F82/Q8	0.33	0.37	0.35	1	0.37	1
20	F82/Q9	0.97	0.94	0.96	2	0.23	2
21	F82/Q10	0.97	0.94	0.96	2	0.24	2
22	F86/Q1	0.99	0.97	0.98	2	0.18	2
23	F86/Q2	0.72	0.70	0.71	1	0.20	1
24	F86/Q3	0.97	0.96	0.97	2	0.19	2
25	F86/Q4	0.06	0.07	0.07	1	0.33	1
26	F86/Q5	0.98	0.98	0.98	2	0.17	2
27	F86/Q6	0.23	0.22	0.23	1	0.30	1
28	F86/Q7	0.44	0.47	0.46	1	0.37	1
29	F86/Q8	0.98	0.97	0.98	2	0.17	2
30	F86/Q9	0.05	0.06	0.06	1	0.33	1
31	F86/Q10	0.13	0.12	0.13	1	0.33	1
32	F86/Q11	0.62	0.62	0.62	1	0.35	1
33	F86/Q12	0.99	0.99	0.99	2	0.16	2
34	F86/Q13	0.94	0.92	0.93	2	0.16	2

* The F113 CL2 source-code fragments are found in more than two files. Only the similarity values between F77 and F113 are shown in the table.

and the contribution level values assigned by the human graders (*human CL*).

- H_1: There is a correlation between the PGQT similarity values (i.e., sim_1, sim_2, m_sim) and contribution level values assigned by the human graders (*human CL*).

Hypothesis 2:

- H_0: There is no correlation between the *human CL* and the *PGQT CL* variables.
- H_1: There is a correlation between the *human CL* and the *PGQT CL* variables.

Hypothesis 3:

- H_0: There is no correlation between the *IQR* and *human CL* variables.
- H_1: The greater the *human CL* value the smaller the *IQR*. There is a correlation between these two variables. The relative spread of values (using the IQR) in each set of source-code fragments can indicate the contribution level of source-code fragments.

Table 10.6: Spearman's rho correlations for all source-code fragments

Spearman's rho Correlations					
	sim_1	sim_2	m_sim	PGQT CL	IQR
Human CL	0.87^{**}	0.87^{**}	0.87^{**}	1.00^{**}	-0.85^{**}
Sig. (2-tailed)	0.00	0.00	0.00	0.00	0.00
N	34	34	34	34	34
**. Correlation is significant at the 0.01 level (2-tailed).					

Regarding *hypothesis 1*, Table 10.6 shows that average correlations between PGQT variables (sim_1, sim_2, m_{sim}) and *human CL* are strong and highly significant $r = 0.87$,

$p < 0.01$. This suggests that PGQT performs well in identifying the contribution level of source-code fragments.

Regarding *hypothesis 2*, Table 10.6 shows an increase in correlations between the *human CL* and the similarity values provided by PGQT (i.e. sim_1, sim_2, m_{sim}) when the PGQT values are classified into categories (*PGQT CL*). The correlations were strong and significant between the *human CL* and the *PGQT CL* variables with $r = 1.00$, $p < 0.01$. Classifying the PGQT similarity values has improved correlations with human judgement, i.e., a 0.13 increase in correlations, increasing from $r = 0.87, p < 0.01$ to $r = 1.00, p < 0.01$.

Regarding *hypothesis 3*, Table 10.6 shows that the correlations for variables *IQR* and *human CL* are strong and highly significant $r = -0.85$, $p < 0.01$. These findings suggest that taking into consideration the distribution of the similarity values between the source-code fragment and all the files in the corpus can reveal important information about the source-code fragment in question with regards to its contribution towards evidence gathering.

Before accepting any of the hypotheses it is worth investigating the correlations between variables when source-code fragments are grouped by the files they originated from. Table 10.7 shows the correlations.

Table 10.7 shows that the correlations are equal for the *human CL* and PGQT values (i.e. sim_1, sim_2, m_{sim}) and for the *human CL and IQR* variables. Note that although the

Table 10.7: Spearman's rho correlations for each file computed separately.

Spearman's rho Correlations					
	sim_1	sim_2	m_sim	IQR	PGQT CL
F113 Human CL	0.77	0.77	0.77	-0.77	1.00
Sig. (2-tailed)	0.23	0.23	0.23	0.23	0.00
N	4	4	4	4	4
F75 Human CL	0.87^{*}	0.87^{*}	0.87^{*}	-0.87^{*}	1.00^{**}
Sig. (2-tailed)	0.01	0.01	0.01	0.01	0.00
N	7	7	7	7	7
F82 Human CL	0.86^{**}	0.86^{**}	0.86^{**}	-0.86^{**}	1.00^{**}
Sig. (2-tailed)	0.00	0.00	0.00	0.00	0.00
N	10	10	10	10	10
F86 Human CL	0.87^{**}	0.87^{**}	0.87^{**}	-0.87^{**}	1.00^{**}
Sig. (2-tailed)	0.00	0.00	0.00	0.00	0.00
N	13	13	13	13	13
**. Correlation is significant at the 0.01 level (2-tailed).					
*. Correlation is significant at the 0.05 level (2-tailed).					

significance of correlations for F113 is computed as statistically low (i.e. 0.23), for the purposes of this research this correlation is still considered important because the aim is to investigate whether PlaGate can identify the contribution levels of any number of given source-code fragments when compared to human judgement. In conclusion, based on these findings H_1 of hypotheses 1, 2, and 3 can be accepted.

10.4 Conclusion

This Chapter presents an evaluation of the investigation component of PlaGate, i.e. PlaGate's Query Tool. The idea put into practice with the PlaGate's Query Tool (PGQT) is that plagiarised files can be identified by the *distinct* source-code fragments they contain, where distinct source-code fragments are those belonging to the contribution level 2 category, and

it is these source-code fragments that these can be used as indicators of plagiarism. PlaGate computes the relative importance of source-code fragments within all the files in a corpus and indicates the source-code fragments that are likely to have been plagiarised. A comparison of the results gathered from PGQT with human judgement, revealed high correlations between the two. PGQT has shown to estimate well the importance of source-code fragments with regards to characterising files.

On occasions, when given a source-code fragment, PGQT fails to detect its relative files which results in the source-code fragment being misclassified as low contribution when it is a high contribution source-code fragment. This was especially the case if the source-code fragment is mostly comprised of single word terms and symbols (e.g. +, :, >). This is because during pre-processing these are removed from the corpus. Future experiments are planned to evaluate the performance of PlaGate when keeping these terms and characters during pre-processing. This was also one of the reasons PGDT failed to detect some of the suspicious file pairs. This raises the questions of what is the impact of removing symbols on the effectiveness of source-code plagiarism with LSA?, and would replacing symbols by tokens (i.e such as those used by string matching algorithms) improve detection performance?.

Plagiarism is a sensitive issue and supporting evidence that consists of more than a similarity value can be helpful to academics with regards to gathering sound evidence of plagiarism. The experiments with similarity values revealed that in some cases, the single pair-wise similarity values between source-code fragments and files were not sufficient

indicators of the source-code fragments' contribution towards proving plagiarism. This is because relatively high similarity values were given to source-code fragments that were graded as low contribution by human graders. This issue was dealt with by classifying the similarity values provided by PGQT into previously identified contribution categories. Correlations have shown an improvement in results, i.e., higher agreement between classified similarity values and the values provided by human graders.

Chapter 11

Conclusions, Discussion and Future work

In this work we have proposed a definition of source-code plagiarism based on academic perspectives, and studied the application of LSA to detect similar source-code files and for the task of investigating the relative similarity between source-code files. Our proposed system is a model that allows the combination of LSA with external plagiarism detection tools to enhance the source-code plagiarism detection and investigation process.

11.1 Conclusions and Discussion

The research presented in this thesis proposes a novel and detailed definition of what constitutes source-code plagiarism from the academic perspective. We have conducted a survey

and based on academic responses we have created a definition of source-code plagiarism. To our knowledge, this is the first definition that attempts to define source-code plagiarism.

The research also proposes a novel way to approach plagiarism detection and investigation using the semantic information encapsulated by the information retrieval technique LSA and existing plagiarism detection algorithms. We are not aware of any other work that investigates the applicability of LSA as a technique for detecting similar source-code files with a view to detecting plagiarism. The thesis proposes a novel model for combining existing plagiarism detection tools with LSA for improving plagiarism detection. Through this application of LSA, this research expands on the possible applications of LSA and creates a bridge between information retrieval and source-code file similarity detection and investigation.

The performance of LSA is dependent on parameters driving its effectiveness. These parameters are task dependent, corpus dependent and inter-dependent. We conducted an analysis on the impact of parameters driving the effectiveness of source-code similarity detection with LSA, and we are not aware of other such research. Conducting investigation into parameters revealed the parameter combinations that appear optimal for our datasets, and also revealed parameter combinations that look promising for source-code corpora. Using those parameters, we developed an LSA based tool, PlaGate, that can function alone or be integrated with existing plagiarism detection tools for improving source-code similarity detection and investigation. The experimental evaluations show that LSA alone is well suited for the task of source-code similarity detection, and when combined with external

plagiarism detection tools further improves detection performance. In addition, we applied PlaGate to the task of relative similarity detection and visualisation of source-code artifacts. The PlaGate tool, has been developed that can function alone or be integrated with current plagiarism detection tools. The main aims of PlaGate are:

- to detect source-code files missed by current plagiarism detection tools,
- to provide visualisation of the relative similarity between files, and
- to provide a facility for investigating similar source-code fragments and indicate the ones that could be used as strong evidence for proving plagiarism.

The experimental findings revealed that PlaGate can complement plagiarism detection tools by detecting similar source-code files missed by them. This integration resulted in improved recall at the cost of precision, i.e. more true positives but also more false positives in the list of detected files. In the context of source-code plagiarism detection, string-matching based detection systems have shown to detect fewer false positives than an LSA based system.

Choosing the optimal parameter settings for *noise reduction* can improve system performance. Choice of dimensions is the most important parameter influencing the performance of LSA. Furthermore, automatic dimensionality reduction is still a problem in information retrieval. Techniques have been proposed for automating this process for the task of automatic essay grading by Kakkonen *at al.* [66]. For now, PlaGate uses parameters that

were found as giving good detection results from conducting previous experiments with source-code datasets.

During experimental evaluations, we have considered the creation of artificial datasets in order to investigate whether LSA would detect particular attacks. However, from the perspective of whether a specific attack can be detected by LSA, the behaviour of LSA is unpredictable – whether a particular file pair classified under a specific attack (i.e. change of an *if* to a *case* statement) is detected does not depend on whether LSA can detect a particular change as done in string matching algorithms. It depends on the semantic analysis of words that make up each file and the mathematical analysis of the association between words.

In the context of source-code plagiarism detection the behavior of LSA is far less predictable than string-matching plagiarism detection tools. The behaviour of an LSA based system depends heavily on the corpus itself and on the choice of parameters which are not automatically adjustable. Furthermore as already mentioned, an LSA based system cannot be evaluated by means of whether it can detect specific plagiarism attacks due to its dependence on the corpus, and this makes it difficult to compare PlaGate with other plagiarism detection tools which have a more predictable behavior.

The advantages of PlaGate are that it is language independent, and therefore there is no need to develop any parsers or compilers in order for PlaGate to provide detection in different programming languages. Furthermore, files do not need to parse or compile to be included in the comparison process. In addition, most tools compute the similarity between two files, whereas PlaGate computes the relative similarity between files. These are two

very different approaches which give different detection results.

String-based matching tools are expected to provide fewer false positives (i.e. better precision) when compared to techniques based on information retrieval algorithms. However, the detection performance of string-matching tools tends to suffer from local confusion and JPlag also has the disadvantage of not being able to include files that do not parse in the comparison process. PlaGate and string matching tools are sensitive to different types of attacks and this thesis proposes an algorithm for combining the two approaches to improve detection of suspicious files.

One limitation to this study was concerned with the datasets used for conducting the experiments. The only source-code datasets that were available to us for conducting the experiments were those provided by academics in our department. It was also very time demanding to devise an exhaustive list of similar file pairs for each dataset. Furthermore, we had observed the behaviour of weighting schemes and dimensionality and found a pattern of similar behaviour for the particular datasets we experimented with. However, this raises the question,

- will particular pattern of behaviour change when using other source-code datasets with different characteristics (e.g. different number of files and dictionary size)?

To answer the above question, one would need to conduct experiments using more source-code corpora in order to investigate the behaviour of LSA's parameter settings.

It would also be interesting to investigate which parameters work best by analysing the

corpora characteristics. For example, which parameters drive the effectiveness of source-code similarity detection with LSA when using C++ corpora, or corpora written in other programming languages? More open questions are discussed in Section 11.2 that follows.

11.2 Open Questions and Future Work

This Section brings together unanswered questions that were raised in previous chapters and introduces new questions for future research.

Document pre-processing is a common procedure in information retrieval. The experiments discussed in this thesis focus mainly on the impact of those pre-processing parameters specific to Java source-code, i.e. source-code comments, Java reserved terms (i.e. keywords), and skeleton code. We have not performed stemming or stop-word removal on the comments within the source-code files. This raises a question:

- what is the impact of applying stemming and/or stop-word removal to source-code comments on the effectiveness of source-code similarity detection with LSA?

During pre-processing we have converted all uppercase letters to lower case, however, we are aware that in programming languages such as Java, terms such as *Book* and *book* may hold different semantic meanings, therefore semantics might be lost from converting uppercase letters to lower case letters.

This leaves an open question:

- what is the impact of changing identifiers such as converting all upper-case to lower-case letters on the retrieval performance of LSA?

Results from the small experiment described in Chapter 7 revealed that separating identifiers and keeping comments in the source-code increases noise in the data, and decreases LSA performance. Performance of LSA is corpus dependent and also depends on choice of parameters, and this raises two questions,

- what is the impact of separating identifiers comprising of multiple words, on the effectiveness of source-code similarity detection with LSA, using large datasets? and
- what is the impact on computational efficiency for conducting such a task?.

In order to answer the above questions, one needs to perform detailed experiments and statistical analysis investigating the performance of LSA when using different parameter combinations.

Furthermore, symbols in source-code do carry meaning (e.g. $y > 4$ and $y < 4$), and by removing those symbols during pre-processing, important meaning from files may also be removed. This raises the question of,

- how to treat symbols in programming languages prior to applying LSA?

Possible ways of doing this would be to add the symbols to the term dictionary used to create the term-by-file matrix. Another way of treating symbols would be to replace them

with words (e.g. replace symbol $-$ with the word *minus*), or even to categorise symbols and to replace each one with their category name (e.g. replace occurrences of the mathematical symbols $+, -$ with the word *arithmetic*).

Experiments with how to treat symbols, would be of greater importance when applying LSA to languages such as Perl, which are heavily based on symbols. In addition to the question of,

- how effective is LSA for detecting source-code plagiarism in languages other than Java?

The following questions remain open,

- how effective is LSA for detecting source-code plagiarism in languages heavily based on symbols (e.g. Perl)?
- what parameters drive the effectiveness of LSA for detecting source-code plagiarism in symbol based languages?
- how can parameters be automatically optimised based on the corpus?
- what is the impact of changing identifiers such as correcting spelling mistakes, stemming, etc. on the retrieval performance of LSA for source-code similarity detection?

The similarity measure applied to compute the distance between two vectors influences the retrieval performance of LSA [143]. In the experiments described in this thesis, we

have used the cosine similarity measure to compute the distance between two vectors. This leaves the open questions,

- how does the cosine measure compare to other distance measures for the task of similar source-code file detection with LSA? and
- what is the impact of applying pivoted cosine document length normalisation [128] on LSA effectiveness for similar source-code file detection?

With regards to dimensionality reduction in LSA, the number of dimensions impacts on computational effort, and our results show that sometimes increasing the number of dimensions can improve precision. This raises the question,

- how much precision should we compromise by adjusting dimensionality on improving computational effort?

To answer this question one would need to look at LSA's precision and the computational effort of LSA when using various dimensionality settings. Both computational speed and LSA performance (indicated by evaluation measures such as precision) are important, and in the context of applying LSA to the task of source-code similarity detection, care must be taken not to use too many dimensions if it means slightly improving performance.

PlaGate is in prototype mode and future work includes integrating the tool within Sherlock [62]. In addition, work is planned to investigate parameter settings for languages other than Java, and to adjust PlaGate to support languages other than Java. Further future work

also includes research into the automatic adjustment of pre-processing parameters in order to improve the detection performance of PlaGate.

Bibliography

[1] A. Ahtiainen, S. Surakka, and M. Rahikainen. Plaggie: Gnu-licensed source code plagiarism detection engine for Java exercises. In *Baltic Sea '06: Proceedings of the 6th Baltic Sea Conference on Computing Education Research*, pages 141–142, New York, NY, USA, 2006. ACM.

[2] A. Aiken. Moss: A system for detecting software plagiarism. Software: www.cs.berkeley.edu/ aiken/moss.html, accessed: July 2008.

[3] A. Amir, M. Farach, and S. Muthukrishnan. Alphabet dependence in parameterized matching. *Information Processing Letters*, 49(3):111–115, 1994.

[4] J. Aslam, E. Yilmaz, and V. Pavlu. A geometric interpretation of r-precision and its correlation with average precision. In *SIGIR '05: Proceedings of the 28th Annual International ACM SIGIR Conference on Research and Development in Information Retrieval*, pages 573–574, New York, NY, USA, 2005. ACM.

[5] R. Baeza-Yates and B. Ribeiro-Neto. *Modern Information Retrieval*. ACM Press / Addison-Wesley, 1999.

[6] B. Baker. On finding duplication and near-duplication in large software systems. In *WCRE '95: Proceedings of the Second Working Conference on Reverse Engineering*, pages 86–95, Washington, DC, USA, 1995. IEEE Computer Society.

[7] B. Baker. Parameterized pattern matching by Boyer-Moore type algorithms. In *SODA '95: Proceedings of the Sixth Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 541–550, Philadelphia, PA, USA, 1995. Society for Industrial and Applied Mathematics.

[8] M. Bartlett. Tests of significance in factor analysis. *British Journal of Psychology*, 3:77–85, 1950.

[9] J. Beasley. The impact of technology on plagiarism prevention and detection: Research process automation, a new approach for prevention. In *Plagiarism: Prevention, Practice and Policy Conference*, pages 23–29, St James Park, Newcastle upon Tyne, UK, 2004. Newcastle: Northumbria University Press.

[10] B. Belkhouche, A. Nix, and J. Hassell. Plagiarism detection in software designs. In *ACM-SE 42: Proceedings of the 42nd Annual Southeast Regional Conference*, pages 207–211, New York, NY, USA, 2004. ACM.

[11] R. Bennett. Factors associated with student plagiarism in a post-1992 University. *Journal of Assessment and Evaluation in Higher Education*, 30(2):137–162, 2005.

[12] H. Bergsten. *JavaServer Pages*. O'Reilly Media, 3rd edition edition, 2003.

[13] M. Berry. Large-scale sparse singular value computations. *The International Journal of Supercomputer Applications*, 6(1):13–49, Spring 1992.

[14] M. Berry and M. Browne. *Understanding Search Engines: Mathematical Modeling and Text Retrieval (Software, Environments, Tools), Second Edition*. Society for Industrial and Applied Mathematics, Philadelphia, PA, USA, 2005.

[15] M. Berry, Z. Drmac, and E. Jessup. Matrices, vector spaces, and information retrieval. *SIAM Review*, 41(2):335–362, 1999.

[16] M. Berry, S. Dumais, and G. O'Brien. Using linear algebra for intelligent information retrieval. Technical Report UT-CS-94-270, University of Tennessee Knoxville, TN, USA, 1994.

[17] M. Bjorklund and C. Wenestam. Academic cheating: frequency, methods, and causes. In *European Conference on Educational Research*, Lahti, Finland, 1999.

[18] S. Brin, J. Davis, and H. García Molina. Copy detection mechanisms for digital documents. In *SIGMOD '95: Proceedings of the 1995 ACM SIGMOD International conference on Management of Data*, pages 398–409, New York, NY, USA, 1995. ACM.

[19] A. Britt, P. Wiemer-Hastings, A. Larson, and C. Perfetti. Using intelligent feedback to improve sourcing and integration in students' essays. *International Journal of Artificial Intelligence in Education*, 14:359–374, 2004.

[20] A. Broder. On the resemblance and containment of documents. *Sequences*, 00:21, 1997.

[21] A. Broder. Identifying and filtering near-duplicate documents. In *CPM '00: Proceedings of the 11th Annual Symposium on Combinatorial Pattern Matching*, pages 1–10, London, UK, 2000. Springer-Verlag.

[22] J. Carroll and J. Appleton. Plagiarism: A good practice guide. World Wide Web electronic publication: www.jisc.ac.uk, accessed: July 2001.

[23] R. Cattell. The scree test for the number of factors. *Multivariate Behavioral Research*, 1:245–276, 1966.

[24] C. Chen, N. Stoffel, M. Post, C. Basu, D. Bassu, and C. Behrens. Telcordia LSI engine: Implementation and scalability issues. In *RIDE '01: Proceedings of the 11th International Workshop on research Issues in Data Engineering*, pages 51–58, Washington, DC, USA, 2001. IEEE Computer Society.

[25] P. Clough. Plagiarism in natural and programming languages: An overview of current tools and technologies. Technical Report CS-00-05, University of Sheffield, 2000.

[26] G. Cosma and M. Joy. Source-code plagiarism: A U.K academic perspective. Research Report No. 422, Department of Computer Science, University of Warwick, 2006.

[27] G. Cosma and M. Joy. Source-code plagiarism: A U.K academic perspective. In *Proceedings of the 7th Annual Conference of the HEA Network for Information and Computer Sciences, HEA Network for Information and Computer*, 2006.

[28] G. Cosma and M. Joy. Towards a definition of source-code plagiarism. *IEEE Transactions On Education*, 51:195–200, 2008.

[29] G. Cosma and M. Joy. PlaGate: An approach to automatically detecting and investigating source-code plagiarism. *IEEE Transactions On Computers*, May 2008. Under Review.

[30] F. Culwin, A. MacLeod, and T. Lancaster. Source code plagiarism in U.K H.E computing schools, issues, attitudes and tools. Technical Report SBU-CISM-01-02, South Bank University, London, September 2001.

[31] W. Decoo. *Crisis on campus: Confronting academic misconduct*. MIT Press Cambridge, England, 2002.

[32] S. Deerwester, S. Dumais, T. Landauer, G. Furnas, and R. Harshman. Indexing by latent semantic analysis. *Journal of the American Society of Information Science*, 41(6):391–407, 1990.

[33] G. Denhière and B. Lemaire. A computational model of children's semantic memory. In *Proceedings of the 26th Annual Meeting of the Cognitive Science Society (CogSci'2004)*, pages 297–302. Lawrence Erlbaum Associates, 2004.

[34] S. Dey and A. Sobhan. Impact of unethical practices of plagiarism on learning, teaching and research in higher education: Some combating strategies. In *7th International Conference on Information Technology Based Higher Education and Training ITHET '06*, pages 388–393, 2006.

[35] M. Dick, J. Sheard, C. Bareiss, J. Carter, T. Harding D. Joyce, and C. Laxer. Addressing student cheating: definitions and solution. *SIGCSE Bulletin*, 35(2):172–184, 2003.

[36] C. Ding. A similarity-based probability model for latent semantic indexing. In *Research and Development in Information Retrieval*, pages 58–65, 1999.

[37] J. Donaldson, A. Lancaster, and P. Sposato. A plagiarism detection system. *SIGCSE Bulletin*, 13(1):21–25, 1981.

[38] S. Dumais. Improving the retrieval of information from external sources. *Behavior Research Methods, Instruments and Computers*, 23(2):229–236, 1991.

[39] S. Dumais. LSI meets TREC: A status report. In *Text REtrieval Conference*, pages 137–152, 1992.

[40] S. Dumais, G. Furnas, T. Landauer, S. Deerwester, and R. Harshman. Using latent semantic analysis to improve access to textual information. In *CHI '88: Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, pages 281–285, New York, NY, USA, 1988. ACM.

[41] L. Ferrè. Selection of components in principal component analysis: A comparison of methods. *Computational Statistics and Data Analysis*, 19(6):669–682, 1995.

[42] A. Flint, S. Clegg, and R. Macdonald. Exploring staff perceptions of student plagiarism. *Journal of Further and Higher Education*, 30:145–156, 2006.

[43] P. Foltz. Using latent semantic indexing for information filtering. *SIGOIS Bulletin*, 11(2-3):40–47, 1990.

[44] K. Fredriksson and M. Mozgovoy. Efficient parameterized string matching. *Information Processing Letters*, 100(3):91–96, 2006.

[45] G. Furnas, T. Landauer, L. Gomez, and S. Dumais. Statistical semantics: analysis of the potential performance of keyword information systems. *Human Factors in Computer Systems*, pages 187–242, 1984.

[46] D. Gitchell and N. Tran. Sim: a utility for detecting similarity in computer programs. *SIGCSE Bulletin*, 31(1):266–270, 1999.

[47] R. Gravina, M. Yanagisawa, and K. Akahori. Development and evaluation of a visual assesment asistant using latent semantic analysis and cluster analysis. In *Proceedings of International Conference on Computers in Education*, pages 963–968, 2004.

[48] D. Grossman and O. Frieder. *Information Retrieval: Algorithms and Heuristics (The Information Retrieval Series)(2nd Edition)*. Springer, December 2004.

[49] M. Halstead. Natural laws controlling algorithm structure? *ACM SIGPLAN Notices*, 7(2):19–26, 1972.

[50] M. Halstead. *Elements of Software Science (Operating and programming systems series)*. Elsevier Science, New York, USA, 1977.

[51] S. Hannabuss. Contested texts: issues of plagiarism. *Library Management*, 22(6/7):311–318, 2001.

[52] D. Harman. An experimental study of factors important in document ranking. In *SIGIR '86: Proceedings of the 9th Annual International ACM SIGIR conference on Research and Development in Information Retrieval*, pages 186–193, New York, NY, USA, 1986. ACM.

[53] T. Hoad and J. Zobel. Methods for identifying versioned and plagiarized documents. *Journal of the American Society for Information Science and Technology*, 54(3):203–215, 2003.

[54] J. Horn. A rationale and test for the number of factors in factor analysis. *Psychometrika*, 30:179–185, 1965.

[55] R. Hubbard and S. Allen. An empirical comparison of alternative methods for principal component extraction. *Journal of Business Research*, 15(2):173–190, 1987.

[56] R. Idury and A. Schaffer. Multiple matching of parameterized patterns. In *CPM '94: Proceedings of the 5th Annual Symposium on Combinatorial Pattern Matching*, pages 226–239, London, UK, 1994. Springer-Verlag.

[57] T. Jenkins and S. Helmore. Coursework for cash: the threat from on-line plagiarism. In *Proceedings of the 7th Annual Conference of the Higher Education Academy Network for Information and Computer Sciences*, pages 121–126, Dublin, Ireland, 29-31 August 2006.

[58] E. Jessup and J. Martin. Taking a new look at the latent semantic analysis approach to information retrieval. *Computational information retrieval*, pages 121–144, 2001.

[59] E. Jones. Metrics based plagiarism monitoring. *Journal of Computing Sciences in Colleges*, 16(4):253–261, 2001.

[60] K. Jones. A statistical interpretation of term specificity and its application in retrieval. *Document Retrieval Systems*, pages 132–142, 1988.

[61] K. Joreskog. Some contributions to maximum likelihood in factor analysis. *Psychometrika*, 32(4):433–482, 1967.

[62] M. Joy and M. Luck. Plagiarism in programming assignments. *IEEE Transactions on Education*, 42(1):129–133, 1999.

[63] jspmaker. Jspmaker v1.0.1. Website: http://www.hkvstore.com/jspmaker/, accessed: July 2008.

[64] H. Kaiser. The application of electronic computers to factor analysis. *Educational and Psychological Measurement*, 20:141–151, 1960.

[65] T. Kakkonen, N. Myller, E. Sutinen, and J. Timonen. Automatic essay grading with probabilistic latent semantic analysis. In *Proceedings of the 2nd Workshop on Building Educational Applications Using Natural Language Processing at the 43rd Annual Meeting of the Association for Computational Linguistics*, pages 29–36, Ann Arbor, Michigan, USA, 2005.

[66] T. Kakkonen, N. Myller, E. Sutinen, and J. Timonen. Comparison of dimension reduction methods for automated essay grading. *Natural Language Engineering*, 1:1–16, 2005.

[67] T. Kakkonen and E. Sutinen. Automatic assessment of the content of essays based on course materials. In *Proceedings of the International Conference on Information Technology: Research and Education 2004 (ITRE 2004)*, pages 126–130, London, UK, 2004.

[68] T. Kakkonen, E. Sutinen, and J. Timonen. Noise reduction in LSA-based essay assessment. In *Proceedings of the 5th WSEAS International Conference on Simulation, Modeling and Optimization (SMO'05)*, pages 184–189, Corfu Island, Greece, 2005.

[69] J. Kasprzak and M. Nixon. Cheating in cyberspace: Maintaining quality in online education. *Association for the Advancement of Computing In Education*, 12(1):85–99, 2004.

[70] S. Kawaguchi, P. Garg, M. Matsushita, and K. Inoue. Automatic categorization algorithm for evolvable software archive. In *IWPSE '03: Proceedings of the 6th Interna-*

tional Workshop on Principles of Software Evolution, pages 195–200, Washington, DC, USA, 2003. IEEE Computer Society.

[71] S. Kawaguchi, P. Garg, M. Matsushita, and K. Inoue. Mudablue: An automatic categorization system for open source repositories. In *APSEC '04: Proceedings of the 11th Asia-Pacific Software Engineering Conference*, pages 184–193, Washington, DC, USA, 2004. IEEE Computer Society.

[72] S. Kawaguchi, P. Garg, M. Matsushita, and K. Inoue. MUDABlue: An automatic categorization system for open source repositories. In *Proceedings of the 11th Asia Pacific Software Engineering Conference (APSEC 2004)*, pages 184–193, November 2004.

[73] P. Keith-Spiegel, B. G. Tabachnick, B. E. Whitley, and J. Washburn. Why professors ignore cheating: Opinions of a national sample of psychology instructors. *Ethics and Behavior*, 8(3):215–227, 1998.

[74] E. Kintsch, D. Steinhart, G. Stahl, and the LSA Research Group. Developing summarization skills through the use of LSA-based feedback. *Interactive Learning Environments*, 8(2):87–109, 2000.

[75] A. Kontostathis. Essential dimensions of latent semantic indexing (lsi). In *HICSS '07: Proceedings of the 40th Annual Hawaii International Conference on System Sciences*, page 73, Washington, DC, USA, 2007. IEEE Computer Society.

[76] A. Kontostathis and W. Pottenger. A framework for understanding latent semantic indexing (LSI) performance. *Information Processing and Management*, 42(1):56–73, 2006.

[77] A. Kuhn, S. Ducasse, and T. Girba. Enriching reverse engineering with semantic clustering. In *WCRE '05: Proceedings of the 12th Working Conference on Reverse Engineering*, pages 133–142, Washington, DC, USA, 2005. IEEE Computer Society.

[78] A. Kuhn, S. Ducasse, and T. Gírba. Semantic clustering: Identifying topics in source code. *Information and Software Technology*, 49(3):230–243, 2007.

[79] T. Lancaster and F. Culwin. Classifications of plagiarism detection engines. *Italics*, 4(2), 2005.

[80] T. Landauer and S. Dumais. A solution to Plato's problem: The latent semantic analysis theory of the acquisition, induction, and representation of knowledge. *Psychological Review*, 104(2):211–240, 1997.

[81] T. Landauer, P. Foltz, and D. Laham. Introduction to latent semantic analysis. *Discourse Processes*, 25:259–284, 1998.

[82] T. Landauer, D. Laham, and M. Derr. From paragraph to graph: Latent semantic analysis for information visualization. In *Proceedings of the National Academy of Sciences of the United States of America*, volume 101, pages 5214–5219, April 2004.

[83] T. Landauer, D. Laham, and P. Foltz. Learning human-like knowledge by singular value decomposition: A progress report. In *Advances in Neural Information Processing Systems*, volume 10. The MIT Press, 1998.

[84] T. Landauer, D. Laham, B. Rehder, and M. Schreiner. How well can passage meaning be derived without using word order: A comparison of latent semantic analysis and humans. In *COGSCI-97*, pages 412–417, Stanford, CA, 1997. Lawrence Erlbaum.

[85] S. Lappin. An introduction to formal semantics. In M. Aronoff and J. Rees-Miller, editors, *The Handbook of Linguistics*, pages 369–393. Blackwell, Oxford, 2000.

[86] P. Larkham and S. Manns. Plagiarism and its treatment in higher education. *Journal of Further and Higher Education*, 26(4):339–349, 2002.

[87] K. Lochbaum and L. Streeter. Comparing and combining the effectiveness of latent semantic indexing and the ordinary vector space model for information retrieval. *Information Processing and Management: an International Journal*, 25(6):665–676, 1989.

[88] C. Lundquist, D. Grossman, and O. Frieder. Improving relevance feedback in the vector space model. In *CIKM '97: Proceedings of the Sixth International Conference on Information and Knowledge Management*, pages 16–23, New York, NY, USA, 1997. ACM.

[89] M. Lungu, A. Kuhn, T. Gîrba, and M. Lanza. Interactive exploration of semantic clusters. In *3rd International Workshop on Visualizing Software for Understanding and Analysis (VISSOFT 2005)*, pages 95–100, 2005.

[90] E. Tempero M. Lin, R. Amor. A Java reuse repository for eclipse using LSI. In *Proceedings of the 2006 Australian Software Engineering Conference (ASWEC06)*. IEEE, 2006.

[91] J. Maletic and A. Marcus. Supporting program comprehension using semantic and structural information. In *International Conference on Software Engineering*, pages 103–112, 2001.

[92] J. Maletic and N. Valluri. Automatic software clustering via latent semantic analysis. In *ASE '99: Proceedings of the 14th IEEE International Conference on Automated Software Engineering*, page 251, Washington, DC, USA, 1999. IEEE Computer Society.

[93] U. Manber. Finding similar files in a large file system. In *Proceedings of the USENIX Winter 1994 Technical Conference*, pages 1–10, San Fransisco, CA, USA, 17–21 1994.

[94] A. Marcus. *Semantic-driven program analysis*. PhD thesis, Kent State University, Kent, OH, USA, 2003.

[95] A. Marcus, A. Sergeyev, V. Rajlich, and J. Maletic. An information retrieval approach to concept location in source code. In *Proceedings of the 11th IEEE Working*

Conference on Reverse Engineering (WCRE2004), Delft, The Netherlands, pages 214–223, November 9-12 2001.

[96] S. Marshall and M. Garry. How well do students really understand plagiarism? In *Balance, Fidelity, Mobility: Proceedings of the 2005 ASCILITE Conference*, Brisbane, Australia, 2005.

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

[98] D. McCabe, L. Trevino, and K. Butterfield. Cheating in academic institutions: A decade of research. *Ethics and Behavior*, 11(3):219–232, 2001.

[99] L. Moussiades and A. Vakali. PDetect: A clustering approach for detecting plagiarism in source code datasets. *The Computer Journal*, 48(6):651–661, 2005.

[100] M. Mozgovoy. Desktop tools for offline plagiarism detection in computer programs. *Informatics in Education*, 5(1):97–112, 2006.

[101] M. Mozgovoy. *Enhancing Computer-Aided Plagiarism Detection*. Dissertation, Department of Computer Science, University of Joensuu, Department of Computer Science, University of Joensuu, P.O.Box 111, FIN-80101 Joensuu, Finland, November 2007.

[102] M. Mozgovoy, K. Fredriksson, D. White, M. Joy, and E. Sutinen. Fast plagiarism detection system. *Lecture Notes in Computer Science*, 3772/2005:267–270, 2005.

[103] M. Mozgovoy, S. Karakovskiy, and V. Klyuev. Fast and reliable plagiarism detection system. In *Frontiers in Education Conference - Global eEngineering: Knowledge without Borders*, pages S4H–11–S4H–14, 2007.

[104] S. Myers. Questioning author(ity): ESL/EFL, science, and teaching about plagiarism. *Teaching English as a Second or Foreign Language (TESL-EJ)*, 3(2):11–20, 1998.

[105] S. Nadelson. Academic misconduct by university students: Faculty perceptions and responses. *Plagiary*, 2(2):1–10, 2007.

[106] P. Nakov. Latent semantic analysis of textual data. In *CompSysTech '00: Proceedings of the Conference on Computer systems and Technologies*, pages 5031–5035, New York, NY, USA, 2000. ACM.

[107] P. Nakov, A. Popova, and P. Mateev. Weight functions impact on LSA performance. In *Proceedings of the EuroConference Recent Advances in Natural Language Processing (RANLP'01)*, pages 187–193, 2001.

[108] K. Ottenstein. An algorithmic approach to the detection and prevention of plagiarism. *ACM SIGCSE Bulletin*, 8(4):30–41, 1976.

[109] K. Ottenstein. A program to count operators and operands for ansi fortran modules. Computer Sciences Report TR 196, Purdue University, 1976.

[110] D. Perez, A. Gliozzo, C. Strapparava, E. Alfonseca, P. Rodrguez, and B. Magnini. Automatic assessment of students' free-text answers underpinned by the combina-

tion of a bleu-inspired algorithm and latent semantic analysis. In *18th International Conference of the Florida Artificial Intelligence Research Society (FAIRS)*, pages 358–363, Florida, U.S.A., 2005. American Association for Artificial Intelligence (AAAI).

[111] C. Perfetti. The limits of co-occurrence: tools and theories in language research. *Discourse Processes*, 25:363–377, 1998.

[112] B. Pincombe. Comparison of human and LSA judgements of pairwise document similarities for a news corpus. Research Report No. AR-013-177, Defence Science and Technology Organisation - Australia, 2004.

[113] plagiarism. (n.d.). Dictionary.com unabridged v 1.1. Webpage: http://dictionary.reference.com/browse/plagiarism, accessed: April 2008.

[114] G. Plotkin. A structural approach to operational semantics. *Journal of Logic and Algebraic Programming (JLAP)*, 60–61:17–139, 2004.

[115] L. Prechelt, G. Malpohl, and M. Philippsen. Finding plagiarisms among a set of programs with JPlag. *Journal of Universal Computer Science*, 8(11):1016–1038, 2002.

[116] G. Rambally and M. Sage. An inductive inference approach to plagiarism detection in computer program. In *In Proceedings of the National Educational Computing Conference*, pages 22–29, Nashville, Tennesse, 1990.

[117] B. Rehder, M. Schreiner, M. Wolfe, D. Lahaml, W. Kintsch, and T. Landauer. Using latent semantic analysis to assess knowledge: Some technical considerations. *Discourse Processes*, 25:337–354, 1998.

[118] S. Robinson and M. Soffa. An instructional aid for student programs. In *SIGCSE '80: Proceedings of the Eleventh SIGCSE Technical Symposium on Computer Science Education*, pages 118–129, New York, NY, USA, 1980. ACM.

[119] L. Salmela and J. Tarhio. Sublinear algorithms for parameterized matching. *Lecture Notes in Computer Science*, 4009/2006:354–364, 2006.

[120] G. Salton, A. Wong, and C. Yang. A vector space model for automatic indexing. *Communications of the ACM*, 18(11):613–620, 1975.

[121] P. Scanlon and D. Neumann. Internet plagiarism among college students. *Journal of College Student Development*, 43(3):374–85, 2002.

[122] S. Schleimer, D. Wilkerson, and A. Aiken. Winnowing: local algorithms for document fingerprinting. In *SIGMOD '03: Proceedings of the 2003 ACM SIGMOD International Conference on Management of Data*, pages 76–85, New York, NY, USA, 2003. ACM.

[123] D. Schmidt. *Denotational semantics: a methodology for language development*. William C. Brown Publishers, Dubuque, IA, USA, 1986.

[124] H. Schütze. Dimensions of meaning. In *Supercomputing '92: Proceedings of the 1992 ACM/IEEE conference on Supercomputing*, pages 787–796, Los Alamitos, CA, USA, 1992. IEEE Computer Society Press.

[125] J. Sheard, A. Carbone, and M. Dick. Determination of factors which impact on it students' propensity to cheat. In *Proceedings of the Fifth Australasian Computing Education Conference*, pages 119–126, Adelaide, Australia, 2003.

[126] N. Shivakumar and H. García Molina. Scam: A copy detection mechanism for digital documents. In *Proceedings of the Second Annual Conference on the Theory and Practice of Digital Libraries*, June 1995.

[127] A. Singhal, C. Buckley, and M. Mitra. Pivoted document length normalization. In *Research and Development in Information Retrieval*, pages 21–29, 1996.

[128] A. Singhal, G. Salton, M. Mitra, and C. Buckley. Document length normalization. Technical report, Cornell University, Ithaca, NY, USA, 1995.

[129] K. Slonneger and B. Kurtz. *Formal Syntax and Semantics of Programming Languages: A Laboratory Based Approach*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1995.

[130] D. Steinhart. *Summary street: An intelligent tutoring system for improving student writing through the use of latent semantic analysis*. Phd dissertation, Department of Psychology, University of Colorado, Boulder, 2001.

[131] R. Story. An explanation of the effectiveness of latent semantic indexing by means of a bayesian regression model. *Information Processing and Management*, 32(3):329–344, 1996.

[132] S. Sudarsun, P. Venkatesh, and K. Sathish. Role of weighting on TDM in improvising performance of LSA on text data. In *Proceedings of the IEEE INDICON*, 2006.

[133] W. Sutherland-Smith. Pandora's box: academic perceptions of student plagiarism in writing. *Journal of English for Academic Purposes*, 4(1):83–95, 2005.

[134] P. Thomas, D. Haley, A. De Roeck, and M. Petre. E-assessment using latent semantic analysis in the computer science domain: A pilot study. In Erhard Hinrichs Lothar Lemnitzer, Detmar Meurers, editor, *COLING 2004 eLearning for Computational Linguistics and Computational Linguistics for eLearning*, pages 38–44, Geneva, Switzerland, August 28 2004. COLING.

[135] J. Tukey. *Exploratory Data Analysis (Addison-Wesley Series in Behavioral Science)*. Addison Wesley, London, 1977.

[136] W. Velicer. Determining the number of components from the matrix of partial correlations. *Psychometrika*, 41:321–327, 1976.

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

[138] K. Verco and M. Wise. Software for detecting suspected plagiarism: Comparing structure and attribute-counting systems. In John Rosenberg, editor, *Proceedings*

of the First Australian Conference on Computer Science Education, pages 81–88, Sydney, Australia, July 3–5 1996. SIGCSE, ACM.

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

[140] D. Whittington and H. Hunt. Approaches to the computerized assessment of free text responses. In M. Danson and R. Sherrat, editors, *Proceedings of the 3rd Annual CAA Conference*, pages 207–219, Loughborough, UK, 1999. Loughborough University.

[141] P. Wiemer-Hastings. How latent is latent semantic analysis? In *Proceedings of the Sixteenth International Joint Conference on Artificial Intelligence, IJCAI 99*, pages 932–941. Morgan Kaufmann, July 31–August 6 1999.

[142] P. Wiemer-Hastings, K. Wiemer-Hastings, and A. Graesser. Improving an intelligent tutor's comprehension of students with Latent Semantic Analysis. In S. Lajoie and M. Vivet, editors, *Artificial Intelligence in Education*, pages 535–542, Amsterdam, 1999. IOS Press.

[143] F. Wild, C. Stahl, G. Stermsek, and G. Neumann. Parameters driving effectiveness of automated essay scoring with LSA. In Myles Danson, editor, *Proceedings of the 9th International Computer Assisted Assessment Conference (CAA)*, pages 485–494, Loughborough, UK, July 2005. Professional Development.

[144] M. Wise. Yap3: improved detection of similarities in computer program and other texts. *SIGCSE Bulletin*, 28(1):130–134, 1996.

[145] M. Wolfe, M. Schreiner, R. Rehder, D. Laham, P. Foltz, T. Landauer, and W. Kintsch. Learning from text: Matching reader and text by latent semantic analysis. *Discourse Processes*, 25:309–336, 1998.

[146] L. Yi, L. Haiming, L. Zengxiang, and W. Pu. A simplified latent semantic indexing approach for multi-linguistic information retrieval. In *In Proceedings of the 17th Pacific Asia Conference on Language, Information and Computation (PACLIC17)*, pages 69–79, Sentosa, Singapore, 2003. COLIPS Publications.

[147] D. Zeimpekis and E. Gallopoulos. Design of a MATLAB toolbox for term-document matrix generation. Technical Report HPCLAB-SCG, Computer Engineering and Informatics Department, University of Patras, Greece, February 2005.

[148] H. Zha and H. Simon. A subspace-based model for latent semantic indexing in information retrieval. In *In Proceedings of the Thirteenth Symposium on the Interface*, pages 315–320, 1998.

[149] W. Zwick and W. Velicer. Factors influencing four rules for determining the number of components to retain. *Multivariate Behavioral Research*, 17(2):253–269, 1982.

[150] W. Zwick and W. Velicer. Factors influencing five rules for determining the number of components to retain. *Psychologica Bulleti*, 99:432–442, 1986.