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

Georgina Cosma and Mike Joy

Abstract—Plagiarism is a growing problem in academia. Academics often use plagiarism detection tools to detect similar source-code files. Once similar files are detected, the academic proceeds with the investigation process which involves identifying the similar source-code fragments within them that could be used as evidence for proving plagiarism. This paper describes PlaGate, a novel tool that can be integrated with existing plagiarism detection tools to improve plagiarism detection performance. The tool also implements a new approach for investigating the similarity between source-code files with a view to gathering evidence for proving plagiarism. Graphical evidence is presented that allows for the investigation of source-code fragments with regards to their contribution toward evidence for proving plagiarism. The graphical evidence indicates the relative importance of the given source-code fragments across files in a corpus. This is done by using the Latent Semantic Analysis information retrieval technique to detect how important they are within the specific files under investigation in relation to other files in the corpus.

Index Terms—Source-code similarity detection, similarity investigation tool, latent semantic analysis.

1 Introduction

Source-code plagiarism detection in programming assignments is a task many higher education academics carry out. Source-code plagiarism 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 [1]. Cosma and Joy created a detailed definition as to what constitutes source-code plagiarism from the perspective of academics who teach programming on computing courses [1].

Once similar file pairs are detected, the academic proceeds with the task of investigating the similar source-code fragments within the detected files. In source-code files, similarity may be suspicious or innocent. For example, similarity between source-code files may exist innocently due to source-code examples and solutions the students were given during classes [2]. Suspiciously similar source-code files are those that share source-code fragments that are distinct in program logic, approach, and functionality from source-code fragments found in other files in the corpus. This is the kind of similarity that could be used as strong evidence for proving plagiarism [2].

It is important that instances of plagiarism are detected and gathering sound evidence to confidently proceed with plagiarism is a vital procedure [1]. Joy and Luck [3] also 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."

Many tools have been developed for detecting similarities in files [4]. These tools automate the detection process and allow the academic to carry out the investigation process manually. One feature that does not exist in these tools is one that aids the academic during the process of investigating the detected similarity between files for plagiarism.

This paper describes a novel tool, PlaGate, for detecting similar source-code files, and investigating similar source-code fragments with a view to gathering evidence for proving plagiarism.

The main contributions of this paper are summarized as follows:

- an approach for enhancing the plagiarism detection performance of existing algorithms for detecting similarity in source-code plagiarism;
- a technique for semiautomatically investigating source-code fragments and indicating their contribution level (CL) toward evidence gathering for proving source-code plagiarism.

2 Source-Code Plagiarism Detection Tools

Many different plagiarism detection tools exist and these can be categorized depending on their algorithms. This section describes source-code plagiarism detection tools using the three categories identified by Mozgovoy [4], which are the *Fingerprint-based systems*, *String-matching systems*, and *Parameterized matching algorithms*.

Tools based on the fingerprint approach work by creating "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 [4].

- *G. Cosma is with the P.A. College, PO Box 40763, Larnaca 6307, Cyprus.*
- *M. Joy is with the University of Warwick, Coventry CV4 7AL, United Kingdom.*

Manuscript received 24 July 2008; revised 12 Aug. 2009; accepted 7 Nov. 2009; published online 21 Nov. 2011.
Recommended for acceptance by I. Markov.
For information on obtaining reprints of this article, please send e-mail to: tc@computer.org, and reference IEEECS Log Number TC-2008-07-0366.
Digital Object Identifier no. 10.1109/TC.2011.223.

An early plagiarism detection system was an attribute counting program developed by Ottenstein for detecting identical and nearly identical student work [5], [6]. The program used Halstead's software metrics to detect plagiarism by counting operators and operands for ANSI-FORTRAN modules [7], [8].

Robinson and Soffa developed a plagiarism detection program that combined new metrics with Halstead's metrics in order to improve plagiarism detection [9]. Their system ITPAD [9] breaks each program into blocks and builds graphs to represent the structure of each student's program. It then generates a list of attributes based on the lexical and structural analysis and compares pairs of programs by counting these characteristics.

Measure of Software Similarity (MOSS) [10] is based on a string-matching algorithm which divides 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—the more fingerprints they share, the more similar they are [11].

Some of the most well known and recent string-matching-based systems include Yet Another Plague (YAP3) [12], JPlag [13], and Sherlock [3]. In most string-matching-based systems, including the ones mentioned above, the first stage is called tokenization. At this 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 source-code document is then represented as a series of token strings. The tokens for each document are compared to determine similar source-code segments.

YAP3 converts programs into strings 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 [12]. This algorithm 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) by 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 flagged. 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 maximize the number of tokens covered by tiles.

JPlag [13] uses the same comparison algorithm as YAP3, but with optimized runtime 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 can cause similar files that were not parsed 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 RKR-GST algorithm behind JPlag. JPlag is implemented as a web service and contains a simple but efficient user interface, which 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 [3] also implements a similar algorithm to YAP3. 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 customizable [3] to languages other than Java. Sherlock's user interface displays a list of similar file pairs and their degree of similarity, and a display for comparing the pairs of files by indicating their matching blocks of source-code fragments. In addition, Sherlock displays quick visualization 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. Sherlock is a stand-alone tool and not a web-based service like JPlag and MOSS. A stand-alone tool may be preferable to academics with a view to checking student files for plagiarism when taking into consideration confidentially issues.

The DUP tool [14] is based on a parameterized matching algorithm which detects identical and near-duplicate sections of source-code, by matching source-code sections whose identifiers have been substituted (renamed) systematically.

Information retrieval methods have been applied to source-code plagiarism detection by Moussiades and Vakali [15]. 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.

3 What Is Latent Semantic Analysis (LSA)?

Latent Semantic Analysis is an information retrieval technique comprising mathematical algorithms that are applied to text collections. Initially a text collection is preprocessed and represented as a term-by-file matrix containing terms and their frequency counts in files. Matrix transformations are applied such that the values of terms in files are adjusted depending on how frequently they appear within and across files in the collection.

A mathematical algorithm called Singular Value Decomposition (SVD), decomposes this term-by-file matrix into separate matrices that capture the similarity between terms and between files across various dimensions. The aim is to represent the original relationships between terms in a reduced dimensional space such that noise is removed from

the data and therefore uncovering the important relations between terms and documents [16]. LSA aims to find underlying (*latent*) relationships between different terms that have the same meaning but never occur in the same file. In the context of textual information retrieval, noise accounts for variability in term usage.

LSA derives the meaning of terms from approximating the structure of term usage among documents through SVD. This underlying relationship between terms is believed to be mainly due to transitive relationships between terms, that is, terms are similar if they cooccur with the same terms within files [17], [18]. Traditional text retrieval systems cannot detect this kind of transitive relationship between terms and a consequence of this is that relevant files may not be retrieved. LSA categorizes terms and files into a semantic structure depending on their semantic similarity, hence *latent semantic* in the title of the method.

LSA is typically used for indexing large text collections and retrieving files based on user queries. One of the important features of LSA is that it uncovers the important relations between terms by reducing the noise in the data [17], and so can detect similar terms even if they never appear in the same document. LSA thus overcomes the problems of synonymy (different terms can be used to describe the same concept) and polysemy (one term can have more than one meaning) experienced by traditional text retrieval systems [19].

The ability of LSA to identify similar documents that contain different terms describing the same concept seems to be important in the area of plagiarism detection. This is because common plagiarism techniques include renaming variables in the code, and LSA is likely to detect documents that contain such semantic changes. Furthermore, changes to the document structure do not affect the detection of similar documents because LSA treats each document as a *bag of words*. Thus, if two documents are very similar but contain lexical and structural changes as an attempt to hide plagiarism, they are likely to be detected by LSA. Another advantage of using LSA is that it is language independent and therefore it is not needed to develop any parsers for programming languages in order for LSA to provide detection of similar files.

The literature does not appear to describe 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. A thorough literature review by Mozgovoy [4] also identifies this absence in literature.

Nakov [20] describes an application of LSA to a corpus of source-code programs written in the C programming language by Computer Science students. The 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 noncopied programs. The author assumes 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 data sets 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 it is suspected that this was the main reason that the authors report very high similarity values to nonsimilar 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).

Much work has been done in the area of applying LSA to software components. Some of the tools developed include MUDABlue [21] for software categorization, Softwarenaut [22] for exploring parts of a software system using hierarchical clustering, and Hapax [23] 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 categorizing software repositories in order to promote software reuse [24], [25], [26].

4 Similarity in Source-Code Files

Similarity between two files is established by investigating the source-code fragments found within the files for common characteristics. Investigation involves scrutinizing similar source-code fragments and thereafter judging whether the similarity between them appears to be suspicious or innocent. If a significant amount of suspicious similarity is found then the files under investigation can be considered suspicious.

As part of a survey conducted by Cosma and Joy [2], 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. 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 be used as strong evidence for proving plagiarism [2].

The 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 strong evidence for proving plagiarism [2]. 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 4.1 describes the types of similarity found between similar files.

4.1 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 plagiarized. When investigating files for plagiarism some source-code fragments would be used as stronger evidence than others, and from this perspective source-code fragments have varying contributions to the evidence gathering process of proving plagiarism. This section presents a criterion for identifying the contribution levels of source-code fragments with a view to proving plagiarism. This criterion was developed after considering the findings from a survey discussed in [2] and Section 4. 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 [2], and for this reason the criterion developed consists of categories describing similarity in the form of levels rather than similarity values.

The *contribution levels* for categorizing source-code fragments by means of their contribution toward providing evidence with a view to proving 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 (or template) 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 have been modified in a similar manner in the files in which they occur. Examples include *modified* template or other source-code provided to students to structure their programs, source-code fragments presented in lectures or handouts, and source-code fragments which are coincidentally similar due to the nature of the programming language used and other factors discussed in Section 4. 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 levels of similarity (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*.
- **Similarity Level 1—Suspicious.** The files under investigation share similar source-code fragments which characterize 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 1—Low contribution* and *Contribution Level 0—No 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.

5 PlaGate System Overview

The PlaGate tool aims to enhance the process of plagiarism detection and investigation. PlaGate is integrated within external plagiarism detection tools to improve plagiarism detection 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 suspiciously similar files. This tool can be integrated with external plagiarism detection tools for improving their 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 visualization 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 characterized 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. The idea is that files that are suspicious will 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 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. 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 evidence for proving plagiarism.

Fig. 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.

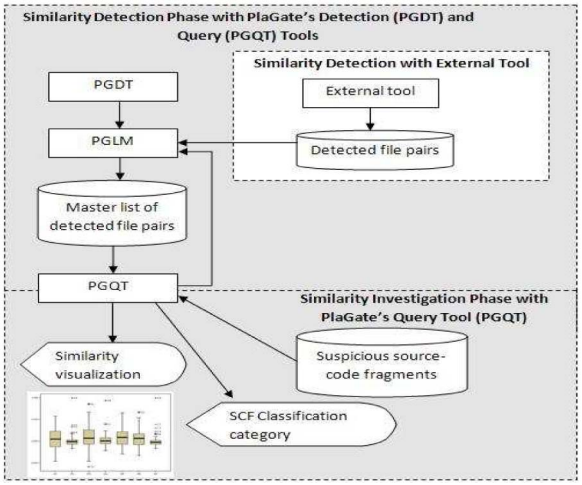

Fig. 1. PlaGate's detection and investigation functionality integrated with an existing plagiarism detection tool.

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.

PGQT can also be used for visualizing the relative similarity of source-code fragments and files with regards to identifying the contribution level of source-code fragments toward proving plagiarism. 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 visualizing the relative similarity between files. This procedure is described as follows:

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

1. PGQT accepts as input a corpus of source-code files, 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.

5.1 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.

6 The LSA Process in PlaGate

The LSA process in PlaGate is as follows:

1. Initially the files are preprocessed, by removing from the files terms that were solely composed of numeric characters, syntactical tokens (e.g., semicolons, colons), terms that occurred in less than two files (i.e., with global frequency less than two), and terms with length less than two. Prior experiments have shown that removing comments improves retrieval performance especially in files which contain suspicious source-code but whose comments are significantly different [27], [28].
2. LSA starts by transforming the preprocessed 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. The purpose of applying weighting algorithms is to increase or decrease the importance of terms using *local* and *global* weights in order to improve detection performance. With *document length normalization* the term values are adjusted depending on the length of each file in the corpus.

The value of a term in a file is $a_{ij} = l_{ij} \times g_i \times n_j$, where l_{ij} is the local weight for term i in file j, g_i is the global weight for term i, and n_j is the normalization factor. Various local and global weighting algorithms exist (see [29]). Exhaustive experiments were conducted with various source-code corpora and the results of those experiments revealed that the *term-frequency* local weight, *normal* global weight, and *cosine normalization* return good results [27].[2] Term frequency is the number of times term i occurs in document j and defined by f_{ij}. The *normal* global weighting is defined by

$$g_i = 1 \Big/ \sqrt{\sum_j f_{ij}^2},$$

and cosine normalization is defined by

$$n_j = \left(\sum_i (g_i l_{ij})^2 \right)^{-1/2}. \quad (1)$$

4. Singular Value Decomposition is then performed on the weighted matrix A. This involves decomposing matrix A into the product of three other matrices: a term by dimension matrix, U, a singular value matrix, Σ, and a file by dimension matrix, V, such that $A = U\Sigma V^T$.
5. Dimensionality Reduction is then performed in which the three matrices are reduced to k dimensions by selecting the first k columns from matrices U, Σ, and V, and the rest of the values are set to zero. The reduced matrices are denoted by U_k, Σ_k, and V_k, such that $A \approx A_k = U_k \Sigma_k V_k^T$. In the experiments described in this paper k was set to 30 dimensions based on results gathered from previous experiments [27], [28].

7 Similarity Detection and Source-Code Fragment Classification in PlaGate

After applying LSA to a source-code corpus, the PGQT or PGDT tools can be used for retrieving suspicious files. Using the PGQT tool 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 nonzero elements contain the weighted term frequency values of the terms, the query vector can be projected to the k-dimensional space by

$$Q = q^T \times U_k \times \Sigma_k^{-1}, \quad (2)$$

[16]. 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 a similarity measure. The cosine similarity measure is very popular and involves computing the dot product between two vectors (e.g., the query vector and file vector) and dividing it by the euclidean distance between the vectors. This ratio gives the cosine angle between the vectors, and the closer the angle is to $+1.0$ the higher the similarity between the query and file vector. Therefore, 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 Berry and Browne [29]

$$cos\Theta_j = \frac{a_j^T q}{\| a_j \|_2 \| q \|_2} = \frac{\sum_{i=1}^m a_{ij} q_i}{\sqrt{\sum_{i=1}^m a_{ij}^2} \sqrt{\sum_{i=1}^m q_i^2}}, \quad (3)$$

for $j = 1, \ldots, n$ where n is equal to the number of files in the data set (or the number of columns in the term-by-file matrix A). 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, $sv = sim(Q, F), \in [-1.0, +1.0]$.

Common terms can exist accidentally in source-code files and due to the factors discussed in Section 4, and this can cause relatively high-similarity values between files in an LSA-based system. For this reason a criterion has been devised for classifying the source-code fragments into the categories described in Section 4.1.

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 7. 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 PGDT detection tool uses a different algorithm for detecting similar file pairs in PlaGate. Using the PGQT tool and thus 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, function 4 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 Deerwester et al. [30]

$$(V_k \Sigma_k)(V_k \Sigma_k)^T. \quad (4)$$

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

The output from PlaGate is visualized using Tukey's boxplots (also called box-and-whisker plots) [31]. To construct a boxplot the elements of the SV vector are 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.

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

2. The experiments included all the data sets that were involved in the experiments described in this paper.

TABLE 1
The Data Sets

	A	B	C	D
Number of files	106	176	179	175
Number of terms	537	640	348	459
Number of suspicious file pairs	6	48	51	79
Number of files excluded from comparison by JPlag	13	7	8	7

8 Experiments

Four corpora comprising of 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 1 shows the characteristics of each data set.

In Table 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 preprocessing is performed as discussed in Section 6. *Number of suspicious file pairs* is the total number of file pairs that were detected in a corpus and judged as suspicious by academics (the term suspicious is defined in Section 4.1). JPlag reads and parses the files prior to comparing them. Files that cannot be read or do not parse successfully are excluded from the comparison process by JPlag. In Table 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.

In the next two sections that follow results gathered from two experiments are described. *Experiment 1* is concerned with evaluating the performance of PlaGate and two existing external tools, i.e., Sherlock and JPlag, and investigating whether integrating existing plagiarism detection tools with PlaGate enhances source-code similarity detection performance. *Experiment 2* is concerned with the evaluation of the PGQT tool for investigating source-code fragments.

8.1 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 $sim(F_a, F_b) \geq \phi$ will be detected. For the purposes of evaluation the following terms are defined:

- *Suspicious* file pairs: each suspicious file pair, sp, contains files that have been judged by human graders as suspicious (as defined in Section 4.1). A set of suspicious file pairs is denoted by $SP = \{sp_1, sp_2, \ldots, sp_x\}$, where the total number of known suspicious file pairs in a set (i.e., corpus) is $x = |SP|$.
- *Innocent* file pairs: are those pairs that do not share any suspicious similarity but have been detected as suspicious by the tools.
- *Detected* file pairs: are those pairs that have been retrieved with $sim(F_a, F_b) \geq \phi$. 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 $SD \subseteq S$. The total number of detected file pairs is $|DF|$. The total number of suspicious file pairs detected is denoted by $|SD|$, and the total number of innocent file pairs detected is denoted by $|NS|$.
- 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 data set, detection was performed using cutoff values $\phi_p = 0.70$ and $\phi_p = 0.80$. The cutoff value selected for each data set was the one that detected the most suspicious file pairs and fewer innocent.
- 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 suspicious or innocent. The cutoff value is set to the N position in the list of detected file pairs where the number of innocent file pairs detected begins to increase. In computing Sherlock's precision $N = |DF|$.

Recall, denoted by R, where $R \in [0, 1]$, is the proportion of suspicious file pairs that are detected based on the cutoff value, ϕ. Recall is 1.00 when all suspicious file pairs are detected

$$Recall = \frac{|SD|}{|SP|} = \frac{number\_of\_suspicious\_file\_pairs\_detected}{total\_number\_of\_suspicious\_file\_pairs}. \tag{5}$$

Precision, denoted by P, where $P \in [0, 1]$, is the proportion of suspicious file pairs that have been detected in the list of files pairs detected. Precision is 1.00 when every file pair detected is *suspicious*

$$Precision = \frac{|SD|}{|DF|} = \frac{number\_of\_suspicious\_file\_pairs\_detected}{total\_number\_of\_file\_pairs\_detected}. \tag{6}$$

The overall performance of each tool will be evaluated by combining the precision and recall evaluation measures. As a single measure for evaluating the performance of tools for plagiarism detection, the weighted sum of precision and recall will be computed by

$$F = \frac{Precision + 2 \times Recall}{3}, \tag{7}$$

where $F \in [0, 1]$. The closer the value of F is to 1.00 the better the detection performance of the tool.

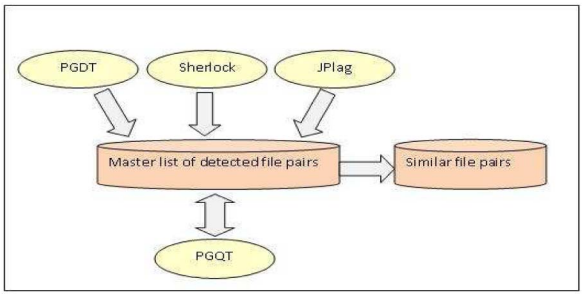

Fig. 2. Methodology for creating a list of suspicious file pairs.

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

8.2 Experiment 1

Experiment 1 is concerned with evaluating the performance of PlaGate against two external tools, JPlag and Sherlock. Detection performance when the tools function alone and when integrated with PlaGate is evaluated. Literature contains evaluations of plagiarism detection tools [13], [32], [4]. The performance of JPlag is considered to be close to that of MOSS [13], therefore JPlag was selected for comparison and integration with PlaGate as it was readily available to us. Furthermore, evaluations revealed that string-matching algorithms JPlag and Yap3 performed better than the parameterized matching algorithm DUP by means of retrieving a higher number of suspicious file pairs. In the evaluations discussed in this paper, Sherlock was also used because it was readily available and used by the Department of Computer Science at the University of Warwick, and it is believed that other institutions also use this tool.

In order to evaluate the detection performance of tools using the evaluation measures discussed in Section 8.1, the total number of suspicious file pairs in each corpus must be known. Fig. 2 illustrates the process of creating the list of suspicious 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. This produced three lists of detected file pairs (based on a given cutoff value as discussed in Section 8.1), where each list corresponds to a tool output. These three lists were merged into a master list. Other known suspicious file pairs identified by academics, but 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 scrutinized the detected file pairs contained in the master list and identified the suspicious file pairs. The final outcome consisted of four lists, i.e., one for each corpus, containing the paired ID numbers of suspicious files.

After constructing the list of suspicious file pairs, the detection performance of each tool is evaluated using the evaluation measures discussed in Section 8.1, and the detection results returned by each tool are based on a given cutoff value. In addition, detection performance is evaluated when PGQT is applied to 1) all detected file pairs in the list, and 2) only to those file pairs judged as suspicious by the academics. Results from the second case will be marked with *PGQT v2* in the description of results.

TABLE 2
Performance of Tools on Data Set A

	Recall	Precision	F
$PGDT\ \phi_p = 0.70$	0.67	0.80	0.71
$Sherlock\ \phi_s = 20$	0.83	0.25	0.64
$Sherlock@20 \cup PGDT$	1.00	0.27	0.76
$Sherlock@20 \cup PGDT \cup PGQT$	1.00	0.23	0.74
$JPlag\ \phi_j = 54.8$	0.50	0.75	0.58
$JPlag \cup PGDT$	0.67	0.67	0.67
$JPlag \cup PGDT \cup PGQT$	0.83	0.71	0.79
$JPlag \cup Sherlock@20$	1.00	0.27	0.76

TABLE 3
Performance of Tools on Data Set B

	Recall	Precision	F
$PGDT\ \phi_p = 0.80$	0.38	0.75	0.50
$Sherlock\ \phi_s = 40$	0.75	0.90	0.80
$Sherlock@40 \cup PGDT$	0.83	0.83	0.83
$Sherlock@40 \cup PGDT \cup PGQT$	0.96	0.78	0.90
$JPlag\ \phi_j = 99.2$	0.42	1.00	0.61
$JPlag \cup PGDT$	0.46	0.79	0.57
$JPlag \cup PGDT \cup PGQT$	0.54	0.79	0.62
$JPlag \cup Sherlock@40$	1.00	0.92	0.97

Tools are expected to detect similar file pairs and it would not be reasonable to penalize a system for detecting suspicious files that after undergoing human judgment are found not to contain suspicious similarity that can be regarded as plagiarism, as their similarity may be due to factors such as those discussed in Section 4.

Evaluation measures are applied to evaluate the detection performance of each of the tools PGDT, JPlag, and Sherlock. Evaluations from combining PGDT and PGQT with Sherlock, and JPlag are also performed.

Following is a summary of the results based on Tables 2, 3, 4, and 5.

- Recall increases when each of the external tools, JPlag and Sherlock, are integrated with PGDT, than when they are used alone. The results show that when combining PGDT with external tools, precision may decrease although this may not always be the case. The constant increase in recall suggests that PGDT and the external tools complement each other by detecting different kinds of plagiarism attacks, and thus suspicious files missed by one tool are likely to be detected by the other tool.

TABLE 4
Performance of Tools on Data Set 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
$Sherlock@30 \cup PGDT$	0.65	0.89	0.73
$Sherlock@30 \cup PGDT \cup PGQT$	0.65	0.89	0.73
$JPlag\ \phi_j = 91.6$	0.37	1.00	0.58
$JPlag \cup PGDT$	0.57	1.00	0.71
$JPlag \cup PGDT \cup PGQT$	0.71	0.88	0.76
$JPlag \cup Sherlock@30$	0.67	0.89	0.74

TABLE 5
Performance of Tools on Data Set 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
$Sherlock@50 \cup PGDT$	0.70	0.82	0.74
$Sherlock@50 \cup PGDT \cup PGQT$	0.82	0.76	0.80
$JPlag\ \phi_j = 100.0$	0.25	1.00	0.50
$JPlag \cup PGDT$	0.28	0.52	0.36
$JPlag \cup PGDT \cup PGQT$	0.30	0.55	0.38
$JPlag \cup Sherlock@50$	0.57	0.90	0.68

- When external tools are integrated with both PGDT and PGQT there is a further increase in recall than when external tools are used alone or with PGDT. However, this integration is likely to cause a slight decrease in precision than when the external tools are integrated only with PGDT.
- Although JPlag returned high precision values across all data sets, it suffered from low recall. Integrating JPlag with PlaGate's PGDT and/or PGQT increased recall considerably but caused a negative impact on precision than when JPlag was used alone.
- Combining JPlag and Sherlock improved recall values for data sets A, B, and C, whereas recall remained the same for data set D. When JPlag was combined with Sherlock precision decreased. However, overall results suggest that by combining the two tools together more suspicious file pairs are detected, which is a desirable outcome for academics.
- JPlag and Sherlock are both based on a string-matching algorithm, but yet their detection performance differs significantly on data sets. Evidence from the experiments does not indicate which of the two tools, JPlag or Sherlock, is best for plagiarism detection, however, the results suggest that combining the techniques together or with PlaGate results in improved detection performance, i.e., more suspicious files are being detected when using a combination of tools than when using a single tool.

Fig. 3 illustrates the value of Precision at various level of Recall for data set D. The performance of the various tools can be quickly read from the chart—the order of the unions indicates the order of sets of documents retrieved by each tool, for example, the $JPlag \cup PGDT \cup PGQT$ curve shows

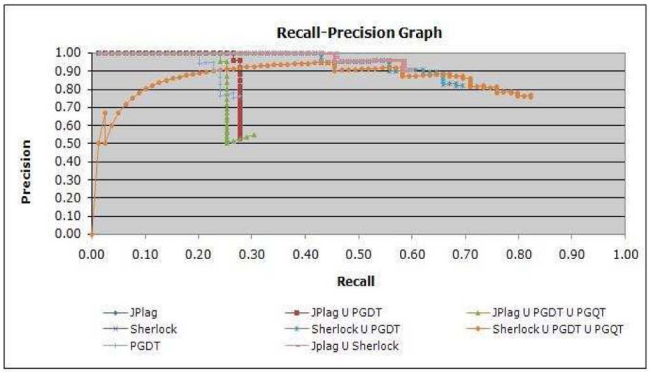

Fig. 3. Recall-precision graph for data set D.

File A	File B

1. public int backtrack Control(IRobot robot)
2. {
3. System.out.println("BACKTRACK");
4. int direction = IRobot.AHEAD;
5. int x = nonWallExits(robot);
6. int y = passageExits(robot);
7. if (x > 2)
8. {
9. if (y > 0)
10. {
11. direction = exploreControl(robot);
12. robotdata.popall();
13. }
14. else
15. {
16. direction = around(robot);
17. robotdata.popall();
18. }
19. }
20. if (x == 2){direction = corridor(robot);}
21. if (x == 1){direction = deadend(robot);}
 	
22. return direction;
23. }
	1. public int backtrackControl(IRobot robot)
	2. {
	3. System.out.println("Going backwards");
	4. if (nonwallexits(robot) > 2)
	5. {
	6. if (passageexits(robot) > 0){
	7. direction = explorerControl(robot);
	8. robotData.popall();}
	9. else
	10. { direction = oppositepassage(robot);
	11. robotData.popall();}
	12. }
	13. if (nonwallexits(robot) == 1){
	14. direction = deadend(robot);}
	15. else
	16. if (nonwallexits(robot) == 2){
	17. direction = corridor(robot);}
	18. return direction;
	19. }

Fig. 4. Source-code extract from a file pair that was detected by PlaGate and not by JPlag or Sherlock.

that the initial points of the particular line are the documents retrieved by JPlag, then PGDT, and the last ones are those retrieved by PGQT. The $JPlag \cup PGDT \cup PGQT$ curve clearly illustrates that integrating PGDT (the data points of the green vertical line) decreases precision. The green line illustrates an increase in recall and precision where the last five points on the line are concerned—which are PGQT's results.

Similarity often occurs in groups containing more than two files. Sherlock and JPlag often failed to detect some of the files from groups containing suspicious files. JPlag failed to parse some files that were suspiciously similar to other files in a corpus (and they were excluded from the comparison process), and also suffered from local confusion which also resulted in failing to detect suspicious file pairs [13]. Local confusion occurs when source-code segments shorter that 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 suspicious file pairs because they were detected as having lower similarity than nonsimilar files (i.e., retrieved further down the list and lost among many innocent) mainly due to local confusion.

Fig. 4 contains an extract from two files that were suspiciously similar. The structural changes, i.e., extra lines of code, caused string-matching tools JPlag and Sherlock to fail to detect the particular file pair. Each of the files contain approximately 385 lines of code, and thus only an extract is shown in this paper.

A careful comparison of the two source-code fragments shown in Fig. 4 reveals the following similarities: lines of code are similar but their location is different within the files; file A introduces extra variables x, and y; lexical differences (different identifier/variable/method names); and importantly the functionality of files A and B remains the same regardless of all the differences that can be considered as techniques for disguising similarity.

PlaGate does not suffer from local confusion because it does not rely 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.

TABLE 6
Data Set A File Pairs and Their Similarity Values

Files under investigation in Dataset A, n=106				
File Pairs	PGQT	H1 LS	H2 LS	PGQT LS
F82-F9	0.96	1	1	1
F75-F28	0.99	1	1	1
F86-F32	0.99	1	1	1
F45-F97	0.77	0	0	0
F73-F39	0.37	0	0	0

Overall results suggest that integrating PlaGate with external tools improves overall detection performance. Plagiarism detection using a combination of two tools is always better than using a single tool.

8.3 Experiment 2

This section is concerned with the evaluation of PGQT's output against human judgment with a view to investigating of source-code fragments for plagiarism. Experiment *2* follows on from experiment *1* discussed in Section 8.2. From the file pairs detected in data sets A and B from experiment *1*, 34 pairs of similar source-code fragments each consisting of one Java method are to be investigated. The file pairs containing the source-code fragments for investigation are: F113-F77, F75-F28, F82-F9, F86-F32. Source-code fragments were extracted from the first file from each file pair (i.e., F113, F75, F82, and F86) and 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 criterion discussed in Section 4.1. 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. Section 8.3.1 discusses the visualization output from investigating the file pairs in data set A. Suspicious and innocent file pairs have been selected as examples for investigating the visualization component of PlaGate's PGQT tool.

Table 6 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. Table 8 shows the source-code fragment ID's and their similarity values to files, and it is described in Section 8.3.3.

As discussed in Section 6, 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 4.1), 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 characterizing other files in the corpus, then these files will also be retrieved indicated by a relatively high similarity value.

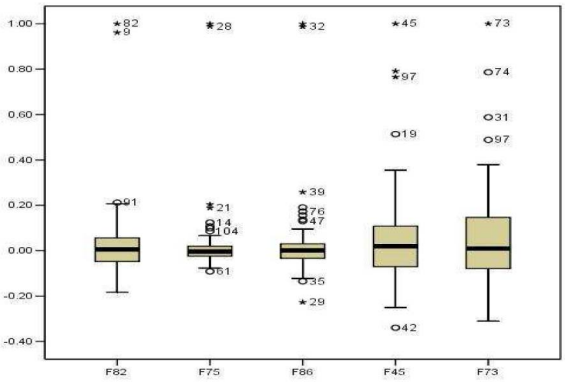

Fig. 5. Data set A: Similar files under investigation are F82-F9, F75-F28, F63-F32, F45-F97, F73-F39. Outliers $sim_i \geq 0.80$: F9-9,82; F75-75,28; F86-86,32.

With regards to source-code fragments belonging to contribution level 1, described in Section 4.1, 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.

8.3.1 Data Set A: Investigation of Source-Code Files Using PGQT

From data set A, files F82, F75, F86, F45, and F73 were used as queries in PGQT. Fig. 5 shows the output from PGQT, which shows the relative similarity between these files and the rest of the files in the corpus.

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 files were returned as extreme outliers with a large separation from the rest of the files in the corpus. Table 7 contains the top five highest outliers for each of the files of data set A. These are indicators that files F82, F75, and F86 are suspicious.

Files F45 and F73 were graded at similarity level 0 (innocent) by the human graders. The boxplots in Fig. 5 and Table 7 show that files F45 and F73 were returned in the top

TABLE 7
Data Set 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

TABLE 8
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.

ranked 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 wide 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.

8.3.2 Data Sets 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 data set A, and file F113 of data set 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 8.3. 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 Figs. 7, 8, 9, and 10, respectively. Table 8 shows the contribution levels assigned to the source-code fragments by the human graders.

File C	File D
Method clearBoard() is a CL1 fragment	Method clearBoard() is a CL1 fragment

1. public void clearBoard(){
2. for(int i =0;i <boardArray.length;i++){
3. for(int n =0;n <boardArray.length;n++)
4. boardArray[i][n] = null;}}
	1. public void clearBoard(){
	2. for(int y =0;y <tilearray.length;y++){
	3. for(int x =0;x <tilearray.length;x++)
	4. tilearray[y][x] = null;}}

Method addElements() is a CL2 fragment	Method addElements() is a CL2 fragment

1. public void addElements(String str){
2. if (noofElements == wordboardlist1.length){
3. String[] wordboardlist2 =
4. new String[wordboardlist1.length*2];
5. for (int i=0; i<wordboardlist1.length; i++){
6. wordboardlist2[i]=wordboardlist1[i];}
7. wordboardlist1=wordboardlist2;}
8. wordboardlist1[noofElements] = str;
9. noofElements++;}
10. public String elementAt(int i){
11. return wordboardlist1[i];}
12. public int size(){
13. return noofElements;}
	1. public void addElements(String str){
	2. if (Elements == wordboardlist1.length){
	3. String[] wordboardlist2 =
	4. new String[wordboardlist1.length*2];
	5. for (int i=0; i<wordboardlist1.length; i++){
	6. wordboardlist2[i]=wordboardlist1[i];}
	7. wordboardlist1=wordboardlist2;}
	8. wordboardlist1[Elements] = str;
	9. Elements++;}
	10. public String elementAt(int i){
	11. return wordboardlist1[i];}
	12. public int size(){
	13. return Elements;}

Method StringArray() is a CL2 fragment	StringArray() is a CL2 fragment

1. public String[] StringArray(){
2. String[] newArray = new String[noofElements];
3. for(int i =0;i <noofElements;i++){
4. newArray[i]=elementAt(i);}
5. return newArray;}
	1. public String[] StringArray(){
	2. String[] newarray = new String[Elements];
	3. for(int i =0;i <Elements;i++){
	4. newarray[i]=elementAt(i);}
	5. return newarray;}

Fig. 6. CL1 and CL2 Source-code extracts from a file pair that was detected by PlaGate and not by JPlag or Sherlock.

The fragments shown in Fig. 6, demonstrate the difference between source-code belonging to the CL1 and CL2 categories. The fragments were extracted from file pairs detected by PlaGate $(sim_i = 0.99)$ but failed to be detected by JPlag or Sherlock. The particular file pair could not be parsed by JPlag, and Sherlock failed to detect it since the strings are different, and because the fragments were found in different locations within each file. JPlag is also sensitive to this kind of attack (i.e., local confusion). LSA did detect them since it is not sensitive to systematic-renaming or code-shuffling. Extra comments have been added above each fragment containing its name and contribution level category.

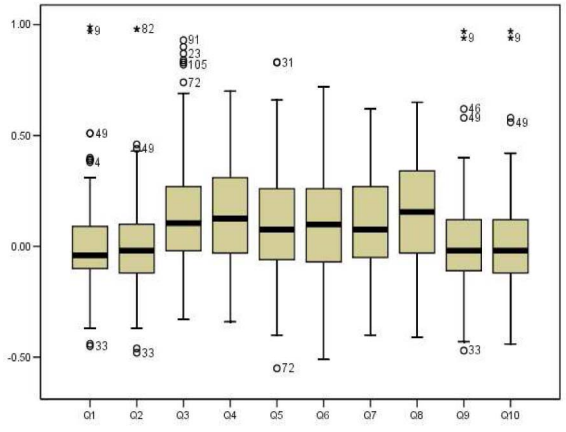

Fig. 7. F82 boxplot: Similar files under investigation are F82 and F9. Outliers $sim_i \geq 0.80$: Q1—82, 9; Q2—9, 82; Q3—91, 54, 23, 26, 104, 105, 94; Q5—22, 31; Q9—9, 82; Q10—9, 82.

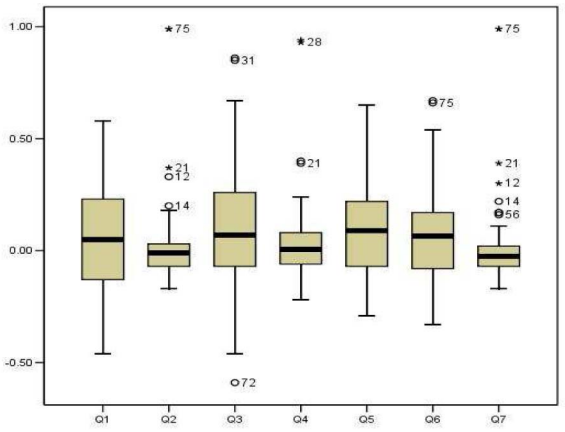

Fig. 8. F75 boxplot: Similar files under investigation are F75 and F28. Outliers $sim_i \geq 0.80$: Q2—75, 28; Q3—22, 31; Q4—75, 28; Q7—75, 28.

In the case of the source-code fragments extracted from file F82 (Fig. 7), 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 8). 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 data set. 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, and Q8 are 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 8, the observations from boxplots match the academic judgments.

Fig. 8 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 8 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 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 cannot be considered as particularly important fragment in files F75 and F28.

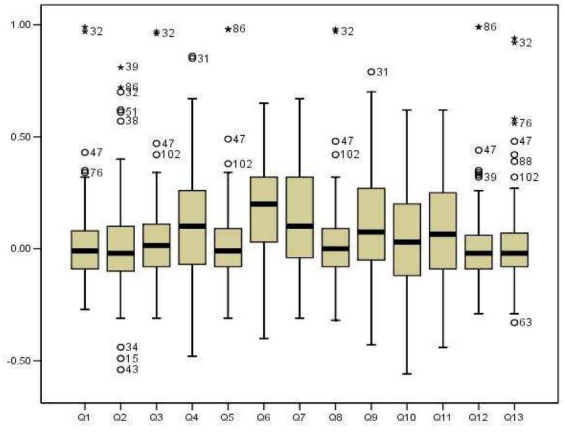

Fig. 9. F86 boxplot: Similar files under investigation are F86 and F32. Outliers $sim_i \geq 0.80$: Q1—86, 32; Q2—39; Q3—86, 32; Q4—86, 85; Q5—86, 32; Q8—86, 32; Q12—86, 32; Q13—86, 32.

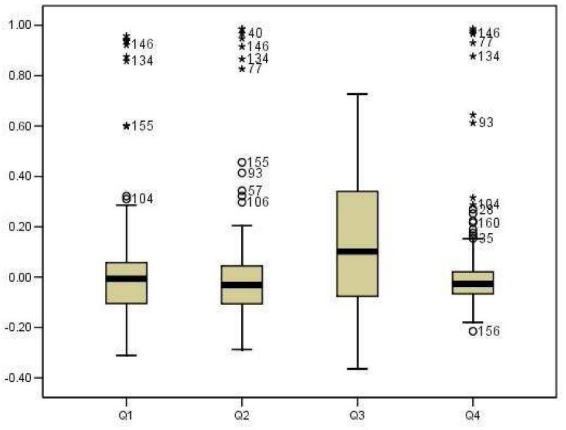

Fig. 10. F113 boxplot: Similar files under investigation are F113 and F77. Outliers with $sim_i \geq 0.80$: Q1—121, 149, 40, 113, 146, 134, 77; Q2—121, 149, 40, 113, 146, 134, 77; Q4—121, 149, 40, 113, 146, 134, 77. Q1, Q2, and Q4 are CL2 SCFs, Q3 is a CL1 SCF.

The boxplots corresponding to source-code fragments extracted from file F86 shown in Fig. 9 and from file F113 shown in Fig. 10 follow a similar pattern. In Fig. 10, 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. During the detection process JPlag only identified 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 scrutinizing these revealed that they all shared suspicious source-code fragments. The suspicious source-code fragments are Q1, Q2, and Q4 shown in Fig. 10.

In summary, the boxplots follow either Pattern 1 or Pattern 2. These are discussed as follows:

- 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 data sets are lower than the median values of the contribution level 1 data sets.
  - 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 toward 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.

8.3.3 Comparing PGQT's Results with Human Judgment

This section describes and explores hypotheses formed from analyzing the output of PGQT in Section 8.3.2. Table 8 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 4.1), column IQR shows the IQR values computed from the boxplot, and *Human CL* is the contribution level provided by the academics based on the criterion in Section 4.1.

Observing the similarity values provided by LSA and comparing those to the similarity levels provided by the human graders, revealed a pattern—grouping 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 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:

TABLE 9
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).					

Hypothesis 1.

- *H_0: There is no correlation between the PGQT similarity values (i.e., sim_1, sim_2, m_sim) 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.*

Regarding *hypothesis 1*, Table 9 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 9 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 judgment, 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 9 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 toward evidence gathering.

TABLE 10
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).					

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 shows the correlations.

Table 10 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 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 judgment. In conclusion, based on these findings H_1 of hypotheses 1, 2, and 3 can be accepted.

9 Conclusion

This paper proposes a novel approach based on the Latent Semantic Analysis information retrieval technique for enhancing the plagiarism detection and investigation process. A prototype tool PlaGate, 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 visualization 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 findings discussed in this paper 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 suspicious but also more innocent in the list of detected files. In the context of source-code plagiarism detection, string-matching-based detection systems have shown to detect fewer innocent 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.

This paper also proposes a technique for the investigation of source-code fragments. The idea put into practice with the PlaGate's Query Tool is that plagiarized 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 will serve as the main evidence for proving 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 plagiarized. A comparison of the results gathered from PGQT with human judgment have revealed high correlations between the two. PGQT has shown to estimate well the importance of source-code fragments with regards to characterizing files.

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 for proving 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 be sufficient indicators of the source-code fragments' contribution toward 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.

10 Considerations and Future Work

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 et al. [33]. For now, PlaGate uses parameters that were found as giving good detection results from conducting previous experiments with source-code data sets.

During experimental evaluations, the creation of artificial data sets has been considered 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 behavior 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.

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 preprocessing these are removed from the corpus.

Future experiments are planned to evaluate the performance of PlaGate when keeping these terms and characters during preprocessing. This was also one of the reasons PGDT failed to detect some of the suspicious file pairs.

In the context of source-code plagiarism detection the behavior of PlaGate is far less predictable than string-matching plagiarism detection tools. The behavior of an LSA-based system depends heavily on the corpus itself and on the choice of parameters which are not automatically adjustable. Furthermore, 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 it is not needed to develop any parsers or compilers for programming languages in order for PlaGate to provide detection in programming languages. Therefore, files do not need to parse or compile to be included in the comparison process. Furthermore, 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 more reliable results (i.e., better recall and 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 paper proposes an algorithm for combining the two approaches to improve detection of suspicious files.

PlaGate is currently in prototype mode and future work includes integrating the tool within Sherlock [3]. In addition, work is planned to investigate parameter settings for languages other than Java, and to adjust PlaGate to support languages other than Java. In addition, future works includes research into the automatic adjustment of preprocessing parameters and the development of new similarity measures in order to improve the detection performance of PlaGate.

REFERENCES

[1] G. Cosma and M. Joy, "Towards a Definition of Source-Code Plagiarism," *IEEE Trans. Education,* vol. 51, no. 2, pp. 195-200, May 2008.

[2] G. Cosma and M. Joy, "Source-Code Plagiarism: A U.K Academic Perspective," Research Report, No. 422, Dept. of Computer Science, Univ. of Warwick, Coventry, 2006.

[3] M. Joy and M. Luck, "Plagiarism in Programming Assignments," *IEEE Trans. Education,* vol. 42, no. 2, pp. 129-133, May 1999.

[4] M. Mozgovoy, "Desktop Tools for Offline Plagiarism Detection in Computer Programs," *Informatics in Education,* vol. 5, no. 1, pp. 97-112, 2006.

[5] J.K. Ottenstein, "A Program to Count Operators and Operands for ANSI-Fortran Modules," IBM Technical Report CSD-TR-196, June 1976.

[6] J.K. Ottenstein, "An Algorithmic Approach to the Detection and Prevention of Plagiarism," *ACM SIGCSE Bull.,* vol. 8, no. 4, pp. 30-41, 1976.

[7] M.H. Halstead, "Natural Laws Controlling Algorithm Structure?," *ACM SIGPLAN Notices,* vol. 7, no. 2, pp. 19-26, 1972.

[8] M.H. Halstead, *Elements of Software Science.* Elseveir, 1977.

[9] S.S Robinson and M.L. Soffa, "An Instructional Aid for Student Programs," *SIGCSE Bull.,* vol. 12, no. 1, pp. 118-129, 1980.

[10] A. Aiken, "Moss: A System for Detecting Software Plagiarism," Univ. of California-Berkeley, http://theory.stanford.edu/aiken/moss/, 2005.

[11] S. Schleimer, D. Wilkerson, and A. Aiken, "Winnowing: Local Algorithms for Document Fingerprinting," *Proc. the ACM SIGMOD Int'l Conf. Management of Data,* pp. 76-85, 2003.

[12] M.J. Wise, "YAP3: Improved Detection of Similarities in Computer Program and Other Texts," *Proc. 27th SIGCSE Technical Symp.,* pp. 130-134, 1996.

[13] L. Prechelt, G. Malpohl, and M. Philippsen, "Finding Plagiarisms Among a Set of Programs with JPlag," *J. Universal Computer Science,* vol. 8, no. 11, pp. 1016-1038, 2002.

[14] B. Baker, "On Finding Duplication and Near-Duplication in Large Software Systems," *Proc. IEEE Second Working Conf. Reverse Eng.,* pp. 85-95, 1995.

[15] L.M. Moussiades and A. Vakali, "PDetect: A Clustering Approach for Detecting Plagiarism in Source Code Data Sets," *The Computer J.,* vol. 48, pp. 651-661, 2005.

[16] M.W. Berry, S.T. Dumais, and G.W. OBrien, "Using Linear Algebra for Intelligent Information Retrieval," *SIAM Rev.,* vol. 37, no. 4, pp. 573-595, 1995.

[17] A. Kontostathis and W.M. Pottenger, "A Framework for Understanding LSI Performance," *Proc. SIGIR Workshop Math./Formal Methods in Information Retrieval,* In S. Dominich, ed., July 2003.

[18] A. Kontostathis and W.M. Pottenger, "A Framework for Understanding Latent Semantic Indexing (LSI) Performance," *Information Processing and Management,* vol. 42, no. 1, pp. 56-73, 2006.

[19] S.T. Dumais, "Improving the Retrieval of Information from External Sources," *Behaviour Research Methods, Instruments and Computers,* vol. 23, pp. 229-236, 1992.

[20] P. Nakov, "Latent Semantic Analysis of Textual Data," *Proc. Conf. Computer Systems and Technologies,* pp. 5031-5035, 2000.

[21] S. Kawaguchi, K.P. Garg, M. Matsushita, and K. Inoue, "MUDABlue: An Automatic Categorization System for Open Source Repositories," *Proc. Asia Pacific Software Eng. Conf. (APSEC '04),* pp. 184-193, 2004.

[22] M. Lungu, A. Kuhn, T. Girba, and M. Lanza, "Interactive Exploration of Semantic Clusters," *Proc. Int'l Workshop Visualizing Software for Understanding and Analysis (VISSOFT '05),* pp. 40-45, 2005.

[23] A. Kuhn, S. Ducasse, and T. Girba, "Enriching Reverse Engineering with Semantic Clustering," *Proc. Working Conf. Reverse Eng. (WCRE '05),* pp. 113-122, 2005.

[24] J.I. Maletic and A. Marcus, "Supporting Program Comprehension Using Semantic and Structural Information," *Proc. Int'l Conf. Software Eng.,* pp. 103-112, 2001.

[25] J.I. Maletic and N. Valluri, "Automatic Software Clustering via Latent Semantic Analysis," *Proc. 14th IEEE Int'l Conf. Automated Software Eng. (ASE '99),* 1999.

[26] A. Marcus, A. Sergeyev, V. Rajlich, and J. Maletic, "An Information Retrieval Approach to Concept Location in Source Code," *Proc. IEEE 11th Working Conf. Reverse Eng. (WCRE '04),* pp. 214-223, 2001.

[27] G. Cosma, "An Approach to Source-Code Plagiarism Detection and Investigation Using Latent Semantic Analysis," PhD thesis, Dept. of Computer Science, Univ. of Warwick, Coventry, 2008.

[28] G. Cosma and M. Joy, "Parameters Driving the Performance of LSA for Similar Source-Code File Detection," *ACM Trans. Information Systems,* Apr. 2009.

[29] M.W. Berry and M. Browne, *Understanding Search Engines: Mathematical Modeling and Text Retrieval.* second ed., SIAM, 2005.

[30] S. Deerwester, S.T. Dumais, T.K. Landauer, G.W. Furnas, and R.A. Harshman, "Indexing by Latent Semantic Analysis," *J. Am. Soc. Information Science,* vol. 41, no. 6, pp. 391-407, 1990.

[31] J.W. Tukey, *Exploratory Data Analysis.* Addison Wesley, 1977.

[32] M. Mozgovoy, "Enhancing Computer-Aided Plagiarism Detection," dissertation, Dept. of Computer Science, Univ. of Joensuu, Nov. 2007.

[33] T. Kakkonen, E. Sutinen, and J. Timonen, "Applying Validation Methods for Noise Reduction in LSA-Based Essay Grading," *WSEAS Trans. Information Science and Applications,* vol. 9, no. 2, pp. 1334-1342, 2005.

Georgina Cosma received the PhD degree in computer science from the University of Warwick. She is currently a lecturer at P.A. College in Cyprus. Her fields of interest include information retrieval and educational technology.

Mike Joy received the MA degree in mathematics from Cambridge University, the MA degree in postcompulsory education from the University of Warwick, and the PhD degree in computer science from the University of East Anglia. He is currently an associate professor at the University of Warwick, where his research interests center around educational technology and computer science education.