PDE4Java: Plagiarism Detection Engine For Java Source Code: A Clustering Approach

Ameera Jadalla and Ashraf Elnagar
Department of Computer Science
University of Sharjah,
P O Box 27272, Sharjah, UAE
ameera.jadalla@gmail.com; ashraf@sharjah.ac.ae

Abstract:
The educational community across the world is facing the increasing problem of plagiarism. This widespread problem has motivated the need of an efficient, robust and fast detection procedure that is difficult to be achieved manually. The Plagiarism Detection Engine for Java (PDE4Java), introduced in this paper, detects code-plagiarism by applying data mining techniques. Besides computing the similarity between each pair of programs; it finds clusters of suspicious plagiarized programs. The engine consists of three main phases; Java tokenization, similarity measurement and clustering. It was designed for Java source code, but the optional default tokenizer makes it flexible to be used with any other programming language. The simulation results of PDE4Java showed a comparable performance to that of JPlag and it outperformed the expectations when compared to the domain experts' (instructor/TA) findings.

1 Introduction

The concerns about source code plagiarism increasingly rose since 1977. A survey performed in 2002 on a sample of students at Monash and Swinburne universities shows that 85.4% of 137 Monash University students and 69.3% of 150 Swinburne University students admitted to having engaged in academic dishonesty, [2]. The assessment of students' programming submissions has an important effect on the whole computing educational procedure. It is of a great importance to evaluate the programming skills of each student, but the evaluation results become misleading and unreal due to the plagiarism problem. One of the successful policies to prevent and decrease this problem is plagiarism detection. Manual detection was found to be inefficient but it is effort and time consuming (evaluating n programs requires $O(n^2)$ cost). Hence, an automated plagiarism detection system becomes essential, which lead to the emergence of a series of plagiarism detection systems started since mid-1970s, [4].

Parker and Hamblen, [9], defined plagiarism in software as: "a program which has been produced from another program with a small number of routine transformations". The most challenging aspect in code-plagiarism detection is the techniques that the implicated students tend to use to disguise the copied code in order to mislead the grader. Arwin, [2], lists the most common disguises; *which* are changing comments, changing formatting, changing identifiers, changing the order of operands in expressions, changing data types, replacing expressions by equivalents, adding redundant statements, changing the order of time-independent statements, changing the structure of iteration statements, changing the structure of selection statements, replacing procedure calls by the procedure body, introducing no structured statements, combining original and copied program fragments and the translation of source code from one language to another.

The existing plagiarism detection systems are classified into two main categories based on the underlying metrics that they use to find similarity; these are the attribute-counting systems and the structure metric systems, [8].

In the attribute-counting systems each program is represented by several quantities such as number of operands, operators, blanks, control statements, loop statements, conditional statements and variables. Then these values are used to compare each pair of programs in the dataset to compute their similarity. This approach is no longer used because it can be either very insensitive (two programs might share the same measures while they completely differ in the logic) or very sensitive as it ignores the program's structure [2]. It fails easily with the common disguises that students might use such as blanks insertion or deletion that does not affect the structure of the program, (see [12] for details).

On the other hand, the structure metric systems, that were recently used, were shown to have high performance in detecting source-code plagiarism, [12]. These systems use one of four techniques: string matching, [8], abstract syntax tree (AST), [5], program dependence graph, [6], and tokenization, [7],which we adopt in this work. Next, we describe four examples of recent systems.

JPlag, [7]: A Token-based system that is freely available on the Internet for academic use. It supports four different programming languages; Java, C, C++ and scheme and it is platform independent. It output the similarity scale between each pair of programs in the dataset. The major limitation of JPlag is that it requires parsing the dataset; if a program fail to be parsed it will be omitted from the dataset, [4].

MOSS (Measure of Software Similarity), [1]: A free available plagiarism detection system for academic usage only. MOSS supports eight different programming languages and two platforms; UNIX and Windows. It uses a string matching algorithm to divide the source-code programs into k-grams, hash them, select a subset of these hashes as fingerprints and finally

compare these fingerprints, [8, 10].

YAP series (Yet Another Plague), [14]: Token-based systems that treat programs as sequence of strings. The last version of YAP (YAP3) introduces a totally novel algorithm to face the presence of block-moves in programs. Namely: the Running-Karp-Rabin Greedy-String-Tiling (RKR-GST) algorithm, [13].

This paper presents code-plagiarism as a data mining problem and proposes a token based plagiarism detection engine for Java. The main contribution of this paper is of twofold. First, fast similarity computations as the engine indexes the tokens of the dataset and uses a technique similar to that of the search engines for fast and efficient retrieval. Second, besides displaying the pairwise similarity of the programs in the dataset, it finds clusters of suspicious plagiarized programs. Each cluster contains programs that are very close to each other. Clusters make it easier for instructors/graders to analyze and study the system findings.

The remainder of the paper is organized as follows: In Section 2, we present PDE4Java such that each subsection describes a phase of the engine, in details, with a brief background about the underlying used techniques. In Section 3, we describe the experimental results and the performance of the proposed system against the results of the domain expert and to the state-of-the-art JPlag plagiarism detection system. We conclude with our future work in Section 4.

2 Plagiarism Detection Engine

This section describes PDE4Java in detail. Each subsection presents one phase of the system with a brief background of the underlying technique. PDE4Java accepts input files in two different formats: Java source code or tokens representation of the programs. The output consists of two parts: clusters of the suspected plagiarized programs and the pairwise similarity for each pair of programs in the dataset.

```
class Hello {
public static void main(String args[ ])
{
// printing...
System.out.println("HELLO JAVA");
}
}
```

Table 1: Hello.java

Keyword	Identifier	Separator
Keyword	Keyword	Keyword
Identifier	Separator	Identifier
Identifier	Separator	Separator
Separator	Separator	keyword
Identifier	Separator	Separator
Separator	EOF	EOF

Table 2: Hello.txt

2.1 Tokenization

This is the first phase of the system where each program is transformed into a stream of representative tokens. To extract the tokens from the source code, a parser (tokenizer) is required. Java is the default programming language that PDE4Java was designed for. The Java tokenizer was generated using JLex and Java CUP. JLex takes a specially-formatted specification file containing the details of a lexical analyzer, and then creates a Java source file for the corresponding lexical analyzer. Java CUP is a system for generating LALR parsers from simple specifications. Using these tools with Java grammar, we generated the required tokenizer for Java. If another tokenizer is to be used, then the user should provide the system with the tokens representation of the dataset, directly, instead of the source code files. By the end of this phase, we get rid of two common plagiarism disguises; format alteration (such as blanks insertion or deletion) and identifier renaming. Tables 1 and 2 show an example for Java code and its tokens' representation.

2.2 N-Gram Representation

Each program was represented as a stream of tokens; a structural representative version. In this phase of the engine, we will break that sequence of tokens into smaller blocks, or what is called the n-grams representation. N-gram is a sub-string of size n from a given sequence. Then, each program would be a sequence of 4-gram tokens; each new token consists of 4 tokens of the original file. To accomplish this task, the sliding window technique with n of size four is used. To illustrate how this technique works, see Table 3. One advantage of using the n-gram representation is that any change in the tokens sequence would affect only a small number of neighboring n-grams, which reduce the problem of statement reordering or code insertion that might be used as disguising for the copied code. It was reported that the most appropriate size of n-gram for a plagiarism detection system is 4, [3]

Tokens	IDs	4-grams
Keyword	0	0260
Identifier	2	2600
Separator	6	6002
Keyword	0	0026
Keyword	0	0262
Identifier	2	
Separator	6	
Identifier	2	

Table 3: Each Token is given an ID that is used to create the n-gram.

2.3 Inverted Index Construction

Inverted index, which is used in search engines, is a data structure that maps words to their locations in a set of documents. It consists of two parts: a lexicon and a series of inverted lists, [3]. Each inverted list stores a list of documents that are mapped to a lexicon (word). The main goal of the indexing is to optimize the speed and efficiency of a search query. In our application, each pair of programs is compared such that for each n-gram (token) in program A, we check whether it exists or not in program B. It is inefficient to compare each token of A with all tokens in B; hencefore, we index all n-gram tokens for all programs in the dataset. The lexicon part stores all distinct n-gram that appears in the programs of the dataset, while the inverted lists store information about the source-code files that contain each n-gram with the frequency of that n-gram in each file. For example, in Table 4, the n-gram that equals to 0026 appears twice in source-code file number 2 and one time in source-code file number 5.

n-gram		**Frequency; List**
0260	$\Rightarrow$	$\{(2,3)\}$
2600	$\Rightarrow$	$\{(5,1)\}$
6002	$\Rightarrow$	$\{(1,1),(3,2),(4,1)\}$
0026	$\Rightarrow$	$\{(2,2),(5,1)\}$
0262	$\Rightarrow$	$\{(3,1),(5,2)\}$

Table 4: N-grams and its frequency list.

2.4 Similarity Measurement

There are a limited number of tokens in any programming language, but as PDE4Java is using the 4-gram representation instead of normal tokens, there are several different tokens' combinations. As a result, the dataset consists of asymmetric n-grams files. The most important factor to consider when comparing two asymmetric files is to ignore the matching absences (00 matches); otherwise the similarity value would be, incorrectly, very high. Jaccard coefficient is commonly used for handling objects with asymmetric attributes and it was found to give the lowest error rate compared with seven well-known similarity measures, [8]. The following equation defines the Jaccard coefficient, where: f_{11} represents the total number of n-gram blocks that appeared in programs A and B, f_{01} represents the total number of n-gram blocks that appeared in B but not in A, f_{10} represents the total number of n-gram blocks that appeared in A but not in B:

$$\texttt{Jaccard coefficient} = \frac{f_{11}}{f_{01} + f_{10} + f_{11}}$$

2.5 Clustering

Clustering is the most significant contribution of PDE4Java. It is an important feature that provides instructors/graders with fast and easy analysis tool for the system findings. DBSCAN is a density-based clustering algorithm that produces a partitional clustering, in which the number of clusters is automatically determined by the algorithm. Points in low-density regions are classified as noise and omitted; thus, DBSCAN does not produce a complete clustering. DBSCAN is a center-based approach where a point is considered as a center of a cluster if the number of points within a specified radius, `Eps`, of that point exceeds a minimum threshold, `MinPts`, [11].

The plagiarism detection problem entails a number of unique properties. First, the majority of the programs in the dataset should not belong to any cluster which means that partitional clustering is needed. Second, it is difficult and impractical to pre-determine the number of clusters. Third, the resulting clusters might vary in their sizes and shapes. Finally, we need to set our preferred similarity value that is used for clustering. DBSCAN features matches with those of the system. The main issue when using DBSCAN is how to determine the two parameters: `Eps` and `MinPts`. In our case, two points (programs) at least are enough to produce a cluster, hence MinPts should be set to two. In PDE4Java, `Eps` is a user specified value that is set at `0.8` by default. This value is determined based on the simulation results, which showed that `0.8` matches with the domain expert's findings.

3 Simulation and Performance Evaluation

The system's output consists of two parts: pairwise similarity of the programs and clusters of the suspicious plagiarized programs. Hencefore, we applied two different testing procedures to evaluate each part. The dataset that is used for pairwise similarity measuring consists of 34 Java source-code files that vary in the logical executable lines of code (SLOC-L) from 20 up to 63 SLOC. Figure1 illustrates 34 pairs of programs and the percentage similarity of each pair according to JPlag and PDE4Java.

For the clustering performance evaluation, the system was tested using seven different datasets, each of which contains students' submissions of Java assignments. Table 5 shows these datasets, number of programs within each dataset, the domain expert and the system reports, and the maximum similarity value detection took place. We manually verified the clusters reported by PDE4Java, which was true for all cases. For example, dataset 1.2 contains a collection of 28 Java source-code files, the domain expert did not report any plagiarism, while the system has detected two programs that share 90% of the source-code.

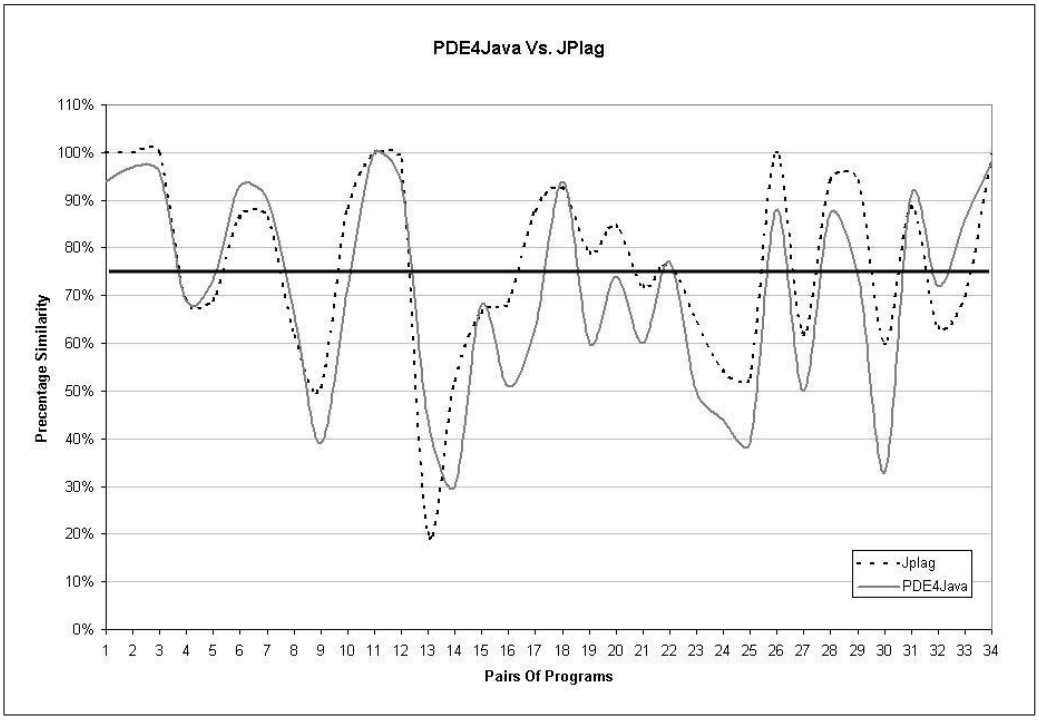

Figure 1: Comparison between JPlag and PDE4Java findings.

The results showed that there were two cases of plagiarism in datasets 1.2 and 3.1 that were detected only by the system. In dataset 1.3, the domain expert has found a plagiarism group of two source code files only while the system has detected three source code files, which includes the domain expert findings.

Dataset	Size	Similarity Value	Domain Expert	System Report
1.1	49		clean	clean
1.2	28	0.91	clean	$\{13, 9\}$
		0.83	$\{16, 21\}$	$\{16, 21\}$
1.3	34	0.87	$\{16, 21\}$	$\{9, 16, 21\}$
2.1	37		clean	clean
2.2	39	0.91	$\{22, 11\}$	$\{22, 11\}$
		0.90	$\{9, 16, 4, 13\}$	$\{9, 16, 4, 13\}$
2.3	38		clean	clean
3.1	29	0.86	clean	$\{21, 14\}$
3.2	38	0.95	$\{9, 13\}$	$\{\ 9,\ 13\}$

Table 5: Domain Expert Report VS. System Report.

4 Conclusion and Future Work

Plagiarism in source-code submissions is a serious problem that has motivated researchers to find effective automated detectors. This paper proposed a plagiarism detection engine for Java source-code files (PDE4Java). Mainly, The system computes and displays the similarity value between each pair of programs in a dataset. Compared to the well-known state-of-the-art plagiarism detection systems, PDE4Java has one extra feature, which is adaptive reporting of the clusters of suspicious plagiarized programs. The performance of the system pairwise similarity measurement showed promising results compared to JPlag findings. To further evaluate the findings of the system, it was compared with the reports of domain experts (TA/grader). In some tests, the system reported more clusters than the domain expert. Manual verification of the extra clusters confirmed the system output to be true positives.

The problem of the source-code plagiarism became more complicated with the availability of the Internet and the increasing number of the WWW sites. To detect web plagiarism, an efficient approach to optimize the speed and the accuracy of the detection process is required. This is a challenging problem which we are tackling as an extension of this work.

References

[1] Alex Aiken. Moss: A system for detecting software plagiarism, 2005.

[2] Christian Arwin and S.M.M. Tahaghoghi. Plagiarism detection across programming languages. *Proceedings of the 29th Australasian Computer Science Conference*, 48:277–286, 2006.

[3] Steven Burrows, Seyed M. M. Tahaghoghi, and Justin Zobel. Efficient and effective plagiarism detection for large code repositories. *Proceedings of the Second Australian Undergraduate Students' Computing Conference (AUSCC04)*, pages 8–15, 2004.

[4] Fintan Culwin, Anna MacLeod, and Thomas Lancaster. Source code plagiarism in UK HE computing schools, issues, attitudes and tools. Technical report, South Bank University (SBU) SCISM Technical Report, 2001.

[5] Young-Chul Kim, Yong-Yoon Cho, and Jong-Bae Moon. A plagiarism detection system using a syntax-tree. *International Conference on Computational Intelligence*, 1:23–26, 2004.

[6] Chao Liu, Chen Chen, Jiawei Han, and Philip S. Yu. GPLAG: Detection of software plagiarism by program dependence graph analysis. *The 12th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, pages 872–881, 2006.

[7] Michael Philippsen Lutz Prechelt, Guido Malpohl. Finding plagiarism among a set of programs with JPlag. *Journal of Universal Computer Science*, 8(11):1016–1038, 2002.

[8] Lefteris Moussiades and Athena Vakali. PDetect: A clustering approach for detecting plagiarism in source code datasets. *The Computer Journal*, 48(6):651–661, 2005.

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

[10] Saul Schleimer, Daniel S. Wilkerson, and Alex Aiken. Winnowing: local algorithms for document fingerprinting. *Proceedings of the 2003 ACM SIGMOD international conference on Management of data*, pages 78–85, 2003.

[11] Pang-Ning Tan, Michael Steinbach, and Vipin Kumar. *Introduction to Data Mining.* Addison Wesley, Pearson Education, Boston, 2006.

[12] Kristina L. Verco and Michael J. Wise. Software for detecting suspected plagiarism: Comparing structure and attribute-counting systems. *Proceedings of the First Australian Conference on Computer Science Education*, pages 81–88, 1996.

[13] Michael J. Wise. String similarity via greedy string tiling and running karp-rabin matching, 1993.

[14] Michael J. Wise. YAP3: Improved detection of similarities in computer program and other texts. *ACM SIGCSE*, 28:130–134, 1996.