Journal of Engineering
and Applied Science

RESEARCH Open Access

Classification feature sets for source code plagiarism detection in Java

Eman Hosam*, Mayada Hadhoud, Amir Atiya and Magda Fayek

*Correspondence:
eman.hosam@eng.cu.edu.eg

Computer Engineering Department, Faculty of Engineering, Cairo University, Cairo, Egypt

Abstract

In programming learning environments, the pressure of delivering many programming assignments makes plagiarism the easiest solution. This highly threatens the learning process; therefore, the need of an automatic, fast, and accurate detection of source code plagiarism becomes essential. To detect whether a pair of Java files is plagiarized, this paper proposes four classification feature sets: (i) structural histogram features, histogram-based features for summarizing similarity matrices; (ii) lexical per-class features, extracted from a lexical similarity matrix between the classes of the two compared files based on character 3-grams; (iii) structural counting features, twelve counting features representing the code structure; and (iv) modified original features: a set of modifications on the features of the used baseline. The results show that the best feature sets in F-measure are the structural histogram features and the lexical per-class features combined, which improve the F-measure by 4% compared to the baseline. The added features slow down the execution time. However, it is still efficient, given that it can classify 70k pairs in 23 min. In addition, we partially re-annotated the SOurce COde Re-use dataset. After the re-annotation, the F-measure of both the baseline and our work is improved, and our work achieves an F-measure of 93.6%, which is 7.5% higher than the new F-measure of the baseline. In addition, some remarks and recommendations are provided for using the SOurce COde Re-use dataset as a benchmark.

Keywords: Source code plagiarism detection, Source code reuse detection, Software similarity detection, Machine learning, Classification, Similarity matrix

Introduction

Source code plagiarism highly complicates the learning process. Some students find it easier to plagiarize programming assignments than spending time to solve them. This makes the instructors also detectives! They should not only focus on teaching students and correcting their mistakes, but they also have to spend huge time to make sure that each student code is truly his code and not plagiarized. Detecting source code plagiarism is very important, not only for the fairness between students but also for making them achieve the learning outcomes of the assignments. Therefore, source code plagiarism detection becomes very essential.

Due to numerous source codes available online and source codes available for the same course in previous years, the task of manually comparing each source code pair becomes infeasible because the number of all possible pairs of N files is: $N * (N - 1) / 2$ (e.g., if $N = 3000$, then the number of program pairs to compare = 4.5 million pairs!). The complexity of detection = $O(N^2)$ times the complexity of one file pair comparison; hence, the computational time is essential, and the need for automatic, fast, and accurate tools for detecting source code plagiarism highly arises. However, not all programs with high similarity are a result of plagiarism. That is why instructors should manually review the results of the plagiarism tools and give the final decision. Therefore, the main target of plagiarism tools is to shrink the number of program pairs (i.e., keep only the highly similar pairs) to be manually reviewed by instructors.

Using raw text plagiarism detection techniques for source code plagiarism detection [1, 2] produces poor results. The text-based detection approach is programming-language independent, but it can be confused by trivial attacks such as identifiers' renaming. Programming languages have a structure different from human languages. In programming languages, there may be changes in the text, but the same semantics are preserved (e.g., renaming identifiers, replacing for with while loops [3–6]). There are mainly two categories of techniques for source code plagiarism detection: attribute-based and structure-based techniques [7].

Attribute-based detection techniques extract numeric characteristics from each file (e.g., number of classes, functions, tabs, underscores). Hence, there will be a feature vector for each file, then, several ways can be applied to estimate the similarity between these vectors (e.g., classification [8–10] or clustering [11]). The results of attribute-based techniques are better than text-based, but attribute-based usually have many false positives because they depend only on numeric features.

Structure-based detection techniques utilize the structure of the targeted programming language. First, it generates a representation of the structure of each file. This representation can be sequential or hierarchical [12]. Sequential representations can be token strings [5, 7, 13] or low-level representation [4, 14]. Hierarchical representations can be trees (e.g., abstract syntax tree [15, 16], parse trees [17]) or graphs (e.g., program dependency graph [6], control flow graphs [18]). After generating a representation for each file, a similarity measure should be used (e.g., subgraph isomorphism algorithm [19, 20] for graphs, and tree kernel algorithm [21] for trees). Structure-based techniques are more accurate, but time-consuming and programming-language dependent.

The gap that we target in our work is enhancing source code plagiarism detection results by incorporating different feature sets that better represent code similarities, while being extracted in an efficient time. This paper proposes four feature sets:

1 Structural histogram features: For measuring the similarity of function signatures, a histogram-based method is used to summarize similarity matrices with considering *all* matrix values.
2 Lexical per-class features: For measuring the lexical similarity of classes, a lexical similarity matrix is built between classes, and we proposed two new concepts: *candidates* and *extreme ranges* for representing the matrix particularly for the similarity comparison.
3 Structural counting features: twelve counting features, representing how similar the code structure is, are extracted (e.g., number of classes, functions, loops).

4 Modified original features: a modified version of the structural features of Ganguly et al. [8] is presented, incorporating the candidates and extreme ranges concepts.

The contributions of this paper are as follows:

- It presents four feature sets for source code plagiarism detection.
- It chooses structural histogram features and lexical per-class features combined to be our best-proposed feature sets in results.
- An increase of 4% in F-measure is achieved compared to (Ganguly et al., 2018) [8] on the SOurce COde Re-use (SOCO-14) dataset [22].
- It performs a time analysis, and the results demonstrate that the classification time is slower due to extracting the proposed feature sets, but still efficient (23 min for classifying 70K pairs).
- It provides a partial re-annotation of the SOCO-14 dataset [22].
- After the re-annotation, the results of both the baseline (Ganguly et al., 2018) [8] and our work are improved. An increase of 7.5% in F-measure is achieved compared to (Ganguly et al., 2018) [8]. An F-measure of 93.6% is reached.
- It provides some remarks on using the SOCO-14 dataset [22].

The rest of this paper is organized as follows. Section 2 presents the four proposed feature sets with the detailed extraction steps. Section 3 presents the experimental results, in addition to the performed time and dataset analysis. Finally, Section 4 presents the conclusion and future work.

Methods

In our work, four feature sets are proposed for classifying if a Java program pair is plagiarized. In the structural histogram features, histograms are used for summarizing function prototypes similarity matrices. On the other hand, the lexical per-class features represent the lexical similarity between each class pair, while introducing and incorporating the candidates and extreme ranges of similarity matrices. Then, the structural counting features represent the structure of the two compared programs in terms of counting some code structure properties such as: number of loops, conditional, and functions. Finally, the modified original features propose a modified version of the structural features of Ganguly et al. [8] incorporating the candidates and extreme ranges of the similarity matrices. The following subsections present each feature set in detail.

Structural histogram features

The target of the structural features is to measure how similar the function signatures of the two compared files. The structural features proposed by Ganguly et al. [8] are extracted from two similarity matrices for function signatures (one for data types and the other for identifiers). They built the similarity matrices for only function signatures, not the body, for keeping the time efficient. You can read more about how the two matrices are built in Ganguly et al. [8], but in general, the two matrices represent how much the functions signatures of the compared files are similar in terms of their data types and identifiers names.

The structural features extracted by Ganguly et al. [8] from the two similarity matrices are the minimum, maximum, and average of each matrix (six features). Aggregating the similarities of each matrix to only the minimum, maximum, and average is exaggerated compaction, ignoring the similarity value of the other cells. That is why we propose a more informative aggregation of the matrix based on the histogram of similarity values.

To build the histogram of similarity values, we divided the similarity range (i.e., from zero to one) which are calculated in [8] into “NPartition” partitions (i.e., sub-ranges). An example of a similarity matrix and its extracted histogram vector using “NPartition” = 4 is shown in Fig. 1. Each matrix cell contains the similarity between the corresponding function signature pair (column and row), as shown in Fig. 1a. The count of similarity values within the range of each partition is calculated and normalized by dividing by the total count of cells, as shown in Fig. 1b. The 4-dimensional feature vector “Hf_Vector” consists of the normalized count of similarity values of each partition. A high value of “Hf_Vector[0]” indicates the high dissimilarity between function pairs, while a high value of “Hf_Vector[3]” indicates high similarity.

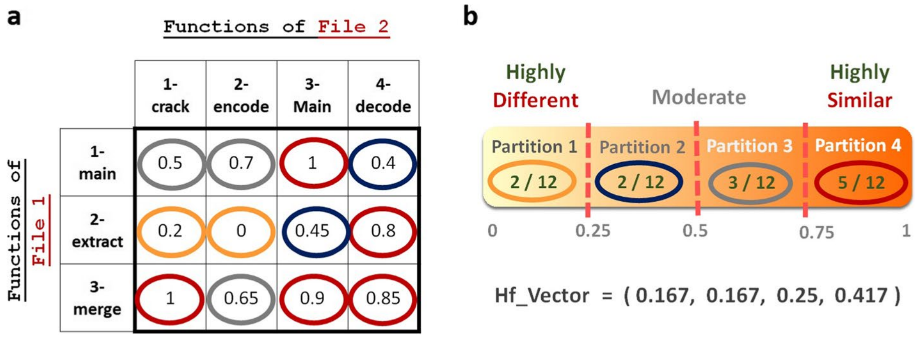
For the two similarity matrices of function signatures [8], we extracted two histogram feature vectors that include eight features in total. These features, structural histogram features, reflect the similarity between function signatures better than only the minimum, maximum, and average.

Lexical per-class features

The target of the lexical per-class features is to measure the lexical similarity between the classes of the two compared files. In the lexical per-class features, the lexical similarity between the two files to compare is measured by (i) calculating the lexical similarity for each class pair of the two files separately, (ii) building a similarity matrix between class pairs, and (ii) summarizing this matrix into three informative features representing the overall lexical similarities between the two files. Before explaining how to extract the lexical per-class features, we need to introduce the following definitions: lexical per-class similarity matrix, candidates of a similarity matrix, and histogram extreme ranges.

Lexical per-class similarity matrix

The per-class similarity matrix is a matrix that includes the similarity of each class pair in the two files to compare. The rows of the matrix represent the classes of the

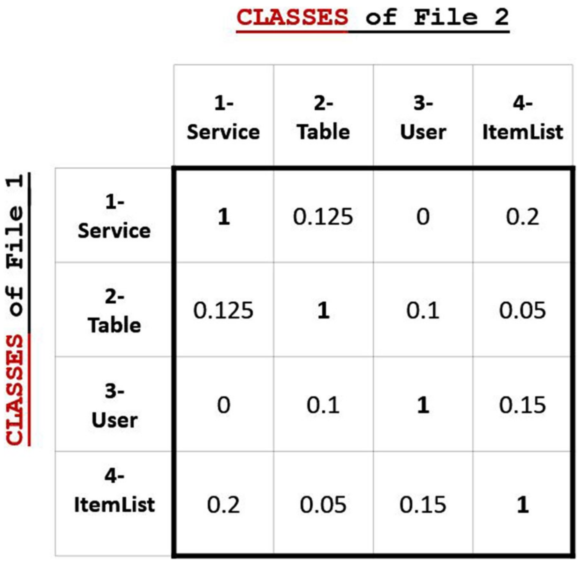


	1-crack	2-encode	3-Main	4-decode
1-main	0.5	0.7	1	0.4
2-extract	0.2	0	0.45	0.8
3-merge	1	0.65	0.9	0.85

Hf_Vector = (0.167, 0.167, 0.25, 0.417)

Fig. 1 **a** A similarity matrix. **b** Its histogram features first file while the columns represent the classes of the second file as shown in Figs. 2 and 3a. Each cell represents the similarity between one class pair.

The lexical per-class similarity matrix is a per-class similarity matrix whose cells contain the lexical similarities between class pairs. The lexical similarity between a class pair is calculated using the cosine similarity of the binary vector representation of the character 3-grams of the two classes (as calculated in Ganguly et al. [8] but for each class pair separately, not for the entire files as a one unit). Some pre-processing steps are performed such as removing java keywords, lowering case, and removing white spaces.

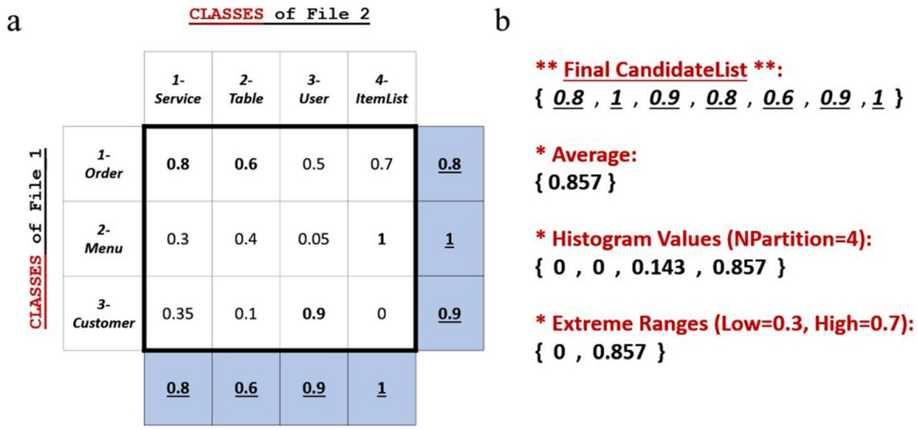


CLASSES of File 2

CLASSES of File 1

	1- Service	2- Table	3- User	4- ItemList
1- Service	**1**	0.125	0	0.2
2- Table	0.125	**1**	0.1	0.05
3- User	0	0.1	**1**	0.15
4- ItemList	0.2	0.05	0.15	**1**

Fig. 2 A lexical per-class similarity matrix of two verbatim-copy files



a

CLASSES of File 2

CLASSES of File 1

	1- Service	2- Table	3- User	4- ItemList	
1- Order	**0.8**	**0.6**	0.5	0.7	0.8
2- Menu	0.3	0.4	0.05	**1**	1
3- Customer	0.35	0.1	**0.9**	0	0.9
	0.8	0.6	0.9	1	

b

** Final CandidateList **:
{ 0.8 , 1 , 0.9 , 0.8 , 0.6 , 0.9 , 1 }

* Average:
{ 0.857 }

* Histogram Values (NPartition=4):
{ 0 , 0 , 0.143 , 0.857 }

* Extreme Ranges (Low=0.3, High=0.7):
{ 0 , 0.857 }

Fig. 3 **a** A similarity matrix. **b** Its candidates and features

Candidates of a similarity matrix

Considering all matrix cells in calculating the features is misleading. For clarifying this, consider the similarity matrix of two example verbatim-copy files shown in Fig. 2. The diagonal cells contain ones because they represent the lexical similarity of the same class; however, the other cell values contain very low similarity values because they compare each class with the other different classes. These low values will be counted in the histogram of dissimilarity partitions, indicating that the files are dissimilar despite being verbatim copies. This example clarifies that, in any similarity matrix, considering all cells in calculating the features is misleading.

Instead, this paper proposes a method to discard the misleading cells and select a subset of the matrix cells, called "candidates," that better and more indicatively represent the matrix. Any feature extracted from a similarity matrix (e.g., minimum, maximum, average, or histogram features) is better to be extracted from the candidates rather than all matrix cells.

The proposed way to extract the candidates of a similarity matrix is as follows: for each row/column which represents the classes of the first/second file, take only the maximum value of this row/column into the candidate list. If the maximum value of row I is in column J (cell of row I and column J), this indicates that class J of the second file is the most similar class to class I in the first file. Figure 3a shows an example of a lexical per-class similarity matrix and its candidates (the outside underlined cells). The minimum, maximum, average, and histogram values that are calculated from the candidate list are shown in Fig. 3b. It is worth noting that any feature normalization division is by seven (the count of the candidates).

It is worth noting that the maximum similarity of both each row and each column is included in the candidates. This is because, as noticed in Fig. 3a, although some classes are mutually the most similar to each other (e.g., "Order" and "Service"), there are some classes that are not mutually the most similar to each other (e.g., the most similar to "Table" is "Order"; however, the most similar to "Order" is "Service"). That is why we included the maximum of both each row and each column.

Histogram extreme ranges

In structural histogram features (Section 2.1), the histogram features were calculated using four partitions as shown in Fig. 1. However, the extreme partitions (i.e., the first and last partitions) are the most indicative because they represent the lowest and highest similarities. We called the histogram values of these two partitions, histogram extreme ranges. The used "Low" and "High" similarity thresholds of the extreme ranges are chosen as 0.3 and 0.7 respectively. Hence, the candidates of the low extreme range are the candidates with values from 0 to 0.3, and the candidates of the high extreme range are the candidates with values from 0.7 to 1. Equation 1 and 2 show how to calculate the low and high extreme ranges from the candidates, respectively. In Fig. 3b, the histogram extreme ranges are also calculated.

$$ExtRange(Low) = \frac{Count(Candidates <= Low)}{Count(AllCandidates)} \tag{1}$$

$$ExtRange(High) = \frac{Count(Candidates >= High)}{Count(AllCandidates)} \quad (2)$$

Extracting the lexical per-class features

Finally, putting it all together, Table 1 explains the main steps of extracting the proposed lexical per-class features. First, the class codes are extracted from the two files. Then, the lexical per-class similarity matrix is built, and the candidates of the matrix are extracted. Finally, the candidates are used for extracting the three lexical per-class features: the average of the candidates and their two histogram extreme ranges. In brief, the lexical per-class features represent the code similarity between the class pairs of the two files and can be used to detect source code plagiarism.

Structural counting features

The target of the structural counting features is to measure how similar the high-level code structure of the two compared files in terms of number of functions, classes, etc. The twelve structural counts shown in Table 2 are extracted from each compared program file. After extracting these counts for each file separately, Eq. 3 is used for each corresponding two counts of the two compared files to calculate their similarity, where "Count1" and "Count2" in the equation represent the count of a specific feature (e.g., number of classes) in File 1 and File 2 respectively. The twelve similarity values calculated for the twelve counts of the two files are called structural counting features.

$$Similarity(Count_1, Count_2) = \frac{Min(Count_1, Count_2)}{Max(Count_1, Count_2)} \quad (3)$$

To understand how Eq. 3 represents the similarity, assume that the number of classes in File 1 and File 2 is 3 and 5 respectively, then the two files are similarly having the count of 3 classes, but File 2 has 2 more classes. Hence, the minimum (i.e., the numerator) represents the count similarity, and the maximum (i.e., the denominator) is for normalization.

Table 1 The extraction steps of the lexical per-class features

Input:	**A file pair: *File1*, *File2***
Output:	**The Lexical Per-Class Features: *LexicalPerClassFeatures***
Procedure:	1. Extract the list of class codes of *File1*: *ClassList1* 2. Extract the list of class codes of *File2*: *ClassList2* 3. Build a lexical per-class similarity matrix, *SimMatrix*, where: SimMatrix[I][J] = LexicalSim(ClassList1[I], ClassList2[J]) 4. Extract the candidate list of *SimMatrix*: *CandidateList* 5. Calculate the Histogram Extreme Ranges of *CandidateList*: *ExtRanges* 6. Add the Average of *CandidateList* into *LexicalPerClassFeatures* (1 feature) 7. Add *ExtRanges* into *LexicalPerClassFeatures* (2 features) 8. Return *LexicalPerClassFeatures* (total 3 features)

Table 2 The structural counts extracted from each file separately

Structural count	Description
Number of classes	Total number of classes, including the nested
Number of interfaces	Total number of interfaces, including the nested
Number of subclasses	Total number of derived classes
Number of functions	Total number of function definitions
Number of loops	Total number of loops: for, while, do-while, or for-each
Number of conditionals	Total number of conditional statements: if statements, switch cases, or ternary operators
Number of function calls	Total number of function calls "C1.add(C2).add(C3);" increments this count by 2.
Number of class fields	Total number of class fields "float X = 0, Y;" increments this count by 2.
Number of variable declarations	Total number of variable declarations Note: Class fields are excluded from this count.
Number of assignment statements	Total number of assignment statements Note: The assignments in declaration lines are included.
Number of comments	Total number of comments
Number of string literals	Total number of string literals "System.out.println("out");" also increments it by 1.

Modified original features

In the modified original features, we propose a modified version of the original features of Ganguly et al. [8] which measure the similarity of the code, the structure, and the style. The original features consist of eight features: (i) one lexical feature which compares the character 3-grams of the two files, (ii) six structural features extracted from two similarity matrices for function signatures (one matrix for the data types and the other for the identifiers), and (iii) one stylistic feature which represents how similar the writing style of the two files is. In the modified original features, we modified only the structural features (which means that the lexical feature and the stylistic feature are not modified).

Unlike the original features that extract the structural features from all the cells of the two similarity matrices, the features are extracted from the candidates of each matrix. In candidates, as explained before, we discard the misleading matrix cells and choose only the most informative cells.

The structural features of the original features were the minimum, maximum, and average of each matrix. Instead of using the minimum and maximum, the two histogram extreme ranges of the candidates of each matrix are used. Therefore, the structural features of the modified original features are the two histogram extreme ranges and the average of the candidates of each matrix (total of six structural features).

Finally, we noticed that including the main function in the similarity matrices of function signatures could be misleading because the function signature of the main functions is the same in Java, so their similarity cell will be always one (which makes the maximum feature of the original features equal to one in nearly all the instances, hence, meaningless).

To sum up, the modified original features include eight features: (i) the same one lexical feature [8], (ii) six structural features for the histogram extreme ranges and the

Table 3 The target and description of the proposed feature sets

Feature set	Target	Count	Description
Structural histogram features	Summarizing the function prototype similarity matrices using histograms (instead of the min., max., and avg.).	8	There are 2 similarity matrices (one for types and the other for identifiers). Each matrix has 4 features representing the normalized count of the 4 histogram partitions of similarity values.
Lexical per-class features	Comparing each class pair in the two programs lexically by the cosine similarity of their character 3-grams. *Note*: The candidates and extreme ranges of the similarity matrix are used.	3	There is 1 feature for the average of the candidate list extracted from the class similarity matrix, and 2 features for the two histogram extreme ranges of the candidate list
Structural counting features	Comparing the two programs based on some counting features representing the program structure such as number of classes, functions, and loops.	12	The 12 counts extracted from each program: classes, interfaces, subclasses, functions, loops, conditionals, function calls, class fields, variable declarations, assignments, comments, string literals. For each 2 counts, the similarity feature is the minimum count over the maximum count.
Modified original features	Proposing a modified version of Ganguly et al. [8] structural features using candidates and histogram extreme ranges of similarity matrices.	8	The 6 structural features are: 2 for the histogram extreme ranges and 1 for the candidates' average for the 2 similarity matrices. The same 1 lexical feature and 1 stylistic feature of Ganguly et al. [8] are used.

Table 4 The number of files and pairs of SOCO-TRAIN

Number of files	Number of pairs	Number of plagiarized pairs
259	33,411	84

average of the candidates for the two similarity matrices, and (iii) the same one stylistic feature [8]. Table 3 lists the four proposed feature sets, with the target and description of each of them.

Results and discussion

In this section, first, the used dataset and performance measures are presented. Then, we present the performed experiments and discuss their results compared to the two used baseline approaches. Finally, the performed time analysis and dataset analysis and remarks are discussed.

SOCO-14 dataset

SOCO-14 [22] is a dataset used for detecting monolingual plagiarism. You can get the dataset by contacting its authors [22]. The Java files of the dataset are targeted in this paper. SOCO-14 consists of two sets: the training set (SOCO-TRAIN) and the test set (SOCO-TEST). SOCO-TRAIN contains 259 Java files (from 000.java to 258.java). The total number of file pairs is: 33K pairs, and the number of the plagiarized pairs is 84 as shown in Table 4.

SOCO-TEST is divided into 6 categories: A1, A2, B1, B2, C1, and C2. The files of the same category solve the same Google Code Jam contest problem [10]. The A1 category solves a simple problem, and the difficulty increases till reaching C2. The total number of files in SOCO-TEST is 12K files. A10041 and A22867 files were corrupted and removed. The detailed sizes of SOCO-TEST are shown in Table 5. It is worth noting that the total number of pairs (i.e., 17.9M) includes only the pairs of the same category.

The ground truth file of SOCO-TRAIN contains 84 lines; one line for each plagiarized pair. Each line is space-separated (e.g., the line “003.java 004.java” means that 003.java and 004.java are plagiarized). The ground truth file of SOCO-TEST contains 221 lines. The plagiarized pairs of all SOCO-TEST categories exist in one ground truth file, but there are no lines containing pairs of different categories (e.g., “A10000 A20000”).

Performance measures

The performance measures used are Precession, Recall, and F-Measure of detecting the plagiarized pairs (the positive class). Precision measures the percentage of the correctly predicted positive with respect to all positive predictions. Recall measures the percentage of the correctly predicted positive with respect to all the positive examples in the dataset. F-Measure combines both precision and recall in one measure. Equations 4, 5, and 6 show how to calculate them. We used the same evaluation script (Python script) that was used in SOCO-14’s task [22] to evaluate its participating systems.

$$Precision = \frac{Count(CorrectlyPredictedAsPlagiarized)}{Count(AllPredictedAsPlagiarized)} \tag{4}$$

$$Recall = \frac{Count(CorrectlyPredictedAsPlagiarized)}{Count(AllPlagiarizedInGroundTruth)} \tag{5}$$

$$F\_Measure = \frac{2 * Precision * Recall}{Precision + Recall} \tag{6}$$

Experimental settings

The proposed approach of Ganguly et al. [8] is time efficient and scalable because it consists of two stages: (i) the first stage (i.e., the information retrieval or IR stage) is a

Table 5 The number of files and pairs of SOCO-TEST after removing corrupted files

Category	Number of files	Number of pairs	Number of plagiarized pairs
A1	3240	5,247,180	54
A2	3092	4,778,686	46
B1	3268	5,338,278	73
B2	2266	2,566,245	34
C1	124	7626	0
C2	88	3828	14
SUM	12,078	17,941,843	221

Table 6 The used parameters' values of the "init.properties" file [23]

Parameter	Description	Value
field	True if the per-field representation is used	True
toptermquery	True if top "num_q_terms" terms (not all terms) are considered for each field after sorting terms by TFIDF score	True
num_q_terms	The number of top terms for each field if "toptermquery" is true	20
lambda	The weight (from 0 to 1) of TF with respect to IDF	0.4
minShingleSize	The minimum size of the word ngrams of terms (Note: the unigrams are included by default)	2
maxShingleSize	The maximum size of the word ngrams of terms	3
num_wanted	For each query document, the top "num_wanted" hit documents (that are sorted by relevance score) are included in the output file.	20

Table 7 The used parameters' values of our feature extraction code

NGram	NPartition	Low	High
3	4	0.3	0.7

time-efficient unsupervised stage that is used to prune the number of file pairs to compare, and (ii) the second stage (i.e., the classification stage) is a binary classification stage that decides whether each file pair outputted from the first stage is plagiarized. We used the same IR Stage and focused on modifying the classification stage by using different features sets and evaluating their effectiveness.

The code of the IR Stage is publicly available [23] by Ganguly et al. [8], and we re-implemented the code of the classification stage from scratch. Regarding the IR stage, Lucene (Version 4.6.0) [24] is used for indexing, and Java Parser (Version 1.0.10) [25] is used for parsing Java source codes. Regarding the classification stage, the Weka tool (Version 3.8) [26] is used for classification, and random forest is used with ten iterations.

The classifier is trained using all file pairs of the SOCO-TRAIN dataset (33.4K instances). SOCO-TRAIN is only used in training the classifier, and any other remaining work is made on the files of SOCO-TEST. In the IR stage, we first indexed all the files of the SOCO-TEST dataset (12K files), then used each document of it again to query the index individually and output the hit documents. After running the IR stage, all the lines of the output file which contains the potential plagiarized pairs are entered into the trained classifier of the second stage to classify it.

Regarding the IR stage [8], each document (source file) is represented by a set of fields (e.g., class names, method calls, values of string literals). Each field consists of multiple terms (e.g., the class names' field of a file containing four classes could have the terms: C1, C2, C3, and C4). Then, this representation is stored for each document in a per-field index. Querying the per-field index compares the values of the same field together, so it does not compare, for example, class names with string literals. The available code of the IR Stage uses a configuration file called "init.properties" [23]. The final parameter values we used for this file and a short description of them are presented in Table 6.

Regarding the classification stage, the parameters used in our feature extraction code are shown in Table 7. NGram is the number of the character n-grams used in the feature sets. NPartition is the number of partitions of the structural histogram features. Low and high are the low and high similarity thresholds of histogram extreme ranges.

Experimental results

Two baselines are used to compare our results with:

- Baseline I (i.e., JPlag tool [5]): which converts each file into a token string then uses the Greedy String Tiling (GST) algorithm to get the similarity between token string pairs. Most of the related work uses it for comparison. It is also worth noting that the JPlag's results reported in this paper are the results that were reported in SOCO-14 [22].
- Baseline II (i.e., Ganguly et al. [8]): which is summarized in Section 3.3. It is chosen as a baseline for its scalability and good results. It is also worth noting that the results of baseline II written in our paper are the results we got using the parameters' values mentioned in Table 6 for the IR stage and using our implementation of the classification stage.

We performed seven experiments. The feature sets used in each experiment with the total number of features are shown in Table 8. It is worth noting that baseline II includes only the original features of Ganguly et al. [8] summarized in Section 2.4; however, all our experiments (except ModOriginal and ModOriginal-MainRmv experiments) include the original features in addition to our proposed feature sets to evaluate how the added feature sets improve the results of baseline II.

The results of applying the entire pipeline (i.e., the IR stage and the classification stage) are shown in Table 9. It is worth noting that all our experiments have results better than baseline I. In addition, the recall of all our experiments is high (more than 96% in all experiments except the StructCounts and Hist4+StructCounts experiments whose recall is 90%).

Regarding the Hist4 experiment, adding the structural histogram features to the original features increased the precision of baseline II (that includes only the original features) by + 2.6% with nearly the same recall, and this improved the F-measure by + 1.8%. This means that representing similarity matrices by the histogram features is

Table 8 The feature sets used in each performed experiment with the total number of features

Experiment	Description of features	Number of features
Hist4	Original features (8) + structural histogram features (8)	16
PerClass	Original features (8) + lexical per-class features (3)	11
Hist4+PerClass	Original features (8) + structural histogram features (8) + lexical per-class features (3)	19
StructCounts	Original features (8) + structural counting features (12)	20
Hist4+StructCounts	Original features (8) + structural histogram features (8) + structural counting features (12)	28
ModOriginal	Modified original features with main included (8)	8
ModOriginal-MainRmv	Modified original features with main excluded (8)	8

Table 9 The precision, recall, and F-measure results of the performed experiments

Experiment	Precision	Recall	F-measure
Baseline I: JPlag [5]	54.2%	29.3%	38.0%
Baseline II: Ganguly et al. [8]	61.2%	97.7%	75.3%
Hist4	63.8%	97.3%	77.1%
PerClass	54.6%	99.1%	70.4%
Hist4+PerClass	66.3%	98.6%	79.3%
StructCounts	59.2%	90.5%	71.6%
Hist4+StructCounts	62.8%	90.0%	74.0%
ModOriginal	61.2%	96.4%	74.9%
ModOriginal-MainRmv	57.1%	96.4%	71.7%

promising and reflects the similarities better than only the minimum, maximum, and average.

Regarding the PerClass experiment, adding the lexical per-class features increased the recall by +1.4% compared to baseline II to reach 99.1%; however, the precision decreased significantly by − 6.6% which leads to less F-measure than baseline II. This improvement is because of incorporating the lexical similarity of each class pair in the two programs and considering only the best candidates of the similarity matrix.

In the Hist4+PerClass experiment, both the structural histogram features and the lexical per-class features were added to the original features. The Hist4+PerClass experiment improved both the precision and recall not only of baseline II but also Hist4; hence, the Hist4+PerClass experiment has the best results we got so far. It increased the F-measure of baseline II [8] by + 4%. It also increased the F-measure of Hist4 by + 2.2% (by increasing the recall by + 1.3% and the precision by + 2.5%).

Regarding the StructCounts experiment, adding the structural counting features to the original features was unexpectedly not beneficial (StructCounts was less in precision by − 2% and recall by − 7.2 which leads to − 3.7% difference in F-measure compared to baseline II).

In the Hist4+StructCounts experiment, both the structural histogram features and the structural counting features were added to the original features. The Hist4+StructCounts experiment improves the results of StructCounts (precision and F-measure are increased by + 3.6% and + 2.4% respectively) but still less than baseline II.

We may be surprised that adding the structural counting features yields worse results, although they reflect the code structure. One probable reason for that is, as it was mentioned before in Section 3.1, that the files of each category in SOCO-TEST solve the same problem, and this may produce similar structural counts (i.e., a similar number of classes, functions, etc.). That could be one of the reasons why adding the structural counting features for SOCO-TEST dataset did not improve the results (i.e., they are not distinguishing features for this particular dataset).

Regarding the ModOriginal experiment, the modified original features (which more reasonably modify the original structural features) were used instead of the original features. The F-measure of the modified original features (i.e., the ModOriginal experiment) is nearly the same as the original features (i.e., baseline II), but the proposed modifications are more reasonable. It is worth noting that the ModOriginal experiment includes the main functions in the built similarity matrices.

On the other hand, the ModOriginal-MainRmv experiment removed the main functions from the similarity matrices of function signatures because the Java main functions are identical in the function signature. The results unexpectedly show that removing the main functions yields less precision and F-measure than not removing them (i.e., − 4.1% and − 3.2% respectively); however, removing the main functions signatures still make more sense.

In summary, the feature set that yields the best results reached so far is the features of the Hist4+PerClass experiment, which include 19 features:

- The eight original features [8]
- The eight structural histogram features of 4 partitions
- The three lexical per-class features

The Hist4+PerClass experiment increases the F-measure of baseline II by +4%. Consequently, Hist4+PerClass turns out to be our experiment of choice.

Time analysis

The time analysis for the classification stage of Hist4+PerClass experiment compared to the time of baseline II [8] (the best of our two baselines in results) is shown in Table 10. We performed three runs for each experiment and calculated the average elapsed time. The used processor is Intel(R) Core(TM) i7-6500U CPU @ 2.50GHz 2.60 GHz, and the RAM is 8.00 GB.

The classification stage of Hist4+PerClass takes 23 min to classify 70K pairs (i.e., 0.02 s per pair). Hist4+PerClass takes elapsed time more than of baseline II because Hist4+PerClass extracts 11 features more than baseline II, and this increases the F-measure by 4%. In general, the elapsed time in the proposed Hist4+PerClass experiment is efficient because it does not involve any complex representation for code structure.

Dataset analysis

We performed three analyses on the used dataset (SOCO-14 [22]). Regarding the first analysis, when we extracted the false positive pairs of Hist4+PerClass experiment (111 pairs) to analyze why our system misclassifies them, we discovered that 74 out of the

Table 10 The elapsed time of the classification stage of our best-proposed experiment (Hist4+PerClass) compared to baseline II

Experiment	Number of features	Number of pairs	Elapsed time of classification
Baseline II [8]	8	69,987	744.67 s = 12.41 min
Hist4+PerClass	19	69,987	1395.33 s = 23.26 min

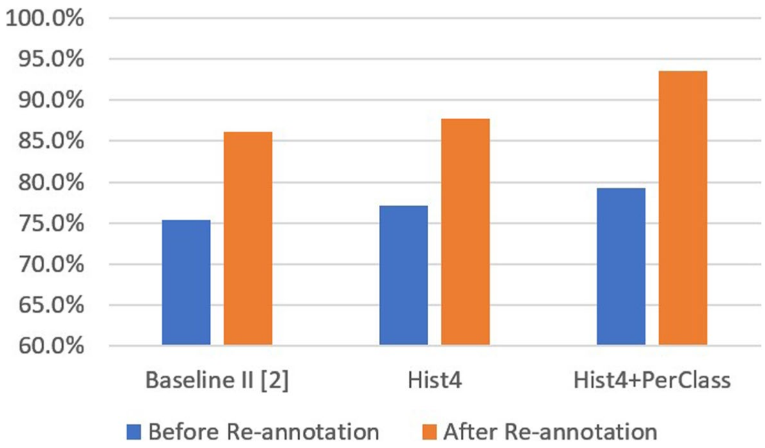


Fig. 4 The F-measure before and after the dataset re-annotation

Table 11 The results before and after the dataset re-annotation

Experiment	**Precision**		**Recall**		**F-measure**	
	Before	**After**	**Before**	**After**	**Before**	**After**
Baseline II [8]	61.2%	79.0%	97.7%	94.6%	75.3%	86.1%
Hist4	63.8%	82.2%	97.3%	93.9%	77.1%	87.7%
Hist4+PerClass	66.3%	88.8%	98.6%	99.0%	79.3%	93.6%

111 false positive pairs are true positives (correctly predicted by our system but wrongly labeled in SOCO-14 [22]). We could not perform this re-annotation on all the dataset pairs because the dataset is very large and contains 12K files. We make the list of the re-annotated pairs (the 111 pairs) publicly available as a justification of no bias in re-annotation [27].

To make sure that we are not favoring our approach in this re-annotation, we calculated the F-measure of baseline II in addition to our best two experiments before and after the re-annotated dataset as shown in Fig. 4. The F-measure of baseline II itself is increased by 10.8% after using the re-annotated dataset. In addition, Hist4+PerClass improves the F-measure of baseline II by 7.5% using the re-annotated dataset, and the F-measure of Hist4+PerClass reaches 93.6%. The detailed results including the precision and the recall are shown in Table 11.

It is worth mentioning again that while high similarity does not necessarily correlate with plagiarism, the main target of the source code plagiarism tools in general is to filter all pairs (numerous pairs, quadratic) and extract only the highly similar pairs (either similar lexically, semantically, etc.), and still a manual evaluation of the extracted pairs and an investigation with the involved students are needed to report plagiarism. That is why re-annotating the dataset pairs for correcting the ground truth labels of the similar pairs is essential.

The second performed dataset analysis is about duplicate files. As stated in SOCO-14 [22], each category of the test set solves a problem different from the other categories. This intuitively means that it does not fit to find duplicate files between different categories. However, when we searched for the verbatim-copy files that belong to different categories, we found that 4600 pairs are duplicates while belonging to different categories. This means that there are 4600 file pairs (that belong to different categories) that are considered negative in the dataset if we combine all categories, but these pairs are positive (e.g., A10000 with A20000, B10197 with B20148). Hence, as a conclusion, combining all categories and considering the pairs of different categories as negative is not accurate.

The third and final analysis we performed on the SOCO-14 dataset [22] is on the number of classes in the test set. We calculated the number of files for each number of classes as shown in Table 12. We found that more than 90% of the files of the test set contain at most one class. Consequently, we can conclude that if the detection approach strongly depends on the number of classes, the SOCO-14 dataset is not the best dataset to show the approach effectiveness because its majority of files contain only one class.

Conclusions

Source code plagiarism is considered a major ethical problem that affects the learning process negatively. This urges the need to detect it and apply a strict penalty to the involved students. This paper introduces four feature sets for detecting source code plagiarism: (i) structural histogram features which propose a more indicative way to summarize similarity matrices using a histogram of similarity values, (ii) lexical per-class features which represent the lexical similarity between the class pairs of the two files by building a lexical per-class similarity matrix and summarizing it to a set of descriptive features, (iii) structural counting features which extract twelve counting features representing the code structure (e.g., number of classes, functions), and (iv) modified original features which modify the feature set of Ganguly et al. [8] by using the candidates and the extreme ranges.

The feature sets of the best results are the structural histogram features and the lexical per-class features combined (i.e., a 4% F-measure increase compared to Ganguly et al. [8]), and their time for classification is efficient (23 min for classifying 70K pairs). Moreover, we provided a partial dataset re-annotation that yields an F-measure 7.5% more than Ganguly et al. [8], and an F-measure of 93.6% is achieved. The modified original features are shown to be more reasonable while maintaining nearly the same F-measure. After dataset analysis, we found that there are duplicate files between its different categories, and SOCO-14 [22] is not recommended if the approach depends on numerous classes. Our future work is to use different structure-based feature sets such as abstract syntax trees, dependency graphs, etc. In addition, we will perform further experiments on the information retrieval stage of Ganguly et al. [8] (e.g., adding more fields, trying different parameter values).

Table 12 The number of classes' analysis on SOCO-TEST [22]

Number of classes	Number of files of SOCO-TEST
Any number of classes (all files)	12,078
Less than or equal to one	10,970
Equal two	854
Greater than or equal three	254

Abbreviations

SOCO-14	SOurce COde re-use dataset
SOCO-TRAIN	The TRAINing set of SOurce COde re-use dataset
SOCO-TEST	The TEST set of SOurce COde re-use dataset
IR	Information retrieval

Supplementary Information

The online version contains supplementary material available at https://doi.org/10.1186/s44147-022-00155-8.

Additional file 1. It is our partial re-annotation of 111 pairs of SOCO-14 dataset. URL of the same file [27]: https://cutt.ly/NQO6ctK.

Acknowledgements

Not applicable.

Authors' contributions

EH is the corresponding author that made all the technical work (the implementation) and the major contributor in writing the manuscript. MH, AA, and MF are the supervisors who discussed and suggested different ideas with EH. All authors read and approved the final manuscript.

Funding

This study had no funding from any resource.

Availability of data and materials

It is the URL of our partial re-annotation of 111 program pairs of SOCO-14 dataset. Re-annotation URL [27]: https://cutt.ly/NQO6ctK

Declarations

Competing interests

No explicit competing interests other than that our affiliation is Faculty of Engineering at Cairo University.

Received: 5 February 2022 Accepted: 18 October 2022
Published online: 08 November 2022

References

1. Baer N, Zeidman R (2012) Measuring whitespace pattern sequences as an indication of plagiarism. Journal of Software Engineering and Applications 5(4):249–254
2. Shay I, Baer N, Zeidman R (2010) Measuring whitespace patterns as an indication of plagiarism. In Proceedings of Annual ADFSL Conference on Digital Forensics, Security and Law, St. Paul, pp. 63–72
3. Faidhi JA, Robinson SK (1987) An empirical approach for detecting program similarity and plagiarism within a university programming environment. Computers and Education 11(1):11–19
4. Karnalim O (2016) Detecting source code plagiarism on introductory programming course assignments using a bytecode approach. In Proceedings of 2016 International Conference on Information & Communication Technology and Systems (ICTS), IEEE, pp 63–68
5. Prechelt L, Malpohl G, Philippsen M et al (2002) Finding plagiarisms among a set of programs with jplag. Journal of Universal Computer Science 8(11):1016–1038
6. Liu C, Chen C, Han J, Yu PS (2006) Gplag: detection of software plagiarism by program dependence graph analysis. In Proceedings of the 12th ACM SIGKDD international conference on Knowledge discovery and data mining (KDD'06), Philadelphia, USA, pp. 872–881
7. Sulistiani L, Karnalim O (2019) Es-plag: Efficient and sensitive source code plagiarism detection tool for academic environment. Computer Applications in Engineering Education 27(1):166–182
8. Ganguly D, Jones GJ, Ramirez-De-La-Cruz A, Ramirez-De-La-Rosa G, Villatoro-Tello E (2018) Retrieving and classifying instances of source code plagiarism. Information Retrieval Journal 21(1):1–23
9. Ullah F, Wang J, Farhan M, Habib M, Khalid S (2021) Software plagiarism detection in multiprogramming languages using machine learning approach. Concurrency andComputation: Practice and Experience 33(4):e5000
10. Ramirez-de-la Cruz A, Ramirez-de-la Rosa G, Sanchez-Sanchez C, Jimenez-Salazar H (2014) On the importance of lexicon, structure and style for identifying source code plagiarism. In Proceedings of the Forum for Information Retrieval Evaluation (FIRE'14). ACM Press, New York, pp. 31–38
11. Moussiades L, Vakali A (2005) Pdetect: a clustering approach for detecting plagiarism in source code datasets. The Computer J 48(6):651–661
12. Karnalim O (2021) Source code plagiarism detection with low-level structural representation and information retrieval. International Journal of Computers and Applications 43(6):566–576
13. Jadalla A, Elnagar A (2008) Pde4java: Plagiarism detection engine for java source code: a clustering approach. International Journal of Business Intelligence and Data Mining 3(2):121–135
14. Rabbani FS, Karnalim O (2017) Detecting source code plagiarism on. net programming languages using low-level representation and adaptive local alignment. Journal of Information and Organizational Sciences 41(1):105–123

15. Fu D, Xu Y, Yu H, Yang B (2017) Wastk: a weighted abstract syntax tree kernel method for source code plagiarism detection. Scientific Programming, pp. 1–8
16. Duracik M, Hrkut P, Krsak E, Toth S (2020) Abstract syntax tree based source code antiplagiarism system for large projects set. IEEE Access, 8:175347–175359
17. Song HJ, Park SB, Park SY (2015) Computation of program source code similarity by composition of parse tree and call graph. Mathematical Problems in Engineering, pp. 1–12
18. Chae DK, Ha J, Kim SW, Kang B, Im EG (2013) Software plagiarism detection: a graph-based approach. In Proceedings of the 22nd ACM international conference on Information and Knowledge Management, ACM, pp. 1577–1580
19. Ullmann JR (1976) An algorithm for subgraph isomorphism. Journal of the ACM (JACM) 23(1):31–42
20. Ullmann JR (2011) Bit-vector algorithms for binary constraint satisfaction and subgraph isomorphism. Journal of Experimental Algorithmics (JEA) 15:1–1
21. Collins M, Duffy N (2002) Convolution kernels for natural language. Advances in Neural Information Processing Systems. MIT Press, Cambridge, 14:625–632
22. Flores E, Rosso P, Moreno L, Villatoro-Tello E (2014) On the detection of source code re-use. In Proceedings of the Forum for Information Retrieval Evaluation, Association for Computing Machinery, New York, NY, USA, FIRE '14, pp. 21–30
23. Ganguly D (2014) Yasocs. https://github.com/gdebasis/YASOCS. Accessed 5 Feb 2022
24. Lucene version 4.6.0. https://archive.apache.org/dist/lucene/solr/4.6.0/. Accessed 28 Sep 2022
25. Java parser version 1.0.10. https://jar-download.com/artifacts/com.google.code.javaparser/javaparser/1.0.10/source-code. Accessed 28 Sep 2022
26. Weka tool (version 3.8). https://weka.informer.com/3.8/. Accessed 28 Sep 2022
27. Hosam E (2020) Soco re-annotation. https://cutt.ly/NQO6ctK. Accessed 5 Feb 2022

Publisher's Note

Springer Nature remains neutral with regard to jurisdictional claims in published maps and institutional affiliations.