On the Importance of Lexicon, Structure and Style for Identifying Source Code Plagiarism

Aarón Ramírez-de-la-Cruz
Departamento de Tecnologías de la Información
Universidad Autónoma Metropolitana
Unidad Cuajimalpa, México D. F.

Gabriela Ramírez-de-la-Rosa*
Departamento de Tecnologías de la Información
Universidad Autónoma Metropolitana
Unidad Cuajimalpa, México D. F.

Christian Sánchez-Sánchez
Departamento de Tecnologías de la Información
Universidad Autónoma Metropolitana
Unidad Cuajimalpa, México D. F.

Héctor Jiménez-Salazar
Departamento de Tecnologías de la Información
Universidad Autónoma Metropolitana
Unidad Cuajimalpa, México D. F.

ABSTRACT

Source code plagiarism can be identified by analyzing similarities of several and diverse aspects of a pair of source code. In this paper we present three types of similarity features that account for three aspects of source code documents, particularly: *i)* lexical, *ii)* structural, and *iii)* stylistics. From the lexical view, we used a character 3-gram model without considering reserved words for the programming language in revision. For the structural view, we proposed two similarity metrics that take into account the *function's signatures* within a source code, namely the data types and the identifier's names of the function's signature. The third view consists on accounting for several stylistics' features, such as the number of white spaces, lines of code, upper letters, etc. Accordingly, we proposed 8 similarity features to represent pairs of source code in order to, under a supervised approach, identify plagiarized pairs of source codes. We use a set of more than 32000 source code documents from Java and C to perform our experiments. The results show the pertinence of our set of features to identify plagiarism for source code documents that satisfy particular conditions, such as, source code that solve difficult problems.

CCS Concepts

•Information systems → Content analysis and feature selection; Near-duplicate and plagiarism detection; •Applied computing → *Document analysis;*

*Corresponding author. E-mail address: `gramirez@correo.cua.uam.mx`

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.

FIRE '14, December 05-07, 2014, Bangalore, India
© 2015 ACM. ISBN 978-1-4503-3755-7/15/08...$15.00
DOI: http://dx.doi.org/10.1145/2824864.2824879

Keywords

Lexical, structural and stylistic features; Document representation; Plagiarism detection; Source code plagiarism

1. INTRODUCTION

Plagiarism detection in source code documents is a topic of growing interest for both the software industry and the academia. In the software industry, systems that automatically detect plagiarism cases help in the prevention of intellectual property infringements. In the academia, due to the enormous among of forums, blog, repositories, etc., students can easily download almost any source code required for their computing assignments; thus, automatic systems are helpful to avoid this increasing dishonest practice among students.

The first formally stated definition of the problem was proposed in 1987 by Faidhi and Robinson [6]. According to their proposal, the most frequent modifications performed by a source code's plagiarist are categorized in a set of seven sub-types or levels according to the severity of plagiarism, shown as follows:

Level 0 : Exact copy of the original source.
Level 1 : Modification in comments.
Level 2 : Changing identifier names.
Level 3 : Changing variable position (using different indentation level or switching the order of variables in a symmetric function).
Level 4 : Procedural combination.
Level 5 : Changing program statements.
Level 6 : Changing logical control.

However, it is important to notice that programmers who plagiarize source code usually apply not one, but several obfuscation techniques to avoid detection. Therefore, even though there are several proposed techniques to detect different types of source code plagiarism, it is very difficult for a single automatic system to detect all of these different types of obfuscation practices.

Particularly, the main research tackling a source code plagiarism is mostly centered on the analysis of the structure of the source code documents mainly focusing on the study of syntactic trees and tracking function's calls, to mention a few. These type of methods have been proved useful in this task; however, its complexity and its computation is usually

expensive.

Therefore, we proposed a method that, at the same time, takes into account different aspects of a source code and also accounts for the structure in a more inexpensive way than previously proposed methods. Accordingly, we propose different representations for a source code, namely: character n-grams, data types, identifiers' names, and some stylistics features. Our intuitive idea is that by means of the capture of several aspects of a source code, it will be possible to identify some of the most common practices performed by programmers when they plagiarize source code.

Thus, in this paper we posed the following research questions:

Q1: Which aspect of a source code is more important in order to identify plagiarism?

Q2: It is possible to capture some of the structure of source code files by looking only at function's signatures?

The rest of this paper is organized as follows. Section 2 presents some related work concerning to algorithms for identifying source code plagiarism that use natural language processing techniques. Section 3 describes our proposed method; based on getting the 8 similarity measures extracted from lexical, structural and stylistic aspects of source codes. Then, Section 4 describes the experimental evaluation, particularly the data collection, evaluation measures, and our initial analysis done over our proposed features. Obtained results and a deeper analysis are described in Section 5. Finally, Section 6 depicts our conclusions and some future work ideas.

2. RELATED WORK

Lately, developed automated systems to identify source code re-use are applying natural language processing (NLP) techniques that are been adapted to this specific context. One example of those systems is one that takes into account a remanence trace left after a copy of source code, such as, white space patterns [2]. The intuitive idea behind this approach indicates that a plagiarist camouflages almost every thing when copying a source code but the white spaces. Accordingly, it compute similarities between source codes taking into account the use of letters (all represented as `X`) and white spaces (represented as `S`). As another example of automatic systems that employ NLP techniques, are those based on word n-grams [1, 12]. These works consider several features of source code, such as, identifiers, number of lines, number of hapax, etc. Their obtained results were very promising.

Some other works employed transformations techniques based on LSA, for example the work presented in [5]. The authors of this work focused on three components: preprocessing (keeping or removing comments, keywords or program skeleton), weighting (combining diverse local or global weights) and the dimensionality of LSA. The experiments were based on information retrieval: given a query as a source code aimed to obtain the most similar source codes.

A slightly different approach was presented in [7], they used syntax trees to represent each source code. Their plagiarism detection technique is based on abstract syntax tree algorithm (AST-CC algorithm). Each document is represented in terms of nodes that are likely to being plagiarized into a hash table. They argue that the AST-CC algorithm can effectively detect several plagiarism cases such as, changing the variable name, reordering the sequence of the expression evaluation and changing part of the code statements.

As can be observed, a common characteristic of previous works is that they attempt to capture several aspects from source codes into one single/mixed representation (*i.e.,* a single view) in order to detect source code re-used. Contrary to these previous methods, our hypothesis states that each aspect (*i.e.,* either structural or superficial elements) provides its own important information, that if mixed with other aspects their importance could be diluted, thus better results would be obtained when each aspect is considered independently when representing source codes.

3. SIMILARITY MEASURES

Our proposed method depicts a supervised classification approach that represents a pair of source codes D_1 and D_2 by means of eight distinct features, namely:

$$\langle f_{lexical}, f^1_{structural}, \ldots, f^6_{structural}, f_{stylistic} \rangle \quad (1)$$

Proposed features aim at measuring several aspects of source code that help to capture some of the most common practices among plagiarist when camouflaging plagiarized sections. As can be seen in Expression 1, we divided these features into three categories according to the information that they are able to capture, namely: *lexical*, *structural* and *stylistic* features.

3.1 Lexical Feature

Given that the use of words n-grams [4] and characters n-grams [15], as representation of documents, have showed good performances on identifying plagiarism in both, text documents and source code files [9, 15], we use this representation to measure the lexical aspect of source codes. To some extent this lexical feature has been associated with content information.

As in [9] we represent a source code document as a bag of character 3-grams, where every white spaces and line-breaks were deleted and every letter was lower-cased. Additionally, we eliminated all the reserved words of the two programming languages (Java and C) in our data collection to avoid misleading the classification due to the large amount of keywords in the documents (more information about the data collection is given in Section 4.1).

In order to measure the similarity between a pair of source code documents D_1 and D_2 each code is represented as a vector according to the vector space model [3], where the dimension of these vector is given by the vocabulary of 3-grams in both documents. Then, the similarity between the two source code documents are computed by the cosine similarity (Equation 2).

$$f_{lexical}(D_1, D_2) = \frac{\overrightarrow{D_1} \cdot \overrightarrow{D_2}}{\|\overrightarrow{D_1}\| \|\overrightarrow{D_2}\|} \quad (2)$$

The obtained similarity value from Equation 2 is considered as one feature in our proposed representation (Expression 1), particularly represents the lexical feature ($f_{lexical}$).

3.2 Structural Features

To take into account some structural characteristics of a source code, we decided to measure the similarities of the function's signatures within the documents. To some extent

the signatures give information about the organization and structure of source codes, at low cost.

The proposed structural features (six total) consist of two forms of representation; both of them based on the function's signatures definition within a source code. Accordingly, three of such similarity measures are based solely on the data types, and the other three are based merely on the identifier's name from the signature's function. The general procedure to compute the structural features is described by Algorithm 1.

Algorithm 1. Given two source code documents, D_1 and D_2, compute the structural features as:

1. Extract all signature's function of both D_1 and D_2.
2. Select one representation from the signature's function: *data types* or *identifiers names*, and extract the corresponding features to compute the similarity.
   (a) Compute the similarity between every pair of function within the two documents for data types or for identifiers names.
3. The computed similarities form a similarity matrix where each cell (i, j) represent the similarity between the i-th function and the j-th function. The columns of the matrix represent all functions of document D_1 and the rows all functions of document D_2.
4. From the similarity matrix compute a global similarity for the pair of source code documents D_1 and D_2.

Steps 1 and 3 are straightforward. In **step 2** we use two different type of representations attempting to compare elements that may be considered as part of the structure of the program. The first type of representation takes only the data type within the signature's function (*e.g.*, `char, int, float, String`, etc.). This representation aims to capture when the programmer tries to obfuscate the plagiarism by changing only the parameter's names and the function name itself. The second representation accounts for all the identifiers names within the signatures, *i.e.*, the function name and the parameter's names. The idea behind this second representation can be considered complementary to the former and aims to capture a more complex plagiarism technique, according to [6].

Similarity between Data Types.

Each function is represented as a list of data types. For example, the following function's signature "`int sum(int numX, int numY)`" will be translated into "`int (int, int)`". Our proposed representation also accounts for the frequency of each data type. Then, we need to compare two elements independently to calculate the similarity between two functions: *i)* the return data type, and *ii)* arguments' data types.

Accordingly, in order to compute the return data type similarity we proceed as follows. Given two functions, m^1 and m^2, belonging to source code documents D_1 and D_2, respectively; the similarity of their return data type (sim_r) is 1 if both functions have the same return data type, and 0 otherwise.

Next, to determine the similarity of their arguments' data types we propose a more elaborated strategy. First, we determine the data-types for each function with their frequencies, hence each function is represented as a vector where its components are frequencies of data-types. Then, we compute a similarity between two functions' vectors $\mathbf{m^1}$ and $\mathbf{m^2}$ as defined in Equation 3, where n indicates the number of different data types in both functions, *i.e.,* the vocabulary of data types.

$$sim_a(\mathbf{m^1}, \mathbf{m^2}) = \frac{\sum_{i=1}^{n} \min(\mathbf{m^1}_i, \mathbf{m^2}_i)}{\sum_{i=1}^{n} \max(\mathbf{m^1}_i, \mathbf{m^2}_i)} \quad (3)$$

Heretofore, we have computed the return data type similarity (sim_r), as well as the arguments' data type similarity (sim_a) between the two initial functions, *i.e.*, m^1 and m^2; now it is possible to determine a single value (sim_1) for the data type similarity between these two functions by means of a linear combination as defined in Equation 4, where σ is a scalar that weights the importance of each term and it satisfies that $0 \leq \sigma \leq 1$. For our performed experiments, we established $\sigma = 0.5$ so both parts are considered equally important. Considering $\rho = 1 - \sigma$, then sim_1 is:

$$sim_1(m^1, m^2) = \sigma * sim_r(m^1, m^2) + \rho * sim_a(\mathbf{m^1}, \mathbf{m^2}) \quad (4)$$

Similarity between Identifiers.

Each function is represented as a concatenation of the function's name and the names of all its arguments. We eliminate spaces and every letter is lower-cased. For example, the function '`int sum(int numX, int numY)`' is represented as the string '`sumnumxnumy`'. The next step consists on computing the corresponding character 3-grams representation.

Consequently, given the functions m^1 and m^2, belonging to source code documents D_1 and D_2, respectively; and their corresponding bags of character 3-grams, $\mathtt{m}^1$ and $\mathtt{m}^2$, we compute their similarity using the Jaccard coefficient as follows:

$$sim_2(m^1, m^2) = \frac{\mathtt{m}^1 \cap \mathtt{m}^2}{\mathtt{m}^1 \cup \mathtt{m}^2} \quad (5)$$

Step 4 in Algorithm 1 involves the calculation of a global similarity for a given pair of source code documents D_1 and D_2. This requires the construction of a similarity matrix where each cell is the similarity of every pair of functions in source code documents D_1 and D_2. Therefore, two matrix are computed: $\mathbf{M}^{type}_{D_1,D_2}$ and $\mathbf{M}^{names}_{D_1,D2}$.

Lastly, values of similarity between two codes are defined as shown in Equation 6 and Equation 7, for data types and identifiers' names, respectively; where $f(x)$ represents either the maximum value contained in the matrix, the minimum value contained in the matrix, or the average value among all values from the matrix.

$$sim_{DataTypes}(D_1, D_2) = f(\mathbf{M}^{type}_{D_1,D_2}) \quad (6)$$

$$sim_{Names}(D_1, D_2) = f(\mathbf{M}^{names}_{D_1,D_2}) \quad (7)$$

Notice that selecting either the maximum or the minimum value from $\mathbf{M}^{type}_{D_1,D_2}$ implies that the similarity between D_1 and D_2 is been determined just by considering the similarity of one pair of functions (the most similar or the less similar respectively), whilst the average value considers the similarity between all the possible pairs of functions contained in D_1 and D_2. Our intuition is that all

of these values might convey important information, hence we preserve these values as three elements on Expression 1: $\{f^{T_{min}}_{structural}, f^{T_{max}}_{structural}, f^{T_{average}}_{structural}\}$. Similarly, from $\mathbf{M}^{names}_{D1,D2}$ we take the other three features of the Expression 1, namely, $\{f^{N_{min}}_{structural}, f^{N_{max}}_{structural}, f^{N_{average}}_{structural}\}$.

3.3 Stylistic Feature

Analogous to natural language text documents which inherently contain author specific writing style characteristics, we hypothesize that the source codes also carry programmer specific stylistic features.

Accordingly, we choose a set of 11 stylistic features, namely: the number of lines of code, the number of white spaces, the number of tabulations, the number of empty lines, the number of defined functions, average word length, the number of upper case letters, the number of lower case letters, the number of under scores, vocabulary size, and the lexical richness (*i.e.,* the total number of tokens over the vocabulary size).

To determine the stylistic similarity we use a vector representation for all these features and apply the cosine similarity (Equation 2) to determine the stylistic feature ($f_{stylistic}$) on Expression 1.

4. EXPERIMENTAL EVALUATION

4.1 The Data Collection

The data collection was provided by SoCO competitive evaluation campaign for systems that automatically detect the source code re-use phenomenon [10]. SoCO, Detection of SOurce COde Re-use, is a shared task that focuses on monolingual source code re-use detection. Participant systems were provided with a set of source codes in C and Java programming languages. The task consists on retrieving the source code pairs that have been re-used at a document level.

The data set provided for the shared task is divided into two sets: training and test. On the one hand, the training set (338 source codes in total) was manually labeled where the relevance assessments represent cases of re-use in both directions, *i.e.*, the direction of the re-use is not being detected.

On the other hand, in order to evaluate the systems on the test set (31,975 source codes in total), the organizers applied the standard *pooling* approach [14]. The union of the set of plagiarized pairs reported by the participating systems was used to construct the pool[1]. More information about the collection data set is presented in Table 1.

Table 1: Data collection provided by SOCO competition

	Train subset	
	# of source codes	Re-use code source pairs
C	79	26
Java	259	84
	Test subset	
	# of source codes	Re-use code source pairs
C/C++	19,895	322
Java	12,080	222

It is worth to mention that the test subset is divided into 6 scenarios, namely A1, A2, B1, B2, C1, and C2. Where a scenario means that all source code from it solve the same problem. The complexity of the problems increases from scenario A to C and from 1 to 2 (for a more detailed description see [10, 8]).

Considering, in Table 2 we present information of the source code documents in the test set by scenario. We include information about the total number of source codes per scenario, the average number of lines, the average number of functions within source codes, as well as, the percentage of the total source code documents containing functions (we exclude the '`main`' function in this count).

Table 2: Average of number of functions in the data collection for the test set per scenario

	C/C++ language					
	A1	A2	B1	B2	C1	C2
# of source codes	5408	5195	4939	3873	335	145
avg. # of lines	63.7	68.1	70.0	80.9	180.4	255.2
% docs w/functions	36.3	35.6	61.9	47.8	77.3	83.4
avg. # of functions	2.99	2.98	2.66	2.74	4.86	4.70
	Java language					
	A1	A2	B1	B2	C1	C2
# of source codes	3239	3089	3266	2266	124	88
avg. # of lines	99.3	107.9	90.9	102.5	227.1	361.0
% docs w/functions	98.4	98.3	98.4	98.3	94.3	95.4
avg. # of functions	3.32	3.23	3.52	3.76	9.51	9.48

This Table 2 shows that in general the average number of function in the source codes is low; but it is bigger for scenarios that solve more difficult problems (*i.e.,* C1 and C2). It is also interesting that almost every source code in Java has functions, in contrast with values for source codes in C/C++ (see the row '% docs w/functions').

4.2 Evaluation Measures

As the problem is the identification of pairs of source codes that can be considered as plagiarism, we tackle it as a classification problem with two classes: *plagiarized* and *non-plagiarized*. Thus, the evaluation of the effectiveness of our method was carried out by means of the macro F-measure.

The F-measure is widely used for classification tasks and is computed as a linear combination of the precision and the recall values from the two classes. Equation 8 define the measure formally.

$$F-\text{measure} = \frac{1}{|\mathbb{C}|}\sum_{c_i \in \mathbb{C}}\left[\frac{2 \times \text{Recall}(c_i) \times \text{Precision}(c_i)}{\text{Recall}(c_i) + \text{Precision}(c_i)}\right] \quad (8)$$

$$\text{Recall}(c_i) = \frac{\text{number of correct predictions of } c_i}{\text{number of examples of } c_i}$$

$$\text{Precision}(c_i) = \frac{\text{number of correct predictions of } c_i}{\text{number of predictions as } c_i}$$

4.3 Initial Analysis

An initial analysis was performed on the training subset to determine the amount of information given by each proposed feature. Consequently, we carried out a series of experiments using single views (*i.e.*, single feature). Therefore, we measure the performance of each proposed feature by means of establishing a manual threshold for considering when two codes were plagiarized. That threshold was set from 10 to 90 percent of similarity in increments of 10%. At

[1]The entire collection is available from http://users.dsic.upv.es/grupos/nle/soco/

each threshold we evaluated the F-measure. The results of this evaluation are shown in Figure 1.

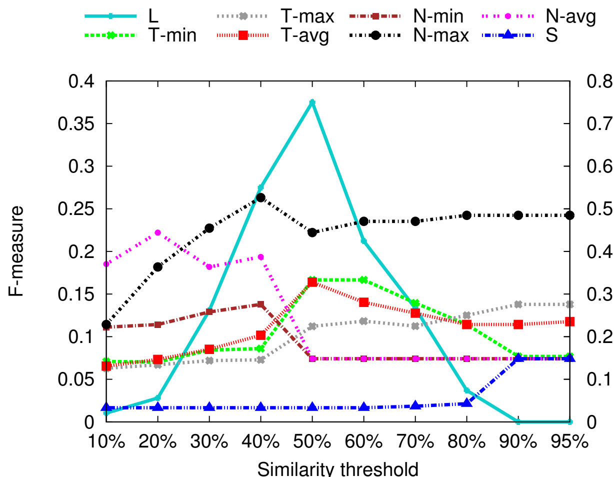


Figure 1: F-measure of classifying source code plagiarism with single features at several similarity threshold. Where, L corresponds to the lexical feature, T means the structure feature taken from the data types, N corresponds to the structure feature taken from the identifiers' names. Note that the right y-axis only applies to the continuous line (Lexical measure).

From graphs in Figure 1 we first notice that the lexical feature is a good indicator at 50% of similarity threshold (F-measure of almost 0.8). For all other features, the F-measure values do not exceed 0.3 of F-score. However, we observe that even with low performance, each type of structural attributes contains information to identify plagiarized cases, some more than other. For instance, feature `N-max` (or $f_{structural}^{N_{max}}$) that is also based on characters 3-grams for the signature's function is the best attribute obtained from the identifier's names of signature's functions.

4.4 Results

This section shows the results of combining our 8 proposed similarity features by means of a supervised approach to classify source code documents into plagiarized and not plagiarized. Based on our initial analysis, we believe that a supervised method will learn the best way of combining the similarity values, instead of try to figure it out manually. For this we use a J48 decision tree implemented in Weka with the default parameters as our classification algorithm [11]. Note that to train the classification model we only use the training set of the data collection, while for testing we used only the test set.

In order to compare the performance of our method, we use the two baselines proposed during the SoCO competition: JPlag [13] and an n-gram based-approach proposed in [8]. JPlag is a popular source code plagiarism tool; it parses and converts each source code into tokens, then using a greedy algorithm it identifies the longest non-overlapped common sub-strings within the tokens. The second baseline uses a character 3-gram model weighted with the term frequency to compute the cosine similarity among a pair of source code (a pair with a similarity above 95% is considered a plagiarism case).

In addition to the previous baselines we propose a third one. Our baseline is based on the similarity value of a pair of source codes that rely solely on the lexical feature described in Section 3.1. This baseline uses a similarity threshold manually set to 50% in accordance with the initial analysis we performed in Section 4.3; that means that a pair of source code is considered a plagiarized case when their lexical similarity is equal or greater to 50%. The Table 3 shows the results of the three baselines and the classification performance of J48 algorithm using our proposed representation.

Table 3: F-measure for 3 baselines and our proposed method on C and Java programming languages

Method	C/C++	Java
JPlag	0.190	0.380
Flores et al [8]	**0.295**	0.556
Lexical feature only	0.013	0.517
Our proposed method	0.013	**0.807**

Our proposed method outperform all three baselines for the Java programming language. However, this is clearly not the case for the C language. To understand this behavior we evaluate each of the 6 different scenarios of the test data independently. Table 4 shows the F-measure for both programming languages on each scenario.

Table 4: F-measure of each 6 scenarios in the test set. A hyphen means that there are not plagiarized cases for that scenario

	A1	A2	B1	B2	C1	C2
C/C++	0.010	0.009	0.024	0.019	**0.737**	-
Java	0.776	0.739	0.847	0.815	-	**1.000**

The F-measure values in Table 4 show that our supervised approach performs better in scenarios C1 and C2 for C/C++ and Java respectively. These results in addition with the information from Table 2 strongly suggest that our proposed approach is more adequate for larger source code documents that seemingly solve more complex problems than on shorter ones that solve simpler problems.

Hitherto, we show the pertinence of our similarity features to identify plagiarized cases on source code documents. In next section, we will discuss to what extent the proposed measures are useful for this task.

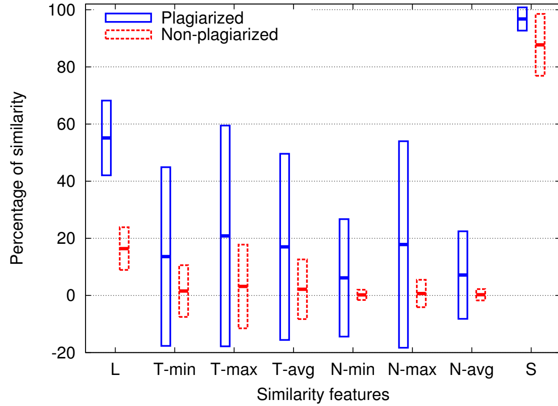
5. FURTHER ANALISYS

This section aims at answering the research questions we posed in this paper, namely *(i)* which aspect of a source code is more important in order to identify plagiarism?; and *(ii)* it is possible to capture some of the structure of source code files by looking at function's signatures?

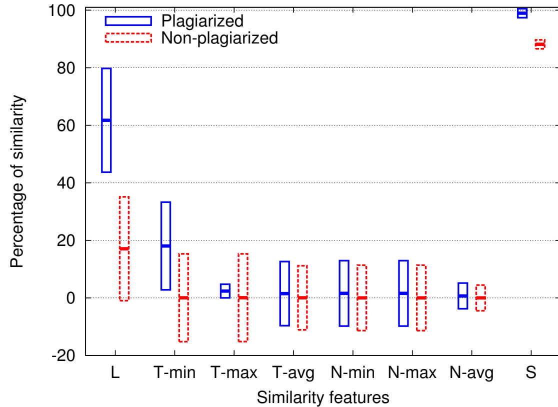
5.1 Similarity among Source Codes from the Same Class

To investigate the impact of each proposed similarity feature we compute the similarity values among pairs of source code labeled as *plagiarized*, *i.e.,* 26 pairs for C and 84 pairs for Java. We also compute the similarity values among pairs of source code labeled as *non-plagiarized*, *i.e.,* 3055 pairs for C and 33,327 pairs for Java. It is worth to mention that we did this computation only on the training set, since this subset is the manually labeled one; therefore, it is reliable.

Accordingly, Figure 2 shows the average and standard deviation of each category per proposed similarity feature. The



(a) C language



(b) Java language

Figure 2: Average similarity and standard deviation per class across proposed features. Middle lines within the bars indicate the average percentage of similarity.

middle lines within the bars represent the average, while the bars show +/− a standard deviation. Continuous lines indicate values for plagiarized cases, while dotted lines illustrate values for non-plagiarized cases. Note that all similarity values are always equal or grater than zero.

From Figure 2a and Figure 2b we are able to draw several conclusions. First of all, the similarity of *non-plagiarized* pairs of source code on both languages is very low, between 0 and 20%, which is consistent with the finding derived from our initial analysis (see Section 4.3).

Secondly, the lexical similarity by itself proved to be a good feature for distinguishing both classes, since the bars (in Figure 2) are not overlapping in either language. The reason could be that the type of plagiarized (posed by Faidhi [6]) present on the data collection are the simpler ones.

Third, there is a high similarity of writing style (stylistic similarity) among source code documents for both programming languages. This finding suggests that the style's features commonly used on text documents might not be well suitable for source codes; that is, we need to focus on finding a good set of stylistics characteristics that capture the particularities of source codes.

In addition, we notice that this stylistic similarity is more useful for Java than for C language, since there is not overlapping on the similarities values for the two classes. This is one reason why in our evaluation we obtained better results in Java than in C language (see Table 3).

Fourth, by looking at the structural attributes, we notice a grater similarity variation among pairs of plagiarized source code in C than the variation for the same class in Java. Therefore, structural attributes are not very helpful for C language.

5.2 Correlation among our Proposed Similarity Features

The analysis carried out in the previous section strongly suggests a correlation among all the structural similarity measures. Therefore, to establish if the proposed type of features, namely *lexical*, *structural* and *stylistic*, provide complementary information of source codes such that different aspects of the source code documents are in fact being captured.

To investigate this inquiry we computed the Pearson correlation values for the 8 proposed similarity features from both programing languages. Tables 5 and 6 shows this information from C and Java programming languages, respectively.

Table 5: Correlation matrix for the proposed similarity measures for the C programming language

Considering non-plagiarized cases								
	L	T_{min}	T_{max}	T_{avg}	N_{min}	N_{max}	N_{avg}	S
L	1							
T_{min}	0.05	1						
T_{max}	0.00	0.76	1					
T_{avg}	0.02	0.88	0.93	1				
N_{min}	0.07	0.39	0.33	0.37	1			
N_{max}	0.10	0.48	0.56	0.54	0.52	1		
N_{avg}	0.12	0.56	0.44	0.53	0.78	0.75	1	
S	0.14	-0.08	-0.06	-0.08	-0.03	-0.043	-0.05	1

Considering plagiarized cases								
	L	T_{min}	T_{max}	T_{avg}	N_{min}	N_{max}	N_{avg}	S
L	1							
T_{min}	0.31	1						
T_{max}	0.46	0.76	1					
T_{avg}	0.42	0.93	0.94	1				
N_{min}	0.25	0.70	0.53	0.65	1			
N_{max}	0.53	0.66	0.96	0.87	0.50	1		
N_{avg}	0.40	0.93	0.84	0.95	0.72	0.79	1	
S	0.33	0.10	0.23	0.18	-0.05	0.20	0.11	1

Table 6: Correlation matrix for the proposed similarity measures for the Java programming language

Considering non-plagiarized cases								
	L	T_{min}	T_{max}	T_{avg}	N_{min}	N_{max}	N_{avg}	S
L	1							
T_{min}	0.01	1						
T_{max}	0.01	0.91	1					
T_{avg}	0.01	0.89	0.93	1				
N_{min}	0.03	0.26	0.22	0.30	1			
N_{max}	0.03	0.28	0.25	0.30	0.97	1		
N_{avg}	0.03	0.24	0.20	0.29	0.99	0.99	1	
S	0.19	0.00	0.00	0.00	0.00	0.00	0.00	1

Considering plagiarized cases								
	L	T_{min}	T_{max}	T_{avg}	N_{min}	N_{max}	N_{avg}	S
L	1							
T_{min}	0.07	1						
T_{max}	0.07	1.00	1					
T_{avg}	0.05	0.85	0.85	1				
N_{min}	0.07	0.89	0.89	0.52	1			
N_{max}	0.07	0.89	0.89	0.52	1.00	1		
N_{avg}	0.07	0.99	0.99	0.92	0.82	0.82	1	
S	0.31	0.05	0.05	0.07	0.02	0.02	0.06	1

The correlation values shown in Tables 5 and 6, support the same conclusion emerged before. That is, similarity features extracted from the structural view are positively correlated. More specifically, structural features originated from the same representation, *i.e.,* data type (T_{min}, T_{max} and T_{avg}) or identifier's name (N_{min}, N_{max} and N_{avg}), are strongly correlated in both classes for both languages. However, the structure similarities extracted from the data types, T_{min}, T_{max} and T_{avg}, versus the features from the

identifiers, N_{min}, N_{max} and N_{avg}, are less correlated for the *non-plagiarized* class (with correlation values from 0.2 to 0.3).

In conclusion, taking different representation from the signature's functions contribute, to some extent, with a slightly different information from distinguish the negative classes (*non-plagiarize*). Nevertheless, only one of the three variation of each representation will be needed.

Regarding the stylistic (S) feature, tables show that it contains important and different information to any of the other seven attributes. Particularly, stylistic feature is specially useful for Java (see the last row in Table 6 for both classes). Likewise, the lexical (L) feature also provides different information to any of other seven attributes, specially for Java (see second column in Tables 5 and 6).

5.3 Impact of Similarity Features per Scenarios

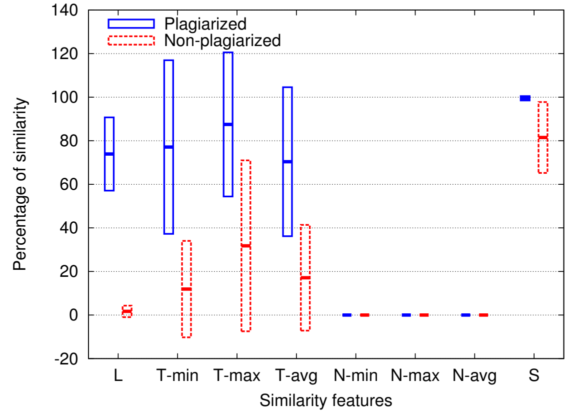
In this latest analysis we study the impact of our proposed features regarding to the difficulty of the problems that source codes in question solve. We carried out the same computation we did in subsection 5.1. We compute the similarity percent of documents in the same class for each scenario in the test set. Recall that each scenario gather source codes that solve the same problem and also the difficulty of the problems increases from scenario A to C [10, 8].

The results we obtained for scenarios A and B are very similar to those we show in Figure 2 for both languages; in contrast with the results we obtained for the most difficult scenario *i.e.,* C1 for C/C++ and C2 for Java.

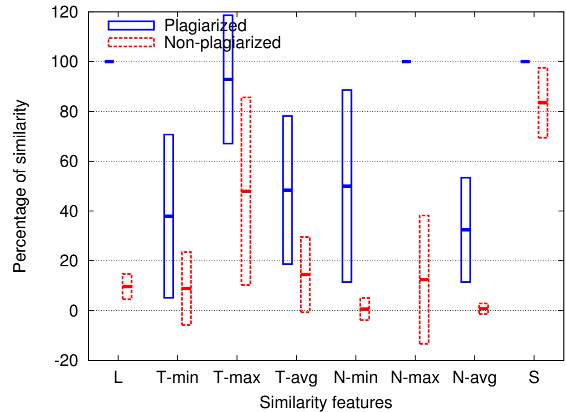
Figure 3 shows the average of similarities and standard deviation for the scenario C1 and C2, for C/C++ and Java, respectively. While lexical (L) and stylistic (S) feature have a similar behavior to that seen before, the structural features tell another story.

To start, the structural features taken from the identifier's names are not useful at all for C/C++. Furthermore, the structural features taken from the data types are very useful for both cases, since there are almost not overlap between classes; additionally, the similarity average are more distant that the values we saw for the training set (in Figure 2).

This new information gives us more evidence on the importance of the structure, captured by the function's signatures only, to identify cases of plagiarism in source code. In particular, source codes that solve complicated problems, that need more lines of codes and, consequently, more functions.



(a) Scenario C1 in C language



(b) Scenario C2 in Java language

Figure 3: Average similarity and standard deviation per class across proposed features. Middle lines within the bars indicate the average percentage of similarity.

6. CONCLUSIONS AND FUTURE WORK

In this paper we proposed three different categories of similarity features; namely, lexical, structural and stylistics similarity features. A combination of these features is able to capture important aspects of source codes, which in turn helps to detect plagiarism patterns.

For the lexical feature, we used a modified representation first proposed by Flores [9]. With respect to the structural aspect, we proposed two similarity metrics that consider the function's signatures within the source code. Finally, for the last similarity feature we defined eleven attributes that intent to extract some stylistic characteristics from the original author that are more difficult to obfuscate. We combine these similarity features in a supervised approach that learn how to classify any pair of source code, using a decision tree (J48) algorithm.

We evaluated our supervised approach in the context of SoCO competition, a shared task that focuses on monolingual source code re-use detection. The data collection we used contains more than 32000 source code documents.

We carried out analysis on the aforementioned features from which lexical and stylistic features show salient characteristics on the task of plagiarism detection. Besides, structural features show complementary characteristics. However, further analysis must be done, for instance the analysis of correlation between the proposed features and the source code length.

For our future work we will focus on two directions. One is related to the addition of more types of similarity feature to our model in order to capture more and diverse aspect of source codes. In the second direction, we will pursue the construction of a model for cross-language plagiarism detection, that is, train a model with examples of plagiarized cases for one language and classify source code written in another language.

7. ACKNOWLEDGMENTS

This work was partially funded by CONACyT Mexico Project Grant CB-2010/153315. Additionally, authors would

like to thank UAM Cuajimalpa for its support.

8. REFERENCES

[1] A. Aiken. Moss, a system for detecting software plagiarism, 1994.

[2] N. Baer and R. Zeidman. Measuring whitespace pattern sequence as an indication of plagiarism. *Journal of Software Engineering and Applications*, 5(4):249–254, 2012.

[3] R. A. Baeza-Yates and B. Ribeiro-Neto. *Modern Information Retrieval*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1999.

[4] A. Barrón-Cedeño, C. Basile, M. Degli Esposti, and P. Rosso. Word length n-grams for text re-use detection. In A. Gelbukh, editor, *Computational Linguistics and Intelligent Text Processing*, volume 6008 of *Lecture Notes in Computer Science*, pages 687–699. Springer Berlin Heidelberg, 2010.

[5] G. Cosma and M. Joy. Evaluating the performance of lsa for source-code plagiarism detection. *Informatica*, 36(4):409–424, 2013.

[6] J. A. W. Faidhi and S. K. Robinson. An empirical approach for detecting program similarity and plagiarism within a university programming environment. *Comput. Educ.*, 11(1):11–19, Jan. 1987.

[7] J. Feng, B. Cui, and K. Xia. A code comparison algorithm based on ast for plagiarism detection. In *EIDWT*, pages 393–397, 2013.

[8] E. Flores, A. Barrón-Cedeño, L. Moreno, and P. Rosso. Uncovering source code reuse in large-scale academic environments. *Computer Applications in Engineering Education*, 2014.

[9] E. Flores, A. Barrón-Cedeño, P. Rosso, and L. Moreno. Towards the detection of cross-language source code reuse. In R. Muñoz, A. Montoyo, and E. Métais, editors, *Natural Language Processing and Information Systems*, volume 6716 of *Lecture Notes in Computer Science*, pages 250–253. Springer Berlin Heidelberg, 2011.

[10] E. Flores, P. Rosso, L. Moreno, and E. Villatoro-Tello. PAN@FIRE: Overview of SOCO track on the detection of source code re-use. In *Proc. of FIRE 2014*, December 2014.

[11] S. R. Garner. Weka: The waikato environment for knowledge analysis. In *In Proc. of the New Zealand Computer Science Research Students Conference*, pages 57–64, 1995.

[12] S. Narayanan and S. Simi. Source code plagiarism detection and performance analysis using fingerprint based distance measure method. In *Proc. of 7th International Conference on Computer Science Education, ICCSE '12*, pages 1065–1068, July 2012.

[13] L. Prechelt, G. Malpohl, and M. Philippsen. Finding Plagiarisms among a Set of Programs with JPlag. 8(11):1016–1038, Nov 2002.

[14] M. Sanderson and J. Zobel. Information retrieval system evaluation: Effort, sensitivity, and reliability. In *Proc. of SIGIR '05*, pages 162–169, New York, NY, USA, 2005.

[15] E. Stamatatos. Plagiarism detection using stopword n-grams. *Journal of the American Society for Information Science and Technology*, 62(12):2512–2527, 2011.