Pertinence of Lexical and Structural Features for Plagiarism Detection in Source Code*

A. Ramírez-de-la-Cruz, G. Ramírez-de-la-Rosa,
C. Sánchez-Sánchez, H. Jiménez-Salazar, and E. Villatoro-Tello

Departamento de Tecnologías de la Información,
División de Ciencias de la Comunicación y Diseño,
Universidad Autónoma Metropolitana Unidad Cuajimalpa. México D.F.
aaron.rc24@gmail.com
{gramirez,csanchez,hjimenez,evillatoro}@correo.cua.uam.mx

Abstract. Source code plagiarism can be identified by analyzing several and diverse views of a pair of source code. In this paper we present three representations from lexical and structural views of a given source code. We attempt to show that different representations provide diverse information that can be useful to identify plagiarism. In particular, we present representations based on 3-grams of characters, data type of function's signatures and names of the identifiers of function's signatures. While we used only three representations, more representations can be added. We conducted our analysis over a collection of 79 source code written in C language. Our results show that n-gram representation is very informative, but also that representations taken from the function's signatures are, to some extend, complementaries.

Key words: Lexical and Structural Features, Similarity Calculation, Document Representation, Plagiarism Detection, Natural Language Processing

1 Introduction

Plagiarism detection in source code is a topic of growing interest of many researches since 1980; specifically when institutions such as the Carnegie-Mellon University created policies for their computer science departments [1] to prevent plagiarism among their students. Recent studies [2,3] have shown that there has been an increment in the number of students that plagiarize source codes. For example, in 1995, according to Rosales et al. [2], less than 2% of the students' code passed through the software `pk2` where found guilty of plagiarism, while in 2006 this percentage grows to an almost 10%.

This problem has became an important topic such that researchers started to define the problem more formally. For instance, in 1987 Faidhi and Robinson

* This work was partially supported by CONACyT México Project Grant CB-2010/153315, and SEP-PROMEP UAM-PTC-380/48510349. We also thank to SNI-CONACyT for their support.

[4] proposed a seven level hierarchy that aimed at representing most of the program's modifications used by students when they plagiarize code. As a consequence, many approaches try to identify code plagiarized based on these levels of complexity, that include: no changes at all (level 0), modification in comments, identifiers, variable position, procedures combination, program statements and logical control, from level 1 to 6 respectively.

In addition, over the past years, several methods have been dealing with detecting source code reuse focusing on two main aspects: structural and superficial[1]. However, in order to detect modifications on structural aspects, a deeper and strict analysis is required and it is often imply to have a complete knowledge of some particular programming language.

It is important to mention that similar to plagiarism over text documents, plagiarist of source code also apply several techniques in order to obfuscate or camouflage the plagiarized sections. Accordingly, it is very difficult to focus on all the possible types of obfuscation and incorporate them into a single method. Nevertheless, as in text documents (written in natural language), source code has structure and meaning, and also has inherently (to some extend) the particularities of the original author's written style. In the knowledge of these similarities between text documents and programming languages, we propose the use of natural language processing techniques to present a methodology that incorporate several views (representations) of the source code, which aim at providing more elements to accurately identify source code reuse. Specifically, this paper proposes and analyzes different representations of a source code, namely: n-grams of characters, data types, and identifiers' names. Our intuitive idea indicates that by means of considering different aspects from a source code (including those evaluated here), it will be possible to capture some of the most common practices performed by the plagiarist when they are camouflaging plagiarized sections.

The rest of this paper is organize as follows. In Section 2 we report related methods and its way to tackle the problem. Then, we present our proposed representations in Section 3. Section 4 shows the experimental settings to test our methodology; also, in this section we present the results obtained over a set of source codes written in the programing language C. Finally, in Section 5, we depict some conclusions and future work lines.

2 Related Work

The plagiarism detection problem has been tackled through several approaches, mainly NLP techniques adapted to the specific content of source code. One such works, take into account a trace that remains after a copy of source code: whitespace patterns [5]. In this work the file is converted to a pattern, namely 'whitespace' format: replacing any visible character by `X`, any whitespace by `S`, and leaving newlines as they appear. The method calculates a similarity

[1] From the more external and internal levels of the hierarchy proposed by [4], respectively

index based on the longest common substring (LCS) of both patterns, LCS: $LCS/\max\{l_1, l_2\}$, where l_i is the size of pattern i. To perform their tests, authors used C source code from free software Apache and Linux Kernel. Using the distribution of similarity index for source code pairs, corroborate their hypothesis: pairs of similar code (different Kernel Linux version) have mean and standard deviation high, and pairs of different source code have mean and standard deviation low. However, there is not significative difference for similar and different source code.

Another very common approach is to determine the fingerprint, as the Moss tool, based on n-grams of words from the source code [6]. It is important to consider several features of source code, such as identifiers, number of lines, number of terms per line, number of hapax, etc. In [7] authors carried out an experiment composing a similarity measure which uses a particular weighting scheme aiming to combine the extracted features.

G. Cosma and M. Joy [8] performed a detailed analysis supported on LSA. They focused their work 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. Furthermore, they build four corpora; using for this purpose the tools Moss and Sherlock [9], and human judgements to clean results given by the tools. In order to achieve a good trade-off between components, they used the best preprocessing (remove comments) then, measured the performance based on MAP to select the best weighting (local frequency, normal global frequency, and normalization of document). Observing curves of MAP over dimensionality were determined 15 dimensions. Besides, they tuned the threshold analyzing the lowest positive matching, highest false matching, and separation (difference between lpm and hfm). Finally, the dimensionality was 30 for such corpora.

Notice that a common aspect among the previous works is that authors try to capture several aspects from source codes into one single/mixed representation (*i.e.,* a single view) in order to detect plagiarism. However, our hypothesis indicates that each aspect (*i.e.,* either structural or superficial elements) provides its own important information that can not be mixed with other aspect when representing source codes. Accordingly, we perform an analysis of two main aspects that we consider among the most discriminative for detecting source code reuse.

3 Proposed Source Code Representations

In this section we describe in detail our proposed representations. These representations are divided into two views (*i.e.,* lexical and structural). The goal is to to determine if it is possible to identify pairs of source code with high similarities, thus providing evidence of plagiarism.

The first view considers the lexical characteristics of the source code and tries to capture some superficial modifications. The second view considers some structural characteristics, *i.e.* signature's programming function from source codes, and it is subdivided into two types: *i)* similarities between data type of function's signatures, and *ii)* similarities between identifiers of function's signatures.

3.1 Lexical view: character 3-grams representation

The approach used in this representation was proposed by Flores Sáez [10]. The main idea was to represent a given source code into a bag of n-grams of characters, B_j, where all the blanks and line-breaks are deleted and the letters are changed into lowercase.

Comparison of two codes Given two codes, C_α and C_β, their bag of 3-grams is computed as we mentioned before; then, each code is represented as a vector $\mathbf{B}_\alpha$ and $\mathbf{B}_\beta$ according to the vector space model proposed by [11]. Finally, the similarity between a pair of source codes is computed using the cosine similarity, which is defined as follows:

$$sim_{lexical}(C_\alpha, C_\beta) = \cos(\theta) = \frac{\mathbf{B}_\alpha \cdot \mathbf{B}_\beta}{\|\mathbf{B}_\alpha\| \|\mathbf{B}_\beta\|} \tag{1}$$

3.2 Structural view: data types from the function's signature representation

As we mentioned before, the proposed structural view consists of two representations. The first representation considers the data types of the function's signatures[2]. The idea behind this representation is based on the intuition that plagiarists often are willing to change function's and argument's names, but not the data types of such elements. Thus, by means of using the data types of function's signatures we attempt to compare some elements that belong, to some extent, to the structure of the program.

Accordingly, first we represent each function's signature into 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)`’.

In order to represent each source code, we need a vocabulary formed by all used data types within the source codes in revision, which is called the *data-type vocabulary*. Then, we form the respective vector for each function. For instance, consider that our vocabulary is formed by the elements ‘<`char, double, float, int`>’, then, the representation of a function with the following signature ‘`int sum(int x, int y)`’ will be `(0,0,0,2)`’. Notice that we are not considering the return type at this step.

[2] We will refer just as function to every programming function within a source code.

Below it is defined how two functions are compared. Later, we define a similarity equation that takes into account all the functions in a pair of source code.

Comparison of two functions. To calculate the similarity between two functions, we need to compare two parts of the function's signature: return data type and arguments data types. We measured the importance of each part independently and then we merge them.

Given two functions, m^α and m^β from C_α and C_β respectively; and the return data type of those functions (*i.e.,* m_r^α and m_r^β), we can compute their similarity as Equation 2 states.

$$sim_r(m_r^\alpha, m_r^\beta) = \begin{cases} 1 \; if \; m_r^\alpha = m_r^\beta \\ 0 \; otherwise \end{cases} \tag{2}$$

For the same two functions, m^α and m^β, the data types of their arguments are represented as vectors $\mathbf{m}^\alpha = [a_1^{m^\alpha}, a_2^{m^\alpha}, ..., a_k^{m^\alpha}]$ and $\mathbf{m}^\beta = [a_1^{m^\beta}, a_2^{m^\beta}, ..., a_k^{m^\beta}]$. Where each positions in the vector represents a data type from the vocabulary of data-type, and the value for each element is the frequency of that type in the function's signature. Hence, the similarity of the arguments' data types is calculated as defined in Equation 3.

$$sim_a(\mathbf{m}^\alpha, \mathbf{m}^\beta) = \frac{\sum_{i=0}^{k} \min(a_i^{m^\alpha}, a_i^{m^\beta})}{\sum_{i=0}^{n} \max(a_i^{m^\alpha}, a_i^{m^\beta})} \tag{3}$$

Once we have the similarities from the return data-type and the arguments' data-type we merge them into a linear combination to find the similarity between m^α and m^β; as in Equation 4.

$$sim(m^\alpha, m^\beta) = \sigma * sim_r(m_r^\alpha, m_r^\beta) + (1 - \sigma) * sim_a(\mathbf{m}^\alpha, \mathbf{m}^\beta) \tag{4}$$

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, presented in a following section, we established $\sigma = 0.5$ so both parts are considered equally important.

So far, we have described how to compute the similarities among functions considering both, their data types as well as their arguments names. Next, we describe how we measure the final similarity of two source codes considering all previous computed information.

Comparison of two codes Given two codes, C_α and C_β, we compute a function-similarity matrix $\mathbf{M}_{\alpha,\beta}^{type}$, where all functions in C_α are compare against all functions in C_β. Thus, the final values of similarity between two codes are defined as in Equation 5.

$$sim_{DataTypes}(C_\alpha, C_\beta) = f(\mathbf{M}_{\alpha,\beta}^{type}) \tag{5}$$

where $f(x)$ represents either the maximum value contained in the matrix, or the average value among all values from the matrix. Therefore, if we select the maximum value it implies that we are being more strict for determining when a pair of source codes are in fact plagiarized; and if we select an average value it indicates that we are being less strict, in other words, less rigorous for determining when two source codes are plagiarized. For example, if it does not matter how many functions are present in both source codes, and if just the occurrence of one exact match (*i.e.,* an equal function) is enough for labeling two source codes as plagiarized, then we must employ the maximum value from the function-similarity matrix. However, in a more relaxed criterion, we can use the average similarity among functions and use this value as the final similarity value between a pair of source codes.

3.3 Structural view: names from the function's signatures representation

As a complement for the previous representation, this one considers the structure by using the names of the functions as well as the name of the arguments. Our intuition is that some plagiarists might try to obfuscate the copied elements by means of changing data types, but not the variable's names.

This representation concatenate the name of the function's name with the name of the arguments. First, we convert every character to its lowercase form and we remove white spaces (if present). Thus, the function '`int sum(int x, int y)`' will be represented as '`sumxy`'. After that, we extracted all the 3-grams of characters and form a bag of 3-grams.

Once we have computed the bag of n-grams, we can compute how similar are a pair of functions. Next, we explain how we compare two functions, and then we define how several functions extracted from a pair of source codes are compare in order to determine the similarity between them.

Comparison of two functions Given two functions, m^α and m^β from C_α and C_β respectively; and their corresponding bag of 3-grams b^{m^α} and b^{m^β}, we compute the similarity of this two functions using the Jaccard coefficient as follows:

$$sim(m^\alpha, m^\beta) = \frac{b^{m^\alpha} \cap b^{m^\beta}}{b^{m^\alpha} \cup b^{m^\beta}} \tag{6}$$

Comparison of two codes Similarly to the previous approach, every function in C_α is compared against every function in C_β. From this comparison we obtain a name-similarity matrix $\mathbf{M}^{names}_{\alpha,\beta}$. Hence, the final similarity values of C_α and C_β is defined as established in Equation 7.

$$sim_{Names}(C_\alpha, C_\beta) = f(\mathbf{M}^{names}_{\alpha,\beta}) \tag{7}$$

where $f(x)$ can be set to the maximum value in the matrix, or the average value from the matrix. The meaning of such selection indicates the level of strength in the criteria to determine plagiarism between two source codes.

4 Experimental Results

The experiments performed aim at analyzing the pertinence of each of the proposed representations when determining the similarity between a pair of source codes. As we mentioned in previous sections, three representations were proposed: *i)* a lexical view, described in Section 3.1; and *ii)* a structural view that is composed by two other representations, namely the data-type representation (section 3.2) and the function and arguments' names (section 3.3).

In order to conduct an analysis of the proposed representations we evaluated our proposed approaches using a subset from the training collection of the competition of Detection of Source Code Re-use (SoCO 2014)[3]. This subset consists of 79 source codes in C language, where 26 pairs were tagged as cases of plagiarism by human experts. It is worth to mention that the relevance judgments do not indicate the direction of the plagiarism , *i.e.,* they do not indicate which source code is the original and which is the copy, hence, we do not detect such phenomena.

For each representation we compute the similarities values of each source code files given in the collection. Then, we measure the performance of each proposed representation by means of establishing a manual threshold for considering when two codes are plagiarized (re-used). That threshold was set from 10 to 90 percent of similarity. For each threshold we evaluated the precision, recall and F-measure (based on the relevance judgments). Notice that, at this stage of our investigation we still not identify cases of source code re-use, rather we want to analyze the pertinence of each of the proposed representations for finding similarities within a pair of codes.

4.1 Experiment 1. Lexical view

For this experiment, we use the implementation done by Picazo et al.[12] of the method proposed by Flores [10] (See section 3.1). The results are presented in Figure 1, where we can see values of precision, recall and f-measure for different threshold's similarities values.

As we can see in the figure, it is clear that when the threshold is very relaxed, the recall (i.e. the number of source code pairs that under the lexical representation are labelled as plagiarize and they actually are) is very good. On the contrary, the precision is very poor since the method identify too many pairs of source code as similar. The opposite situation is presented when we are very strict in the decision of how much similar two codes must be to labeled as plagiarize. However, we found that a good compromise is reached at 80% of similarity, when the f-measure is 0.56.

[3] http://users.dsic.upv.es/grupos/nle/soco/

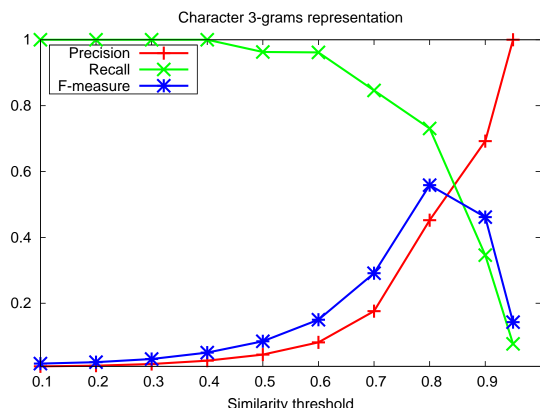

Fig. 1. Lexical view. Best result is obtained with the 80% of similarity between two methods

4.2 Experiment 2. Structural view

In this experiment we evaluated the two approaches: Comparison of Function Signature's Data Types and Comparison of Function Signature's Names.

In both cases, from Equations 5 y 7, we implemented function $f(x)$ as the maximum values of similarity among all the compared functions, and the average of similarities of the methods contained in the files of source codes. With this decision we attempt to compare the performance of this two polices. In this implementation we considered all the functions within the source code except for the `main()` function. The results for the *Comparison of Function Signature's Data Types* approach are presented in Figure 2 and Figure 3.

From the graphs in Figures 2 and 3, we can notice the considerable low performance obtained in comparison with the values obtained with the lexical view. However, this decrement in f-measure is also understandable, since the information from where we computed the similarities is a small part of the whole file of source code, that is function's signature only. Another important remark is that, as we expected, the more strict we are in the policy to determine a pair of source code as plagiarized the similarity among this pair has to be greater (as in the case of using the maximum similarity from the matrix of method similarities computed). The best results are obtained when the similarity is 90% (0.14 of f-measure) when considering the maximum, and 50% (0.16 of f-measure) when considering the average.

An important different between the results are the behavior of precision-recall values. When using the average we can see a expected precision-recall relation (*i.e.*, precision improving while recall decreasing) and we can get a good compromise to find the best configuration. We can not see this pattern when using the maximum.

Regarding to the second approach, *Comparison of Function Signature's Names*,

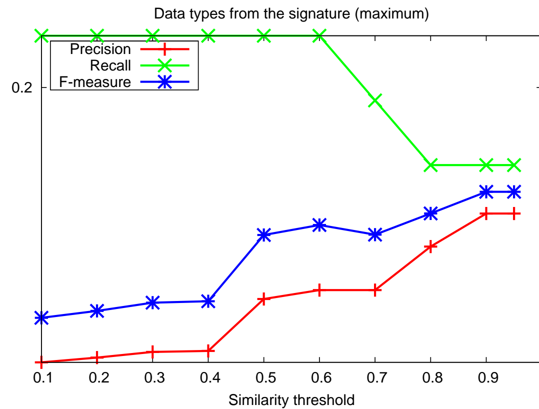

Fig. 2. Structural view: data type of function's signatures using the maximum value of similarities between functions. Best result is obtained with more than 90% of similarity between two methods

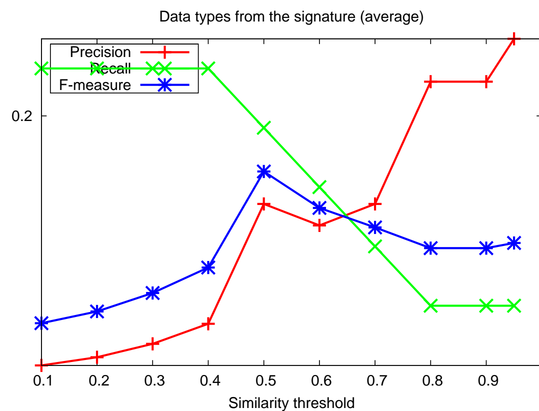

Fig. 3. Structural view: data type of function's signatures using the average value of similarities between functions. Best result is obtained with 50% of similarity between two methods

the graphs in Figure 4 and Figure 5 show that the best F-measure, *i.e.*, 0.26 and 0.22 was obtained when the similarity between codes was 40% and 20%, respectively.

Notice that even when the F-measure is poor for both representation in the structural view, they gives complementary information. While the recall is significant better when the data-type are considered, the precision is much better when the names of the function's signature are taking into account.

Another important remark is that given the results obtained with the lexical view and the fact that this representation uses more information, it might serve

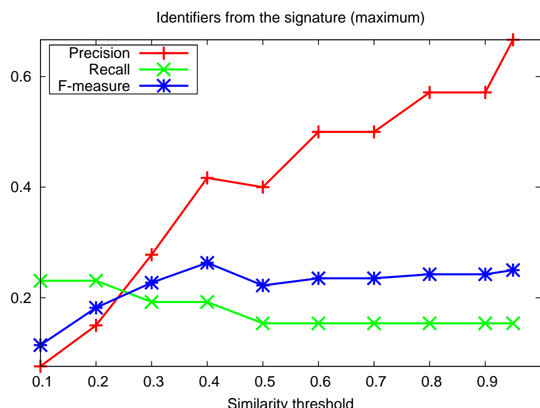

Fig. 4. Structural view: identifiers of function's signatures using the maximum value of similarities between functions. Best result is obtained with 40% of similarity between two methods

as the base approach to consider (or the most important similarity) to identify source code reuse.

More experiments must be done to analyze a good combination of this similarities to build a general method that identify source code plagiarism. Also, this methodology can be extended no only to C language, but to any other language, since our proposed views do not depend of any particularity of a programming language.

5 Conclusions and Future Work

We presented several representations for source code that highlight different views of a source code. In particular, we presented two views: lexical and structural. From the lexical view, we used an implementation of the Flores's method. For the structural view, we proposed two similarities that takes into account function's signatures within the source code. The conducted experiments help us to see that the information from this different approaches can be complementary. Also, we see that the lexical view gives the best similarity since uses the entire source code file. The proposed method can be extended to other views as well to other programming languages.

The future immediate work is to combine different views to determine if a pair of source code is or not been plagiarize. The first idea is learn the most important view, this can be done by looking at the view with the best F-measure. Another idea is to learn this weight in an automatic fashion, using a learning algorithm.

The experiment were performed with a collection of 79 source code in C programing language, but we believe that we can translate the views to another languages, for example Java, without any adjustments to the presented views.

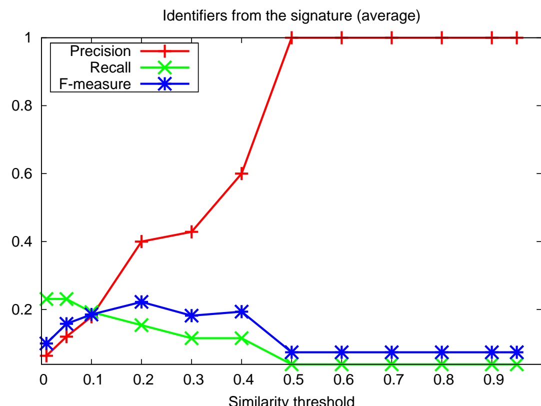

Fig. 5. Structural view: identifiers of function's signatures using the average value of similarities between functions. Best result is obtained with more than 20% of similarity between two methods

References

1. Shaw, M., Jones, A., Knueven, P., McDermott, J., Miller, P., Notkin, D.: Cheating policy in a computer science department. SIGCSE Bull. **12** (1980) 72–76
2. Rosales, F., Dopico, A.G., Rodríguez, S., Pedraza, J.L., Méndez, R., Nieto, M.: Detection of plagiarism in programming assignments. IEEE Trans. Education **51** (2008) 174–183
3. Zhang, D., Joy, M., Cosma, G., Boyatt, R., Sinclair, J., Yau, J.: Source-code plagiarism in universities: a comparative study of student perspectives in china and the uk. Assessment & Evaluation in Higher Education **39** (2014) 743–758
4. Faidhi, J.A.W., Robinson, S.K.: An empirical approach for detecting program similarity and plagiarism within a university programming environment. Comput. Educ. **11** (1987) 11–19
5. Baer, N., Zeidman, R.: Measuring whitespace pattern sequence as an indication of plagiarism. Journal of Software Engineering and Applications **5** (2012) 249–254
6. Aiken, A.: Moss, a system for detecting software plagiarism (1994)
7. Narayanan, S., Simi, S.: Source code plagiarism detection and performance analysis using fingerprint based distance measure method. In: Computer Science Education (ICCSE), 2012 7th International Conference on. (2012) 1065–1068
8. Cosma, G., Joy, M.: Evaluating the performance of lsa for source-code plagiarism detection. Informatica **36** (2013) 409–424
9. Joy, M., Luck, M.: Plagiarism in programming assignments. Education, IEEE Transactions on **42** (1999) 129–133
10. Flores, E.: Reutilización de código fuente entre lenguajes de programación. Master's thesis, Universidad Politécnica de Valencia, Valencia, España (2012)
11. Baeza-Yates, R.A., Ribeiro-Neto, B.: Modern Information Retrieval. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA (1999)
12. Picazo-Alvarez, R., Villatoro-Tello, E., Luna-Ramírez, W.A., Jaimez-Gonzáles, C.R.: Herramienta de apoyo en la detección de reutilización de código fuente. Journal of Research in Computing Science **73** (2014) 45–57