UAM@SOCO 2014: Detection of Source Code Re-use by means of Combining Different Types of Representations

Notebook for SOCO at FIRE 2014

A. Ramírez-de-la-Cruz*
Universidad Autónoma Metropolitana Unidad Cuajimalpa
México D.F.

G. Ramírez-de-la-Rosa†
Universidad Autónoma Metropolitana Unidad Cuajimalpa
México D.F.

C. Sánchez-Sánchez
Universidad Autónoma Metropolitana Unidad Cuajimalpa
México D.F.

W. A. Luna-Ramírez
Universidad Autónoma Metropolitana Unidad Cuajimalpa
México D.F.

H. Jiménez-Salazar
Universidad Autónoma Metropolitana Unidad Cuajimalpa
México D.F.

C. Rodríguez-Lucatero
Universidad Autónoma Metropolitana Unidad Cuajimalpa
México DF.

ABSTRACT

This paper describes the participation of the Language and Reasoning group from UAM-C in the context of the SOurce COde re-use competition (SOCO 2014). We propose different representations of a source code, which attempt to highlight different aspects of a code; particularly: *i)* lexical, *ii)* structural, and *iii)* stylistics. From the lexical view, we used a character 3-gram model without considering all reserved words for the programming language in revision. For the structural view, we proposed two similarity metrics that takes 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. At the end, we combine these different representations in three ways, each of which was a run submission for the SOCO competition this year. Obtained results indicate that proposed representations provide some information that allows to detect particular cases of source code re-use.

Categories and Subject Descriptors

H.3.1 [**Information storage and retrieval**]: Content Analysis and Indexing—*Linguistic processing*; H.3.4 [**Information storage and retrieval**]: Systems and Software

Keywords

Lexical, structural and stylistic attributes, Document representation, Plagiarism detection, Source code re-use

1. INTRODUCTION

Identification of source code re-use is an interesting topic from two points of views. Firstly, the industry that produces software is always looking for protecting their developments, thus they usually search for any sign of unauthorized use of their own blocks of source code. Secondly, in the academic field, it is well known that the habit of copying programs is a common practice among computation students. Such phenomena is also motivated due to all the facilities that web forums, blogs, repositories, etc., offer to share source codes which most of the times have been already debugged and tested.

Consequently, source code re-use detection has became an important research topic, motivating different groups to define the problem more formally in order to build automatic systems to identify such problem. As an example, in 1987 Faidhi and Robinson [5] proposed a seven level hierarchy that aimed at representing most of the program's modifications used by students when they plagiarize source code. As a consequence, many approaches that try to identify plagiarized code are based on these levels of complexity.

However, it is important to notice that programmers that re-use a source code usually apply not one, but several obfuscation techniques when re-using sections from a program. Therefore, even though there are several proposed techniques to detect different types of source code re-use, it is very difficult for a single automatic system to detect all these different types of obfuscation practices.

In this work we propose different representations of a source code, namely: character n-grams, data types, identifiers' names, and stylistics features. Our intuitive idea is that by means of considering different aspects from a source code, it will be possible to capture some of the most common practices performed by programmers when they are re-using a source code.

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 take into account a remanence trace left after a copy of source code, such

*Corresponding author. E-mail address: aaron.rc24@gmail.com

†Principal corresponding author. E-mail address: gramirez@correo.cua.uam.mx

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, 8]. 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 [4]. In this work, authors 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.

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-use. Contrary to these previous works, our hypothesis states that each aspect (*i.e.,* either structural or superficial elements) provides its own important information and can not be mixed with other aspects when representing source codes.

3. SHARED TASK DESCRIPTION

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-use at a document level. The details about the tasks are described in [7].

The data set provided for the shared task is divided into two sets: training and test. The training set has two collections, for C and Java. The Java collection contains 259 source codes while the C collection contains 79 source code. Note that the relevance judgments represent cases of re-use in both directions, *i.e.*, the direction of the re-use is not being detected.

4. PROPOSED SOURCE CODE REPRESENTATIONS

In this section we describe our proposed representations for source code in order to detect several aspects that help to detect source code re-use. We divided these representations into three views: *i.e.,* lexical, structural and stylistics.

4.1 Lexical view: character 3-grams representation

The approach used in this representation was proposed by Flores Sáez [6]. The main idea was to represent source code by means of a bag of character n-grams, B_j, where all the white spaces and line-breaks are deleted and the letters are changed into lowercase. In addition to the original method, we improve the method by eliminating all the reserved words into the document.

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

$$sim_{3grams}(C_\alpha, C_\beta) = \cos(\theta) = \frac{\mathbf{C}_\alpha \cdot \mathbf{C}_\beta}{\|\mathbf{C}_\alpha\|\|\mathbf{C}_\beta\|} \quad (1)$$

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

The proposed structural view consists of two forms of representation. The first representation considers only the data types of the function's signatures[1]. This representation attempts to compare some elements that belong, to some extent, to the structure of the program by means of using the data types of function's signatures.

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)`. Our proposed representation also accounts for the frequency of each data type.

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

Given two functions, m^α and m^β from source codes C_α and C_β respectively. The similarity of their return data type (sim_r) is 1 if they are the same, and 0 otherwise.

The similarity of their arguments data types is a little more elaborated to compute. We use a bag of data-types for each function, we also count each repetition of each data-type. Then we represent each function as a vector. Finally, we compute a similarity between two functions' vectors $\mathbf{m}^\alpha$ and $\mathbf{m}^\beta$ from source codes C_α and C_β respectively as defined in Equation 2.

$$sim_a(\mathbf{m}^\alpha, \mathbf{m}^\beta) = \frac{\sum_{i=0}^{n} \min(\mathbf{m}^\alpha{}_i, \mathbf{m}^\beta{}_i)}{\sum_{i=0}^{n} \max(\mathbf{m}^\alpha{}_i, \mathbf{m}^\beta{}_i)} \quad (2)$$

where n indicates the number of different data types in both source codes,*i.e.,* the vocabulary of data types.

Once we have all the information from the function's signatures, *i.e.*, the similarities from the return data-type and the arguments' data-type; we can compute a single similarity measure. For doing so, we merge the two measures by means of a linear combination, which represents the similarity between m^α and m^β (See Equation 3).

$$sim_1(m^\alpha, m^\beta) = \sigma * sim_r(m^\alpha, m^\beta) + (1-\sigma) * sim_a(\mathbf{m}^\alpha, \mathbf{m}^\beta) \quad (3)$$

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.

Finally, in order to calculate this structural similarity value we perform as follows. Given two codes, C_α and C_β, we compute a function-similarity matrix $\mathbf{M}^{type}_{\alpha,\beta}$, 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 4.

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

$$sim_{DataTypes}(C_\alpha, C_\beta) = f(\mathbf{M}^{type}_{\alpha,\beta}) \quad (4)$$

where $f(x)$ represents either the maximum value contained in the matrix, or the average value among all values from the matrix.

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

As a complement for the previous representation, this representation considers the structure by using the names of the functions as well as the name of the arguments.

This representation concatenate the name of the function's name with the name of the arguments, convert every character into lowercase and removes white spaces (if present). Thus, the function `int sum(int numX, int numY)` is represented as the string `sumnumxnumy`. Then we extracted all the character 3-grams and form a bag of 3-grams.

Once we have computed the bag of n-grams, we can compute how similar are two functions. Given two functions, m^α and m^β from C_α and C_β respectively; and their corresponding vector representation using the bag of 3-grams $\mathbf{m}^\alpha$ and $\mathbf{m}^\beta$, we compute the similarity using the Jaccard coefficient as follows:

$$sim_2(m^\alpha, m^\beta) = \frac{\mathbf{m}^\alpha \cap \mathbf{m}^\beta}{\mathbf{m}^\alpha \cup \mathbf{m}^\beta} \quad (5)$$

Similarly to the previous approach, every function in source code C_α is compared to every function in code 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 6.

$$sim_{Names}(C_\alpha, C_\beta) = f(\mathbf{M}^{names}_{\alpha,\beta}) \quad (6)$$

where $f(x)$ can be set to the maximum value in the matrix, or the average value from the matrix.

4.4 Stylistic view

This representation aims at finding unique properties from the original author such as his/her programming style. In this sense, we compute 11 stylistic features to represent each source code. Then, we use a vector representation and by using a cosine similarity (see Equation 1) we found the similarities between two source code.

The eleven features are: number of lines with code, number of white spaces, number of tabulations, number of empty lines, number of functions, average word length, number of upper case letters, number of lower case letters, number of under scores, number of total number of words in a source code, lexical richness.

5. EXPERIMENTAL EVALUATION

This evaluation was perform with the training provided by the shared task. We carried out a series of experiments using single views in order to find the amount of relevant information given by each representation.

For each experiment we compute the similarities values of each source code files given in the training data. 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.

The results of this evaluation are given in figures 1 to 6. Figure 1 presents the performance of the lexical view, *i.e.*, using a character 3-grams model without considering reserved words. We found that a good compromise between precision and recall is reached at 80% of similarity, when the f-measure is 0.56. Figure 2 shows the performance of the stylistic representation. In general, the results of this representation were not good.

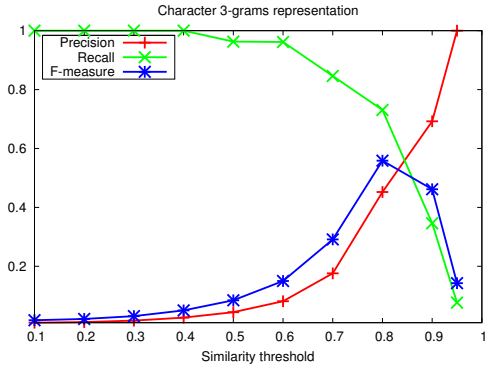

Figure 1: Lexical view. Best result is obtained with the 80% of similarity between two methods

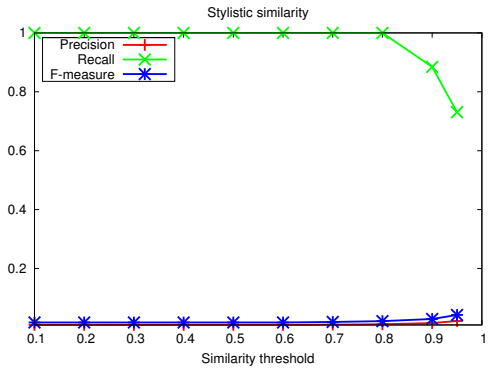

Figure 2: Stylistic view. A high recall is obtained for every similarity threshold, but also very low precision.

For the structural representation, as we mentioned before, we used two representations: data types function's signatures representation and names from the function's signatures representation. For each one, we define two ways to compute the similarity: *(a)* using the maximum value of similarity and *(b)* using the average of the similarities from all the functions into two source codes. Figures 3 and 4 show the data type function's signature representation. 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.

The performance from the second proposed representation, *i.e.* the structural view (name of the function's signature representation) is shown in 5 and 6. The results show that the best F-measure, *i.e.,* 0.26 and 0.22, was obtained

when the similarity threshold between codes was set to 40% and 20%, respectively.

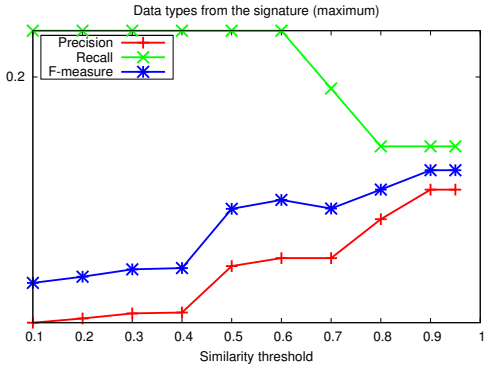

Figure 3: Structural view: data type of function's signatures using the maximum value of similarities between functions. Best result is obtained with more than 90% of similarity between two methods

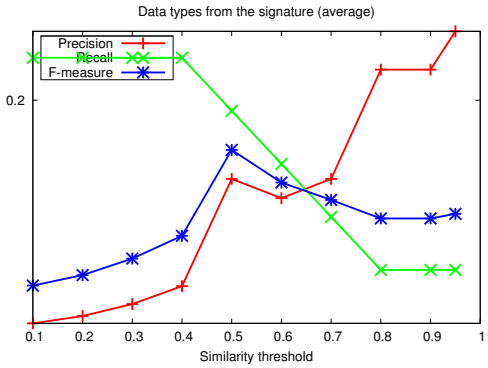

Figure 4: Structural view: data type of function's signatures using the average value of similarities between functions. Best result is obtained with 50% of similarity between two methods

All the results shown here are from C language, however, similar results were obtained in Java data set.

6. SUBMITTED RUNS

We submitted three runs for the task based on three combinations of the proposed representations, considering the performance over the training set. Details about the runs and the results are shown below.

1. **Lexical view only (run 1)**. For this experiment, we used the representation described in section 4.1 using a threshold of similarity of 50%. The results from C and Java are shown in Table 1.

Table 1: Results over the test set for run 1

	Precision	Recall	F-measure
C	0.006	1.00	0.013
Java	0.349	1.00	0.517

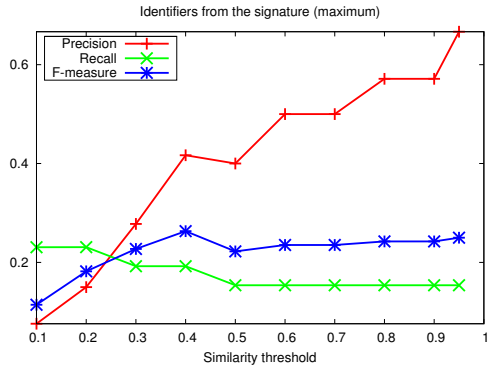

Figure 5: Structural view: identifiers of function's signatures using the maximum value of similarities between functions. Best result is obtained with 40% of similarity between two methods

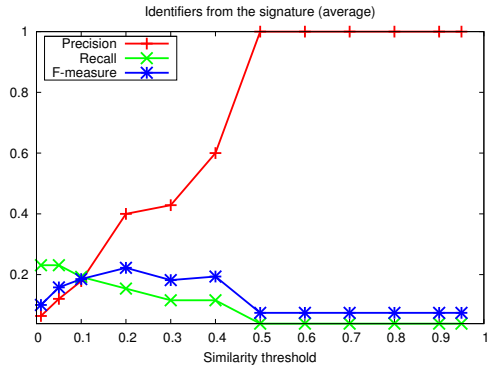

Figure 6: Structural view: identifiers of function's signatures using the average value of similarities between functions. Best result is obtained with more than 20% of similarity between two methods

2. **Combination of Lexical and Structural views (run 2)**. For this experiment, we used a combination of two views: lexical view as in the previous run (LexSim), and the structural view with both representations (DTSim and NameSim). Since the experimental evaluation over the training test shows much better results with the lexical view, we decided to give more weight to this factor. The evaluation experiments also shows that the two representations for the structural view are complementary, thus in order to use this information we use both with equal weight. The Equation 7 shows the employed linear combination.

$$sim = (0.5 * LexSim) + (0.25 * DTSim) + (0.25^{*} NameSim) \quad (7)$$

The results of this experiment are shown in Table 2.

Table 2: Results over the test set for run 2

	Precision	Recall	F-measure
C	0.005	0.950	0.010
Java	0.019	0.928	0.037

3. **Supervised approach (run 3)**. Here we decided to used all the similarities computed from all the views and used a learning algorithm to classify all source code pairs into two classes: re-use and no-re-use. For this we use a J48 decision tree implemented in Weka. The results over the test data are shown in table 3.

Table 3: Results over the test set for run 3

	Precision	Recall	F-measure
C	0.006	0.997	0.013
Java	0.691	0.968	0.807

From previous tables we can see that our best results was achieve for the supervised approach (run3). To get the big picture about the performance in this task, Table 4 shows our best system (run3) agains the best systems in each language and the baseline (the baseline consists of a character 3-gram model weighted using term frequency and cosine measure to compute the similarity. This baseline considers as re-used cases all source code pairs that surpass a similarity threshold of 0.95).

Table 4: Comparison agains the best method

	Precision	Recall	F-measure
C Language			
Our run 3	0.006	0.997	0.013
Best system	0.282	1.00	0.440
Baseline	0.258	0.345	0.295
Java Language			
Our run 3	0.691	0.968	0.807
The 2nd best system	0.530	0.995	0.692
Baseline	0.457	0.712	0.556

From Table 4 we can draw interesting conclusions. First, our obtained recall value for detecting source code re-use in C are competitive with the recall of the best system (1.00 and 0.997), while ours is higher than the baseline. The problem was that our system detected a lot more pairs of source code that were not source code re-used.

The opposite happened with the performances for Java. Here our system performs very well, in recall as well as in precision values, which put our system at the first place in the performance's ranking of all the participant systems in the task.

7. CONCLUSIONS AND FUTURE WORK

In this paper, we have described the experiments performed by the Language and Reasoning group from UAM-C in the context of the SOCO 2014 evaluation exercise. Our proposed system was designed for addressing the problem of source code re-use detection by means of employing different types of representations. Our intuitive idea states that different aspects (*i.e.,* either structural or superficial) provide different (important) information and they must not be mixed with other aspects when representing source codes.

Accordingly, we presented a method that can help to represent a source code in several forms, each of them attempting to highlight different aspect of a code. Particularly, we proposed three representations: *i)* lexical, *ii)* structural and, *iii)* stylistic. From the lexical view, we used a modified implementation of the method proposed by Flores's [6]. For the structural view, we proposed two similarity metrics that consider the function's signatures within the source code. Finally, for the third view we defined eleven features that intent to extract some stylistic attributes from the original author that are more difficult to obfuscate.

Obtained results during the training phase, demonstrate that in fact each type of representation provide some information that can be used to detect some particular cases of source code re-use. A more deep analysis need to be perform in order to determine what are the characteristics of those cases that are accurately detected by each proposed representation and, hence, to come up with a more adequate form of combining these representations.

Finally, obtained results during the test phase motivate us for keep working on the same direction. It is important to remark that although the obtained $F - measure$ were low, it was no the case for the *precision* and *recall* values for the experiments in Java and C respectively. Particularly, for the test experiments performed in the C subset, we believe that the low precision values are due to the fact that several source codes are not just in pure C, and instead, also C/C++ alike programs.

8. ACKNOWLEDGMENTS

This work was supported by CONACyT Mexico Project Grant CB-2010/153315.

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

[5] 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.

[6] E. Flores. Reutilización de código fuente entre lenguajes de programación. Master's thesis, Universidad Politécnica de Valencia, Valencia, España, February 2012.

[7] 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 *Proceedings of the Sixth Forum for Information Retrieval Evaluation (FIRE 2014)*, December 2014.

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