High level features for detecting source code plagiarism across programming languages

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.

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

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

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

ABSTRACT

In this paper we describe the participation of the Language and Reasoning group from UAM-C in the context of the Cross Language SOurce COde re-use competition (CL-SOCO 2015). We proposed a representation of source code pairs by using five high level features; namely: *i*) lexical feature, *ii*) stylistic feature, *iii*) comments feature, *iv*) programmer's text feature, and *v*) structure feature. We combine these different representations in three ways, each of which was a run submission for the CL-SOCO competition. Obtained results indicate that proposed representations provide some information that allows to detect particular cases of source code re-use.

CCS Concepts

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

Keywords

High level feature; Document representation; Plagiarism detection; Source code plagiarism

1. INTRODUCTION

Source code re-use identification has been an interesting topic in two fronts: in the software industry and in the academia. On one hand, software companies are very interested on protect their own software developments; thus, they invest lots of effort and money in trying to do so. On the other hand, the academia, mainly on computing related areas, worries that their students do not plagiarize source code neither from other students nor from the forums on the Internet [5].

The competition of Source Code Re-Use emerge in this context, when in 2014 [8] they invited to the scientific community to a shared task in order to evaluate systems that identify re-use source code cases in a monolingual scenario. Later, in this year 2015, the task considers a cross-language scenario, that is, when a programmer tries to re-use source code from one language, say Java, to another, say C.

In this paper, we present our methodology to solve the problem of finding source code re-use cases across Java and C programming languages. Consequently, we use a set of five high level features within a classification problem, namely: lexical, stylistic, comments, programmer's text, and structural features.

The rest of the paper is organized as follows, in Section 2 we describe the research work more closely related to our proposed methodology. In Section 3, we briefly describe the shared task. Then, in Section 4, we describe the computation of each of our five proposed features. Our Section 5 presents the experiment evaluation carried out on the training set. Section 6 shows the details of submitted runs, and finally in Section 7, we present our conclusions and some future perspectives.

2. RELATED WORK

There are several approaches and tools for finding plagiarism in source code [10]. Some of the most representatives' approaches are those that try to find syntactic similarities through the codes, in the same source language. Some of those are based on searching similar n-grams or small character sequences (strings) between two source codes. Some examples are: the proposed by Flores et al. [7] and, the proposed by Wettel et al. [15] respectively. Likewise, other approaches have tried to detect lexical similarities. For example the proposed approaches by Krinke et al. [12] or Chae et al. [4] that look for graph dependencies (of how methods are called) inside the abstract syntax tree.

Focusing beyond watching a certain characteristic, the approach proposed by Ramirez et al. [13] evaluates a set of three different types of features in order to determine the similarity between code sources. The features are: 1) lexical (character n-grams), 2) structural (function names and parameter names and types), and 3) stylistic (the number of lines of code, the number of white spaces, the number of

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

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). This combination has shown important aspects, with acceptable results, for determining plagiarism between pairs of Java source codes.

On the other hand, some researchers have focused on trying to identify similarities or code clones in software written in different languages. Basically, their methods use an intermediate language to change the codes into it, but at the end, they search for similarities in the same language. An example of such methods is the proposal of Al-Omari et al. [1] in which clones of software, specified in different languages belonging to .NET framework, can be recognized. In this proposal the software is analyzed when it is transformed into the Common Intermediate Language, which is the result of compiling source code in .NET, to finally look for similarities.

Another example of the use of an intermediate language is the work of Brixel et al. [3]. They focus on identifying language-independent code clones, for doing that they preprocessed the source codes, to take them to an intermediate language, and then they use an alignment method based on the parallel principle at local resolution (character level) to compute similarities between documents.

There are other proposals that do not take the source code as main point, instead they focus on the source code file content. That is the case of the approach proposed by Vidhya et al. [14] where two types of metrics are calculated. The first kind of metrics work at method level: Number of lines of code, arguments, function calls, local variables, conditional statements, looping statements, return statements, assignment statements. The second type of metrics work at file level: Number of lines of code, variables declared, methods defined, function calls, and sequences of function calls. The author compare two documents by differences in each of the metrics described before; then, they detect a case of clone using a manual threshold of the average of the differences in the metrics computed.

As can be observed, some of the methods described before are expensive to apply in large collections (for instance those based on compute the syntax tree), others depend on translators for each programming language to being used, and others need a manually thresholds to identify re-use (plagiarized) source code pairs. Contrary to these previous methods, we proposed to use five high level features that are both, easy to compute and considered more than one type of aspect in the source code files.

3. SHARED TASK DESCRIPTION

CL-SOCO, Cross Language Detection of SOurce COde Re-use, is a shared task that focuses on cross-language source code re-use detection. Participant systems were provided with a set of cross-lingual training and test sets of source code files. The task consists on identifying the source code pairs that have been re-use at a document level across programming languages, particularly, Java and C. The details about the tasks are described in [9]. 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. The provided training set has 599 pairs (Java-C) of source code documents and the test set has 79 source code documents.

4. PROPOSED HIGH LEVEL FEATURES

In this section we describe our proposed representation for source code documents as a set of five high level features, in order to identify source code re-use. To compute each of these features, we represent a document in five different ways. Particularly, for the *lexical*, *comments* and *programmer's text* features, we represent each document as a set of characters n-grams; for the *stylistic* feature, we use eleven attributes; and for the *structural* feature we use another ten attributes.

The idea behind these set of high level features is to capture aspect of source code that are inherent to the programmer more than a particular programming language. Thus, the *stylistic feature* capture information about the writing style of the programmer; the *comments' feature* attends for only the information in natural language that the programmer uses to explain the code; *programmer's text feature* takes into account the strings that the program produces, that is, text that, again, little has to do with the programming language *per se*. Additionally, the *lexical* and *structural* features take advantage of the fact that both language at hand (that is, Java and C) share, at some extent, some syntax. Next, we describe each of the five high level proposed features.

Lexical feature. The idea behind this representation is to find a global similarity along the entire document using the representation proposed by Flores [6]. We compute this feature in the same way as is described in [13]. That is, we use a bag of character trigrams where all the white spaces and line-breaks are deleted and the letters are changed into lowercase. Additionally, as we know the language of each source code file *a priori*, we eliminate the reserved words within the document. Consequently, given two source code documents D_1 and D_2 each one is represented as a vector according to the vector space model [2], where the dimension of these vector is given by the vocabulary of character trigrams in both documents. Finally, the lexical feature is compute as the cosine similarity of these two vectors (see Equation 1).

$$sim(D_1, D_2) = \frac{\overrightarrow{D_1} \cdot \overrightarrow{D_2}}{\|\overrightarrow{D_1}\| \|\overrightarrow{D_2}\|} \quad (1)$$

Stylistic feature. As in [13] we took into account a set of eleven stylistic characteristics of the programmer code written style. The characteristics are: 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 feature we use a vector representation for all these attributes and we applied the cosine similarity (Equation 1) between the two vectors.

Comments feature. As we mentioned before, we think that the explanations and details of the procedures that are given in the form of comments have particularities that are inherent to the programmer; thus, these texts allow to capture information that while re-using the code source can be left unmodified. Accordingly, we use everything between blocks $/*...*/$ and everything after // as comments.

First we concatenated all the text written as comments in

the source code document, then we compute the character trigrams. The next procedure was similar to the described previously for the lexical feature; that is, to determine the comments feature we use a vector representation and applied the cosine similarity as in Equation 1.

Programmer's text feature. To compute this feature we considered all the text that were passed as function's arguments (in `prints` sentences, for instance), or string that are assign to some variable (*e.g.* `x="Hello World!"`). All these texts were concatenated together as we did with the previous features. Finally we use the same vector representation using character trigrams; then, by computing the cosine similarity, as in Equation 1, we determine the programmer's text feature for two given source code documents.

Structural feature. For this feature we took into account ten attributes to represent a source code document. These attributes are: the number of relational operations, number of arithmetic operations, number of assignment statements, number of function calls, number of looping statements, number of write access, number of comments, number of functions or procedures defined, number of control flow statements, and number of return statements. These ten attributes form a vector for each document; then, the similarity of two given documents is computed by the cosine similarity given by the Equation 1.

5. EXPERIMENTAL EVALUATION

The evaluation was performed with the training set provided in the shared task (see Section 3). We carried out a series of experiments using single features in order to find the amount of relevant information given by each one of our used high level features.

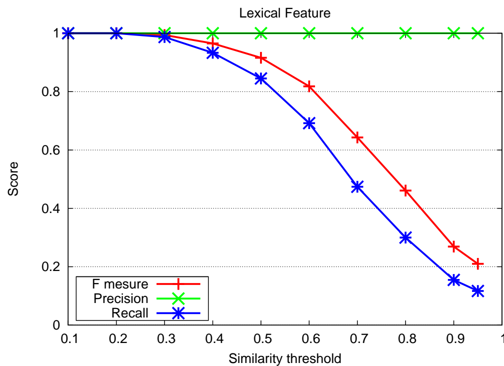

Figure 1: Results of classification when using the *lexical feature* only with manually set thresholds.

For each experiment we computed the similarities values of each source code file given in the training set. Then, we measured 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 in increments of 10%. For each threshold we evaluated the Precision, Recall and F-measure[1].

The results of our evaluation are given in Figures 1 to 5. From these results we can observe that there are three

[1]We compute the F-measure as describe in the evaluation script provided by CL-SOCO 2015 at http://users.dsic.upv.es/grupos/nle/clsoco/

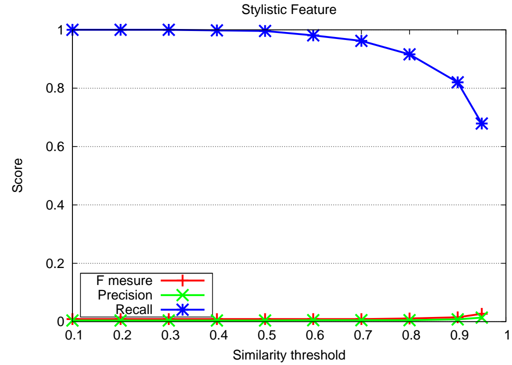

Figure 2: Results of identification of source code re-use when using the *stylistic feature* only with manually set thresholds.

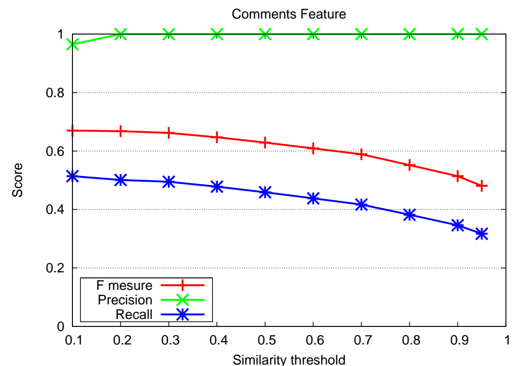

Figure 3: Results of identification of source code re-use when using the *comments feature* only with manually set thresholds.

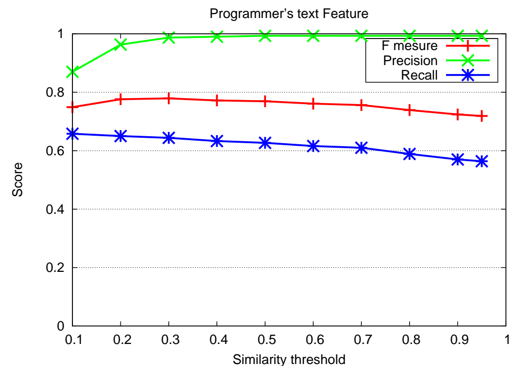

Figure 4: Results of identification of source code re-use when using the *programmer's text feature* only with manually set thresholds.

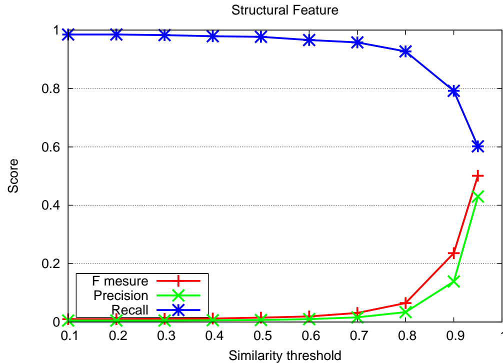

Figure 5: Results of identification of source code re-use when using the *structure feature* only with manually set thresholds.

features, lexical, comments and programmer's text features that perform very well in Precision; this means that by using all the written text in natural language allows to find almost every pair that has been re-use from each other. However, with two features, namely: stylistic and structural, we obtained very good Recall. Consequently, we hypothesize that by using the five high level features as a representation of each pair of code, we may have a compromise of Precision and Recall, hence, a better global performance.

It is worth to mention that we did not consider the entire training set (599 pairs). The reason to do this is that we realized that some pairs labeled as re-use cases were very different (*i.e.*, they solved complete different problems). In order to eliminate noise to our classification models, we computed the similarity (lexical similarity) among the 599 pairs; then, we computed the average ($\overline{x}$) and standard deviation (σ) of those similarity values and we removed all pairs with similarity values below a standard deviation. This way we only conserved pairs with similarity value grater than ($\overline{x} - \sigma$), that give us a total of 477 pairs.

6. SUBMITTED RUNS

We submitted three runs for the posed task based on our proposed representation. For our first two submissions (run 1 and run 2) we tackled the problems as a binary classification problem, where the classes were re-use and not re-use. We trained our model using a Random Forest algorithm with default parameters in the Weka [11] platform.

1. **Monolingual model (Run 1)**. As our five proposed features focus on aspect that little has to do with a specific programming language, we trained a model using source code re-use pairs for C language only, this set was the same provided by SOCO 2014 [8]. This training set contains 79 source code files with 26 pairs of re-use source codes. Once the model was trained we classified the test data (from CL-SOCO 2015) and the results are shown in Table 1.

2. **Cross-language model (Run 2)**. Here we decided to train our model with the training set of cross language examples (the subset of 477 pairs), then we classified the test data and results of this evaluation is shown in the third row in Table 1.

3. **Lexical feature only (run 3)**. For this third run, we compute the lexical similarity of every single pair in the test set. As we see in the training set (Figure 1) we used a threshold of similarity of 30%. The results are shown in Table 1.

Table 1: Results of our submitted runs.

	Precision	Recall	F-measure
Monolingual model (run 1)	0.988	0.634	**0.772**
Cross-language model (run 2)	0.620	0.771	0.687
Lexical feature only (run 3)	0.496	0.962	0.655

In Table 1 we can see that our best model is the monolingual one. This result validates, to some extent, that re-use cases can be identified by aspects that has to do more with the text in natural language than information of particular programming language.

To get a better idea of the performance in this task, in Table 2 we show our best system (run 1) against the average performance results of all participant systems and the second best system. It is worth to mention that our system has the best performance out of a total of 12 systems.

Table 2: Comparison among all participants systems.

	Precision	Recall	F-measure
Our Run 1	0.988	0.634	0.772
Second best	1.000	0.603	0.752
Average all systems	0.916	0.611	0.706

Table 2 shows that our monolingual model globally outperforms the others systems, that is, among all the actual re-use pairs we effectively identify most of them, however we are only 98.8% sure that they are in fact re-use cases.

7. CONCLUSIONS

In this paper, we have described the experiments performed by the Language and Reasoning group from UAM-C in the context of the CL-SOCO 2015 evaluation exercise. Our proposed system was designed for addressing the problem of cross-language source code re-use detection by means of employing five high level features within a classification problem.

Particularly, we proposed the following features: *i*) lexical feature, *ii*) stylistic feature, *iii*) comments feature, *iv*) programmer's text feature, and *v*) structure feature. These features are more oriented to detect aspects that the programmers leave in natural language more than in a particular programming language.

Obtained results indicate that our proposed features can, to some extent, identify cases of source code re-use across Java and C programming language. A deeper analysis need to be perform in order to determine which feature are the most useful in this task and if they are o not correlated.

8. ACKNOWLEDGMENTS

Authors would like to thank UAM Cuajimalpa for its support.

9. REFERENCES

[1] Al-Omari, F., Keivanloo, I., Roy, C. K., and Rilling, J. Detecting clones across Microsoft.NET programming languages. In *Reverse Engineering (WCRE), 2012 19th Conference on Working* (2012), IEEE, pp. 405–414.

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

[3] Brixtel, R., Fontaine, M., Lesner, B., Bazin, C., and Robbes, R. Language-independent clone detection applied to plagiarism detection. In *Source Code Analysis and Manipulation (SCAM), 2010 10th IEEE Working Conference on* (2010), IEEE, pp. 77–86.

[4] Chae, D.-K., Ha, J., Kim, S.-W., Kang, B., and Im, E. G. Software plagiarism detection: a graph-based approach. In *Proceedings of the 22nd ACM international conference on Conference on information & knowledge management* (2013), ACM, pp. 1577–1580.

[5] Chuda, D., Navrat, P., Kovacova, B., and Humay, P. The issue of (software) plagiarism: A student view. *IEEE Transactions on Education 55*, 1 (February 2012), 22–28.

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

[7] Flores, E., Barrón-Cedeno, A., Rosso, P., and Moreno, L. Towards the detection of cross-language source code reuse. In *Natural Language Processing and Information Systems*. Springer Berlin Heidelberg, 2011, pp. 250–253.

[8] Flores, E., Rosso, P., Moreno, L., and Villatoro-Tello, E. 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).

[9] Flores, E., Rosso, P., Moreno, L., and Villatoro-Tello, E. PAN@FIRE 2015: Overview of CL-SOCO track on the detection of cross-language SOurce COde Re-use. In *Proceedings of the Seventh Forum for Information Retrieval Evaluation (FIRE 2015)* (December 2015).

[10] Gondaliya, T. P., Joshi, H., and Joshi, H. Source code plagiarism detection ,SCPDet: A review. *International Journal of Computer Applications 105*, 17 (November 2014), 27–31.

[11] Hall, M., Frank, E., Holmes, G., Pfahringer, B., Reutemann, P., and Witten, I. H. The WEKA data mining software: An update. *SIGKDD Explorations Newsletter 11*, 1 (2009), 10–18.

[12] Krinke, J. Identifying similar code with program dependence graphs. In *Reverse Engineering, 2001. Proceedings. Eighth Working Conference on* (2001), IEEE, pp. 301–309.

[13] Ramírez-de-la Cruz, A., Ramírez-de-la Rosa, G., Sánchez-Sánchez, C., Luna-Ramírez, W., Jiménez-Salazar, H., and Rodríguez-Lucatero, C. UAM@SOCO 2014: Detection of source code re-use by mean of combining different types of representacions.

[14] Vidhya, K., Sumathi, N., and Ramya, D. Cross language higher level clone detection-between two different object oriented programming language source codes. In *Proceedings of International Conference on Inter Disciplinary Research in Engineering and Technology 2014* (2014), ASDF, pp. 21–27.

[15] Wettel, R., and Marinescu, R. Archeology of code duplication: Recovering duplication chains from small duplication fragments. In *Symbolic and Numeric Algorithms for Scientific Computing, 2005. SYNASC 2005. Seventh International Symposium on* (2005), IEEE, pp. 63–70.