PAN@FIRE: Overview of SOCO Track on the Detection of SOurce COde Re-use

Enrique Flores[1], Paolo Rosso[1] Lidia Moreno[1], and Esaú Villatoro-Tello[2]

[1] Universitat Politècnica de València, Spain,
{eflores,prosso,lmoreno}@dsic.upv.es
[2] Universidad Autónoma Metropolitana, Unidad Cuajimalpa, México
evillatoro@correo.cua.uam.mx

Abstract. This paper summarizes the goals, organization and results of the first SOCO competitive evaluation campaign for systems that automatically detect the source code re-use phenomenon. The detection of source code re-use is an important research field for both software industry and academia fields. Accordingly, PAN@FIRE task, named SOurce COde Re-use (SOCO); focused on the detection of re-used source codes in C/C++ and Java programming languages. Participant systems were asked to annotate several source codes whether or not they represent cases of source code re-use. In total three teams participated and submitted 13 runs. The training set consisted of annotations made by several experts, a feature which turns the SOCO 2014 collection in a useful data set for future evaluations and, at the same time, it establishes a standard evaluation framework for future research works.

Keywords: SOCO, Source code re-use, Plagiarism detection, Evaluation framework, Test collections

1 Introduction

Nowadays, the information has become easily accessible with the advent of the Web. Blogs, forums, repositories, etc. have made source code widely available to be read, to be copied and to be modified. Programmers are tempted to re-use debugged and tested source codes that can be found easily on the Web. The vast amount of resources on the Web makes the manual analysis of suspicious source code re-used unfeasible. Therefore, there is an urgent need for developing automatic tools capable of accurately detect the source code re-use phenomenon.

Currently, software companies have a special interest in preserving their own intellectual property. In a survey applied to 3,970 developers, more than 75 percent of them admitted that have re-used blocks of source code from elsewhere when developing their software[3]. Moreover, on the one hand, the academic environment has also become a potential scenario for research in source code re-use because it is a frequent practice among students. A recent survey [3] reveals that

[3] http://www.out-law.com/page-4613

source code plagiarism represents 30% of the cases of plagiarism in the academia field. In this context, students are tempted to re-use source code because, very often, they have to solve similar problems within their courses. Hence, the task of detecting source code re-use becomes even more difficult, since all the source codes will contain (to some extent) a considerable thematic overlap. On the other hand, detection of source code re-use in programming environments, such as programming contests, has an additional challenge, this is the large number of source codes that must be processed for detecting such practises [6], and as a result, source code re-use detection becomes some how unfeasible. Consequently, most of the research on source code re-use detection has been mostly applied to closed groups [16, 14, 9].

Traditionally, the source code re-use detection problem has been approached by means of two main perspectives: *i)* feature comparison, and *ii)* structural comparison. In the first approach, *i.e.* features comparison, the similarity between two programs considers features such as the average number of characters per line, the number of commented lines, etc. [18]. On the contrary, the structural comparison usually takes into account a more complex representation, *e.g.* the representation of a source code made by a compiler, which it can be seeing as a fingerprint representing the structure of a program; then different techniques are applied in order to determine whether or not a case of re-use exists. As examples of structural approaches are [14, 2, 7]. In [14], authors search for the longest non-overlapped common substrings between a pair of fingerprints, whilst in [2], the source code is represented as a dependency graph in order to search for common sub-graphs. Finally, DeSoCoRe [7] proposes a comparison of two source codes at function-level and looks for highly similar functions or methods in a graphical representation.

Whereas at PAN@CLEF the shared task addresses plagiarism detection in texts [13], PAN@FIRE focuses on the detection of source codes that have been re-used in a monolingual context, *i.e.,* using the same programing language. It is worth mentioning that such situation represents a common scenario in the academic environment. Particularly, SOCO involves identifying and distinguishing the most similar source code pairs among a large source code collection. In the next sections we will first define the task and then summarise all participant systems approach as well as their obtained results during the SOCO 2014 shared task.

2 Task description

SOCO shared task focuses on monolingual source code re-use detection, which means that participant systems must deal with the case where the suspicious and original source codes are written in the same programing language. Accordingly, participants are provided with a set of source codes written both in C/C++ and Java languages, where source codes have been be tagged by language to ease the detection. Thus the task consists in retrieving source code pairs that have been re-used. It is important to mentions that this task must be performed at docu-

ment level, hence no specific fragments inside of the source codes are expected to be identified; only pairs of source codes. Therefore, participant systems were asked to annotate several source codes whether or not they represent cases of source code re-use.

This year's task was divided in two main phases: training and testing. For the training phase we provided an annotated corpus for each programming language, *i.e.,* C/C++ and Java. Such annotation includes information about whether a source code has been re-used and, if it is the case, what its original code is. It is worth mentioning that the order of each pair was not important[4], *e.g.*, if X has been re-used from Y, it was considered as valid to retrieve the pair X–Y or the pair X–Y. Finally, for the testing phase the only annotation that has been provided is the one corresponding to the programming language.

3 Corpus

In this section we describe the two corpora used in the SOCO 2014 competition. For the training phase, a corpus composed by source codes written in C and Java programming languages was released. For the testing phase, participants were provided with source codes written in C-like languages (*i.e.,* C and C++) and also in Java language.

3.1 Training Corpus

The training collection consists of source codes written in C and Java programming languages. For the construction of this collection we employed the corpus used in [1]. Source code re-use is committed in both programming languages but only at monolingual level. The Java collection contains 259 source codes, which are labelled from 000.java to 258.java. The C collection contains 79 source codes, labelled from 000.c to 078.c. Relevance judgements represent re-used cases in both directions($X \rightarrow Y$ and $Y \rightarrow X$). Table 1 shows the characteristics of the training corpus and the κ value of the inter-annotator agreement [5].

Table 1. Total number of source codes and re-used source code pairs annotated by three experts. The last column shows their κ value for inter-annotator agreement.

Programing language.	Number of source codes	Re-used source code pairs	Inter-annotator agreement
C	79	26	0.480
Java	259	84	0.668

As can be seen in Table 1 the inter-annotator agreement for the C collection represents a *moderate* agreement whilst for the Java collection the kappa value

[4] An additional challenge in plagiarism detection is to determine the *direction* of the plagiarism, *i.e.,* which document is the original and which the copy.

indicates a *substantial* agreement between annotators [5]. Such results indicate (to some extent) that the provided training corpus represents a reliable data set.

3.2 Test Corpus

The provided test corpus is divided by programming language (C/C++ and Java) and by scenarios (*i.e.,* different thematic). This corpus has been extracted from the 2012 edition of Google Code Jam Contest[5]. Each programming language contains 6 monolingual re-use scenario (A1, A2, B1, B2, C1 and C2). Hence, the name of the files consists of the name of the scenario which they belong to and an identifier, for example, file "B10021" belongs to scenario B1 and its identifier number is 0021. Table 2 shows the number of source codes for each programming language in every scenario.

Table 2. Number of source codes that the test corpus contains by programming language and scenario.

	A1	A2	B1	B2	C1	C2	Total
C/C++	5,408	5,195	4,939	3,873	335	145	19,895
Java	3,241	3,093	3,268	2,266	124	88	12,080

It is important to mention that there is no re-use cases between scenarios, therefore participant systems just need to look for re-used cases among the source code files inside each scenario. For example, participants do not have to submit a re-used case between files "B10021" and "B20013". Notice that the first one belongs to scenario B1 but the second one belongs to B2.

As can be noticed in Table 2, the amount of source codes in the test set is significantly higher than the amount of codes in the training corpus. Therefore, participant systems are some how obligated to develop efficient applications for solving the SOCO task.

Due to the huge size of the corpus is practically impossible to label it manually by human reviewers. Hence, in order to evaluate the performance of participant systems, we prepared a pool formed by the provided detections from the different submitted runs [17]. By means of following this technique, a source code pair needs to appear at least in the 66% of the competition runs to be considered as a relevant judgement. Thus, for the construction of the relevance judgements, we considered all the submitted runs from participant systems with the addition of the two baselines described in the next section. Table 3 indicates the number of identified relevant judgements for each programming language and scenario.

[5] https://code.google.com/codejam/contest/1460488/dashboard

Table 3. Number of relevant judgements per programming language and scenario.

	A1	A2	B1	B2	C1	C2	Total
C/C++	105	92	95	50	8	0	350
Java	115	106	138	76	4	25	464

4 Evaluation Metrics

All the participants were asked to submit a detection file with all the considered re-used source code pairs. Participants were allowed to submit up to three runs. All the results were required to be formatted in XML as shown below. As can be noticed, for each suspicious source code pair it must be one entry of the <reuse_case .../> in the XML file.

```
<document>
<reuse_case source_code1="X1" source_code2="Y1" />
<reuse_case source_code1="X2" source_code2="Y2" />
...
</document >
```

To evaluate the detection of re-used source code pairs we calculate Precision, Recall and F_1 measure. For ranking all the submitted runs we used the F_1 measure in order to favour those systems that were able to obtain (high) balanced values of precision and recall.

Two baselines have been considered for the SOCO 2014 task, which are described below:

- **Baseline 1.** Consists of the well-known JPlag tool [14] using its default parameters. In this model, the source code is parsed and converted into token strings. The greedy string tiling algorithm is then used to compare token strings and identify the longest non-overlapped common substrings.
- **Baseline 2.** Consists of the character 3-gram based model proposed in [8]. In this model, the source code is considered as a text and represented as character 3-grams, where these n-grams are weighted using term frequency scheme. As pre-processing, whitespaces, line-breaks and tabs are removed. All the characters are casefolded and characters repeated more than three times are truncated. Then, the similarity between two source codes is computed using the cosine similarity measure. For this baseline, a code pair is considered as a re-use case if the similarity value is higher than 0.95.

5 Participation Overview

In total three teams participated and submitted 13 runs. Particularly, the Autonomous University of the State of Mexico (`UAEM`) and the Universidad Autónoma

Metropolitana - Unidad Cuajimalpa (UAM-C) have submitted three runs in both programming languages while the Dublin City University (DCU) only in Java.

UAEM [11] used a model for the detection of source code re-use that is divided into four phases. In the first phase only the lexical items of each source language are separated and more than one whitespaces are removed. In the second phase, a similarity measure is obtained for each source code regarding the other source codes. The second phase uses as similarity measure the sum of the different lengths of the longest common substrings between the two source codes, normalised to the length of the longest code. Using the comparisons made for each source code, in the third phase a set of parameters is obtained that allow later the identification of re-used cases. The parameters obtained are: the value of the *distance* (1- similarity), the *ranking* of the distance (rank order of the most similar), the *gap* that exists with the next closest code (it is only calculated for the first 10 closest codes) and, using the maximum gap between the 10 most closest codes, the source codes that are *B*efore or *A*fter the maximum gap *relative difference* are labelled. The result of the third phase is a matrix where each row represents a comparison of a source code with other codes (columns).

For the decision, a source code pair $X \leftrightarrow Y$ will be a re-used case if there is evidence of re-use in both directions, it means, $X \rightarrow Y$ and $Y \rightarrow X$. A re-used case exists when the *distance* is less than 0.45 or the *gap* is greater than 0.14, but also it is important that one of the additional conditions is achieved. The first condition is that the *ranking* must be, at least, in the second position and, the second condition, that the label of the *relative difference* must be *Before*. The first run for C and Java languages were processed with above conditions. However, in some cases the evidence in one direction was very high and in the other direction was almost reliable, but according to the training corpus in Java, in most of the cases this pair was a re-used case. In the second run, if there were not high evidence of re-use in one direction, then the pair can be considered as re-used case whether at least one of the both codes has the *ranking* of 1 and the *relative difference* of *B*efore and the *gap* greater than 0.1.

UAM-C [15] represents the source code in three views attempting to highlight different aspects of a source code: *lexical*, *structural* and *stylistics*. From the lexical view, they represent the source code using a bag of character 3-grams without the reserved words for the programming language. For the structural view, they proposed two similarities that take into account functions' signatures within the source code, e.g., the data types and the identifier names of the functions' signature. The third view consists in accounting for the stylistics' attributes, such as, number of spaces, number of lines upper letters, etc. For each view they computed a similarity value for each pair of source codes and then they established a threshold calculated on the training corpus. In the first run, they only consider the first view with a manually defined similarity threshold of value 0.5. In the second run, they use the first and the second view. From these two views they have three different similarities: lexical similarity (L), data types similarities (DT), and identifiers name similarity (IN). Then, they combined them as: $0.5L * 0.25DT * 0.23IN$, according to a confidence's level manually

established. In the third run, they uses 8 similarities derived from the three proposed views: one similarity for the lexical view, 6 similarities from the second view and 1 more for the stylistic view. Finally, they trained a model using a supervised approach to be used over the test corpus.

DCU [10] undertakes an information retrieval (IR) based approach for addressing the source code plagiarism task. First, they employ a Java parser to parse the Java source code files and build an annotated syntax tree (AST). Then, they extract content from selective fields of the AST to store as separate fields in a Lucene index. More specifically speaking, the nodes in the AST from which they extract information from are the *statements*, *class names*, *function names*, *function bodies*, *string literals*, *arrays* and *comments*. For every source code in the test corpus, they formulate a pseudo-query by extracting representative terms (those having the highest language modelling query likelihood estimate scores). A ranked list of 1000 documents along with their similarities with the query is retrieved after executing the query. The retrieval model that they use is language model (LM). Their model walks down this ranked list (sorted in decreasing order by the similarities) of documents and stops where the relative decrease in threshold in comparison to the previous document similarity is less than a predefined threshold value acting as a parameter. The documents collected this way are then reported as the re-used set of documents. In the first run, separate fields are created for each AST node type, e.g. the terms present in the class names and the method bodies are stored in separate fields. They compute relative term frequency statistics for each field separately. In the second run, an AST is constructed from the program code using a Java parser and then bag-of-words from the selected nodes of the AST are used. However, separate fields are not used to store the bag-of-words. The index is essentially a flat one. In the third run, a simple bag-of-words document representation is used for a program code, i.e., no program structure is taken into account.

6 Results and Analysis

The results obtained by the participants are shown in Table 4 for the programming language C and in Table 5 for Java. As we mentioned before, we ranked obtained results by means of the F_1 measure, given that we prefer systems that are able to obtaing (high) balanced values of Precisions and Recall.

The best results according to F_1 were obtained by UAEM in C and UAM-C in Java. In the C programming language, the two runs of UAEM were able to retrieve all the re-used source code pairs. The rule introduced for retrieving less obvious re-used cases in run 2 had a negative impact on the performance in terms of precision and, therefore, of F_1. Results by UAM-C have been adversely affected in terms of precision by the huge number of retrieved source code pairs (+50K). This may happened because they have removed reserved words and taken into account the number of functions according to the C language. C++ language includes new characteristics such as classes and methods and also includes new reserved words (*e.g. cin* or *cout*). Contrary to the UAEM-C, the other two teams,

as well as the baselines run do not retrieved such amount of re-used pairs. Such fact has a direct impact on the formed pool, since it only takes into account the retrieved pairs with a certain degree of agreement (at least 4 out of 7 runs). has been hindered by the source codes written in C ++ which include classes, methods of classes C not included.

Table 4. Overall evaluation results for C/C++ programing language. The ranking is upon the F_1 values. Baseline 1 corresponds to JPlag model and baseline 2 corresponds to a character 3-grams based model.

Position	Run	F_1	Precision	Recall
1	UAEM-run 1-2	0.440	0.282	1.000
2	UAEM-run 3	0.387	0.240	1.000
#	baseline 2	0.295	0.258	0.345
#	baseline 1	0.190	0.350	0.130
3	UAM-C-run 1	0.013	0.006	1.000
4	UAM-C-run 3	0.013	0.006	0.997
5	UAM-C-run 2	0.010	0.005	0.950

In the Java scenario, `UAM-C` has achieved the best performance with a balance between precision and recall. The combination of all the similarities (lexical, structural and stylistic) measures using a supervised decision tree has been decisive. The second run has been affected by the same phenomenon than in the C scenario: it retrieved +10K re-used source code pairs. The three runs of `DCU` achieved a similar performance. In the second run, the addition of the bag-of-words to the selected nodes of the AST slightly improved the performance of the first run. `DCU` did not select nodes to create a bag-of-words in the third run. This fact has generated slightly lower results

Table 5. Overall evaluation results for Java programing language. The ranking is upon the F_1 values. Baseline 1 corresponds to JPlag model and baseline 2 corresponds to a character 3-grams based model.

Position	Run	F_1	Precision	Recall
1	UAM-C-run 3	0.807	0.691	0.968
2	DCU-run 2	0.692	0.530	0.995
3	DCU-run 3	0.680	0.515	1.000
4	DCU-run 1	0.602	0.432	0.995
#	baseline 2	0.556	0.457	0.712
5	UAEM-run 1	0.556	0.385	1.000
6	UAM-C-run 1	0.517	0.349	1.000
#	baseline 1	0.380	0.542	0.293
7	UAEM-run 2-3	0.273	0.158	1.000
8	UAM-C-run 2	0.037	0.019	0.928

In general, different approaches were applied to solve the problem of source code re-use detection. Proposed approaches vary from string-matching to abstract syntax tree based models. Additionally, given that all these approaches were evaluated under the same conditions employing the same collections, it was possible to make a more fair comparison among participant systems. Accordingly, the best performing model was the string-matching based in C [11] while a combination of lexical, structural and stylistic in Java [15].

7 Remarks and Future Work

In this paper we have presented the overview of the Detection of SOurce COde Re-use (SOCO) PAN track at FIRE. Especially, SOCO 2014 has provided a task specification which is particularly challenging for participating systems. The task was focused on retrieving cases of re-used source code pairs from a large collection of programs. At the same time, SOCO has provided an evaluation framework where all participants were able to compare their obtained results by means of applying different approaches under the same conditions and using the same corpora. With these specifications, the task has turned out to be particularly challenging and well beyond the current state of the art of participant systems.

In total three teams participated and submitted 13 runs. We summarise the followed approaches by each of the participant systems and presented the evaluation of submitted runs along with its respective analysis. In general, different approaches were proposed, varying from string-matching to abstract syntax tree based models. It is important to notice that the participation for the Java language was much higher than for the C programming language (8 *vs.* 5 runs). Nevertheless, the team that achieved the best results in both scenarios (*i.e.,* C/C++ and Java) was the `UAEM` by means of their string-matching approach.

Finally, a note has to be made with respect to the re-usability of test collections, which were calculated using a pool formed by submitted and baseline runs; is that more experiments need to be performed in order to construct a more fine-grained relevance judgements. Nonetheless, both training and test collections represent a valuable resource for future research work on the field of source code re-use identification.

Acknowledgement

We want to thank C. Arwin and S. Tahaghoghi for providing the training collection. PAN@FIRE (SOCO) has been organised in the framework of WIQ-EI (EC IRSES grant n. 269180) and DIANA-APPLICATIONS (TIN2012-38603-C02-01) research projects. The work of the last author was supported by CONACyT Mexico Project Grant CB-2010/153315, and SEP-PROMEP UAM-PTC-380/48510349.

References

1. Arwin, C., Tahaghoghi, S.: Plagiarism detection across programming languages. Proc. 29th Australasian Computer Science Conference, 48, 277–286 (2006)
2. Chae, D., Ha, J., Kim, S., Kang, B., Im, E.: Software plagiarism detection: a graph-based approach. Proc. 22nd ACM Int. Conf. Information & Knowledge Management, CIKM-2013, 1577–1580 (2013)
3. Chuda, D., Navrat, P., Kovacova, B., Humay, P.: The issue of (software) plagiarism: A student view. IEEE Trans. Educ., 55, 22-28 (2012)
4. FIRE (ed.): FIRE 2014 Working Notes. Sixth International Workshop of the Forum for Information Retrieval Evaluation, Bangalore, India, 5–7 December (2014)
5. Fleiss, J.: Measuring nominal scale agreement among many raters. Psychological bulletin. American Psychological Association, 76(5), 378–382 (1971)
6. Flores, E., Barrón-Cedeño, A., Moreno, L., Rosso, P.: Uncovering source code reuse in large-scale academic environments. Comput. Appl. Eng. Educ. doi: 10.1002/cae.21608 (2014) (online)
7. Flores, E., Barrón-Cedeño, A., Rosso, P., Moreno, L.: DeSoCoRe: Detecting Source Code Re-Use across Programming Languages. Proc. 12th Int. Conf. of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, NAACL-2012, 1–4 (2012)
8. Flores, E., Barrón-Cedeño, A., Rosso, P., Moreno, L.: Towards the Detection of Cross-Language Source Code Reuse. Proc. 16th Int. Conf. on Applications of Natural Language to Information Systems, NLDB-2011, Springer-Verlag, LNCS(6716), 250–253 doi: 10.1007/978-3-642-22327-3_31 (2011)
9. Flores, E., Ibarra-Romero, M., Moreno, L., Sidorov, G., Rosso, P.: Modelos de recuperación de información basados en n-gramas aplicados a la reutilización de código fuente. Proc. 3rd Spanish Conf. on Information Retrieval, CERI-2014, 185–188 (2014) (In Spanish)
10. Ganguly, D., Jones, G.: DCU@FIRE-2014: An Information Retrieval Approach for Source Code Plagiarism Detection. In FIRE [4]
11. García-Hernández, R., Lendeneva, Y.: Identification of similar source codes based on longest common substrings. In FIRE [4]
12. Marinescu, D., Baicoianu, A., Dimitriu, S.: Software for Plagiarism Detection in Computer Source Code. Proc. 7th Int. Conf. Virtual Learning, ICVL-2012, 373–379 (2012)
13. Potthast, M., Hagen, M., Beyer, A., Busse, M., Tippmann, M., Rosso, P., Stein, B.: Overview of the 6th International Competition on Plagiarism Detection. In: Cappellato L., Ferro N., Halvey M., Kraaij W. (Eds.) CLEF 2014 Labs and Workshops, Notebook Papers. CEUR-WS.org, 1180, 845–876 (2014)
14. Prechelt, L., Philippsen, M., Malpohl, G.: JPlag: Finding plagiarisms among a set of programs. Tech. Report, Universität Karlsruhe (2000)
15. Ramírez-de-la-Cruz, A., Ramírez-de-la-Rosa, G., Sánchez-Sánchez, C., Luna-Ramírez, W. A., Jiménez-Salazar, H., Rodríguez-Lucatero, C.: UAM@SOCO 2014: Detection of Source Code Reuse by means of combining different types of representations. In FIRE [4]
16. Rosales, F., García, A., Rodríguez, S., Pedraza, J., Méndez, R., Nieto, M.: Detection of plagiarism in programming assignments. IEEE Trans. Educ., 51, 174-183 (2008)
17. Sparck, K., van Rijsbergen, C.: Report on the need for and provision of an "ideal information retrieval test collection. British Library Research and Development Report, 5266, University of Cambridge (1975)

18. Whale, G.: Software metrics and plagiarism detection. Proc. Ninth Australian Computer Science Conf., 231–241 (1990)