On the Detection of SOurce COde Re-use

Enrique Flores*
Universitat Politècnica de València
Spain
eflores@dsic.upv.es

Paolo Rosso
Universitat Politècnica de València
Spain
prosso@dsic.upv.es

Lidia Moreno
Universitat Politècnica de València
Spain
lmoreno@dsic.upv.es

Esaú Villatoro-Tello
Universidad Autónoma Metropolitana
Unidad Cuajimalpa
México D.F.
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 track, 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 five teams submitted 17 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 on the posed shared task.

Categories and Subject Descriptors

A.0 [**General**]: Conference proceedings

General Terms

Measurement, Performance, Experimentation, Languages

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[1]. 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 [4] reveals that 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 [10], 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 [23, 20, 13].

Whereas at PAN@CLEF the shared task addresses plagiarism detection in texts [19], PAN@FIRE focuses on the detection of source codes that have been re-used in a monolingual context, *i.e.,* using the same programming 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.

*Corresponding author.

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

2. RELATED WORK

Although the problem of source code re-use is not new, it is until nowadays that such problem has reached an increasing importance given the rise of material now available in electronic form. Traditionally, the source code re-use detection problem has been approached by means of three main perspectives: *i)* shallow features comparison, *ii)* NLP-related strategies, and *iii)* structural comparison, *i.e.,* graph-based representations.

Shallow techniques involve generating a representation of each source code as a k-tuple, where each component depicts an attribute describing some superficial characteristic from the source code file, for example, number of code lines, number of variables, methods, loops, indentations, commented lines, average length of lines (in characters), etc. Once the representation is obtained, source code re-use is determined by measuring its distance (similarity) against a set of source code files [7, 25] represented by the same set of features. A main drawback of such methodologies is that the number of possible features may be very large and not adequate for all programming languages. Additionally, this type of techniques tend to perform poorly when more sophisticated modifications are performed in source code files.

Regarding the *NLP-related* strategies, the intuitive idea has been to incorporate traditional text similarity measures to the task of source code re-use detection. Such methods, as stated in [21], involve more complex as well as robust approaches. Normally, source code files are treated as text files, hence, common methods such as the traditional *Bag-of-Words*, character n-grams [12, 22], and longest common sub-sequence [2, 15] are among the most popular techniques. One of such work takes into account the "whitespace" indentation patterns of a source code file [2], where a source code document is converted to a pattern, namely whitespace format, replacing any visible character by `X` and any whitespace by `S`, and leaving newlines as they appear. The work reported in [2] then calculates a similarity index based on the longest common substring (LCS) of both patterns. For their experiments, the authors used C sources extracted from the Apache and Linux Kernel. Using a distribution of similarity index for source code pairs, the authors corroborate their hypothesis, namely that similar source code pairs (obtained from different versions of the Linux kernel) exhibit high average LCS based similarity index values, whereas different source codes pairs show low mean similarity index on an average. The work proposed in [20] represents one of the most popular freely-available methods for source code plagiarism identification, and is based on a technique that searches for the longest non-overlapped common substrings between a pair of fingerprints.

Another common approach is to determine the fingerprint of a source code document by making use of the words n-grams[2]. However, this does not consider important characteristics inherent to source codes such as keywords, identifiers names, number of lines, number of terms per line, number of hapax, etc. The work reported in [18] proposes a similarity measure that uses a particular weighting scheme for combining different characteristics of source codes. A major limitation of these reported works is that obtained results are inconclusive because the datasets on which their experiments were conducted do not represent real-life re-use cases and more importantly and not sufficiently large.

In [5] authors performed a detailed analysis of source codes with the help of latent semantic indexing (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 scheme, i.e. given a suspicious source code as a query, they retrieved the most likely candidate original source codes. The dataset was constructed with the help of automated tools namely Moss and Sherlock [16] followed by manual post processing. It was reported that the optimum number of LSA dimensions is 30. A major drawback of the experiments reported in [5] is that these were conducted on small collections of source code files (varying from 106 to 179) which is far from the real-life scenario. A similar work was done in [17] where authors addressed the problem of concept location using an information retrieval model based on LSA. Their proposed approach is employed to find relevant (*semantically related*) parts of source codes.

Finally, the most recent approaches as well as the most complex are the *structure-based* ones. Most of these techniques involve an in-depth representation of source code files. Ideally, such type of representations allow to perform more elaborated comparisons between a pair of source codes, for example, to compare inherent structure features instead of only shallow or lexical features [3, 6, 14].

As an example, the work described in [6] proposes a source code plagiarism detection tool named CCS (code comparison system) that compares source code files by means of using hash values of their abstract syntax trees. Accordingly, CCS performs codes comparison by means of: first traversing the syntax tree and get every hash value of the suspicious and original source codes; then, sub-trees are classified according to the number of child nodes such that only those sub-trees having the same number of child nodes are compared; thus, at the end sub-trees with similar hash values will represent the plagiarised sections within a source code. Similarly, the work proposed by [3] proposed a method for fingerprinting based on syntax trees to retrieve clone clusters of exact matches across a set of software projects. Finally, DeSoCoRe [11] proposes a comparison of two source codes at function-level and looks for highly similar functions or methods in a graphical representation.

Notice that one of the main limitations of previous work is that, all proposed approaches are not comparable among them given that there is no standard source codes collection neither well established metrics suitable for such purposes. In order to overcome such limitations, PAN@FIRE SOCO track defines a standard framework in order to provide a controlled scenario for evaluating source code re-use detection systems. Thus, this paper describes the followed methodology for constructing both training and test sets used during the SOCO competition; and, also a discussion of obtained results is given.

[2] `http://theory.stanford.edu/~aiken/moss/`

3. TASK DESCRIPTION

SOCO shared task focused 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 programming language. Accordingly, participants were provided with a set of source codes written both in C/C++ and Java languages, where source codes have been tagged by language to ease the detection. Thus the task consists in retrieving source code pairs that are likely to be cases of re-use. It is important to mention that this task was performed at document level, hence no specific fragments inside of source codes are expected to be identified; only pairs of source codes files. Therefore, participant systems were asked to annotate several source codes whether or not they represent cases of source code re-use.

SOCO 2014 shared 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, a link to the original source code is provided. It is worth mentioning that the order of each pair was not important[3], *e.g.*, if X has been re-used from Y, it was considered as valid to retrieve the pair X–Y or the pair Y–X. Finally, for the testing phase the only annotation that was provided corresponds to the programming language (C/C++ or Java).

4. 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.

4.1 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 named from 000.java to 258.java. The C collection contains 79 source codes, entitled from 000.c to 078.c. Relevance judgements represent re-use cases in both directions($X \rightarrow Y$ and $Y \rightarrow X$). Table 1 shows the characteristics of the training corpus and the *kappa* (κ) value of the inter-annotator agreement [9].

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 indicates a *substantial* agreement between annotators [9]. Such results indicate (to some extent) that the provided training corpus represents a reliable data set.

[3]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.

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.

Training set (*Inter-annotator agreement*)			
Programming language	Num. of files	#Re-use cases	Value of κ
C	79	26	0.480
JAVA	259	84	0.668

Table 2: Number of source code files contained in the test set, organised by programming language and scenario. The table shows the number of files (# Files), number of identified re-use cases (#Re-used) and the average number of code lines (#Lines) per file within its respective scenario.

C/C++ programming language			
Scenario	#Files	#Re-used	#Lines
A1	5,408	99	63.68
A2	5,195	86	68.11
B1	4,939	86	69.97
B2	3,873	43	80.87
C1	335	8	180.40
C2	145	0	255.21
Total	19,895	322	92.70
Java programming language			
Scenario	#Files	#Re-used	#Lines
A1	3,241	54	99.26
A2	3,093	47	107.87
B1	3,268	73	90.86
B2	2,266	34	102.46
C1	124	0	227.12
C2	88	14	361.00
Total	12,080	222	165.42

4.2 Test Corpus

Provided test corpus was divided by programming language (C/C++ and Java) and by scenarios (*i.e.,* different thematic/problems are considered). This corpus has been extracted from the 2012 edition of Google Code Jam Contest[4]. Each programming language contains 6 monolingual re-use scenarios (A1, A2, B1, B2, C1 and C2). Hence, the name of the files consists of the name of its corresponding scenario and an identifier, for example, file "B10021" belongs to scenario B1 and its identifier number is 0021. An overview of the test dataset characteristics is presented in Table 2. It can be seen that the entire collection of about *20K* and *12K*, for C/C++ and Java respectively, is categorized into six different scenarios. It is important to mention that there is no re-use cases between scenarios, therefore participant systems just needed to look for re-use cases among the source code files inside each scenario. For example, participants did not have to submit a re-use case between files "B10021" and

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

"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. Such distribution of source code files tries to replicate a real life scenario, where a suspicious source code file might have to be compared against a large set of source code files.

In view of the huge size of the test corpus, it was 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 [24]. 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 constructing the relevance assessments, we considered all submitted runs from participant systems with the addition of the two baselines described in the next section. The third column in Table 2 indicates the number of identified relevant judgements for each programming language and scenario. Additionally, Table 2 also depicts the average number of code lines contained in source code files; such value indicates (to some extent) the difficulty (*i.e.*, necessary code lines) of developed programs in order to solve the assigned task.

Finally, it is worth mentioning that two out of the five participant teams submitted their obtained results after the deadline; hence, two different set for the relevance assessments were constructed. Therefore, the *Official relevance assessments* were pooled from the results from the teams `UAEM`,`UAM` and `DCU` plus the two proposed baselines. Whereas the *Unofficial relevance assessments* were formed considering the former configuration plus the results submitted by `Rajat` and `Apoorv` (see Section 7).

5. EVALUATION METRICS

All the participants were asked to submit a detection file with all the source code pairs considered as re-use cases. 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 were considered for the SOCO 2014 task, which are described below:

Baseline 1. Consists of the well-known JPlag tool [20] 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 [12]. 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 preprocessing, 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 well known cosine similarity measure.

It is worth mentioning that for both baselines, a source code pair is considered as a re-use case if the similarity value is higher than 0.95.

6. PARTICIPATION OVERVIEW

In total five teams participated and submitted 17 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. Remaining teams (*i.e.,* `Rajat` and `Apoorv`) submitted their respective runs beyond the allowed official time, nonetheless, their obtained results have been considered for performing the current analysis.

`UAEM` [15] 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-use 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` [22] 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 used 8 similarity values 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` [14] 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 pre-defined 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.

`Rajat` proposed approach is based on a string comparison technique. The proposed method looks for an exact lines match between a pair of source codes files. The used similarity metric consists in calculating a ratio between the number of common lines and the total number of lines of the larger source code file. At the end, the decision process for determining whether or not is a re-use case, is based on the obtained similarity value that must surpass a threshold value of 70%.

`Apoorv` proposes a similar approach to the method described by `Rajat`. Accordingly, the proposed approach considers source code files as text documents, and follows a string matching strategy for measuring similarities. The approach looks for exact lines matching between source code files and estimates a similarity value with the ratio of lines that both source codes have in common. Selected source code pairs as re-use cases are those that obtain the maximum similarity value. At the end, for each source code within the test set, the approach lists a single re-used case with the maximum similarity value.

7. RESULTS AND ANALYSIS

Obtained results by participant systems are shown in Table 3, Table 4, Table 5 and Table 6. Particularly, Table 3 and Table 5 report obtained results for the C/C++ programming language, whereas Table 4 and Table 6 depict results obtained in the Java language. Obtained performance at scenario level (*i.e., thematic* related problems) is shown in Tables 3 and Table 4 for C/C++ and Java programming languages respectively in terms of the F_1 score; whereas that Table 5 and Table 6 report the overall results during the SOCO competition in terms of precision (P), recall (R) and F_1 score.

Notice that the first part of all tables represents the performance obtained when the *official relevance assessments* are considered to evaluate system's performance, whilst the bottom part depicts obtained results when the *unofficial relevance assessments* are considered. As we mentioned before, we ranked obtained results by means of the F_1 measure, given that we prefer systems that are able to obtain (high) balanced values of Precisions (P) and Recall (R).

Observe that, for the C programming language (Table 3) using the *official* relevance assessments, the team from `UAEM` was able to perform well across all the scenarios except C1. Especially for C1, the most complex task according to Table 2, the best result was achieved by the `UAM-C` team. This behaviour indicates, to some extent, that the proposed methodology by `UAEM` is suitable for easy and moderately hard tasks, whilst for more complicated tasks, the approach proposed by `UAM-C` performs better. However, a different

Table 3: F_1 score per scenario considering both the relevant assessments used during the SOCO 2014 track (*i.e.,* Official) as well as the post-competition relevance assessments (*i.e.,* Unofficial). Reported results correspond to the C/C++ programming languages.

Results considering the *Official* relevance assessments						
Team name	Run id	Scenarios (F_1)				
		A1	A2	B1	B2	C1
UAEM	1	**0.382**	**0.372**	**0.587**	**0.531**	0.552
	3	0.321	0.309	0.581	0.521	0.485
UAM-C	1	0.010	0.009	0.024	0.019	0.762
	2	0.008	0.008	0.020	0.007	**0.800**
	3	0.010	0.009	0.024	0.019	0.737
Baseline	1	0.101	0.113	0.245	0.349	0.429
	2	0.294	0.255	0.326	0.323	0.333
Apoorv	1	0.035	0.023	0.022	0.014	0.029
Rajat	1	0.110	0.106	0.236	0.146	0.066
Results considering the *Unofficial* relevance assessments						
Team name	Run id	Scenarios (F_1)				
		A1	A2	B1	B2	C1
UAEM	1	0.125	0.184	0.303	0.250	0.320
	3	0.103	0.150	0.300	0.245	0.276
UAM-C	1	0.003	0.004	0.011	0.007	0.471
	2	0.002	0.003	0.009	0.002	0.500
	3	0.003	0.004	0.011	0.007	**0.533**
Baseline	1	0.125	0.069	**0.386**	**0.486**	0.333
	2	**0.264**	**0.256**	0.340	0.388	0.500
Apoorv	1	0.023	0.015	0.014	0.009	0.024
Rajat	1	0.081	0.115	0.273	0.155	0.068

Table 4: F_1 score per scenario considering both the relevant assessments used during the SOCO 2014 track (*i.e.,* Official) as well as the post-competition relevance assessments (*i.e.,* Unofficial). Reported results correspond to the Java programming languages.

Results considering the *Official* relevance assessments						
Team name	Run id	Scenarios (F_1)				
		A1	A2	B1	B2	C2
UAEM	1	0.514	0.519	0.613	0.523	N/A
	3	0.250	0.234	0.324	0.248	0.500
UAM-C	1	0.474	0.452	0.616	0.447	0.824
	2	0.755	0.058	0.021	0.027	0.111
	3	**0.776**	**0.739**	**0.847**	**0.815**	**1.000**
DCU	1	0.600	0.564	0.652	0.581	0.596
	2	0.701	0.681	0.727	0.673	0.636
	3	0.667	0.676	0.702	0.687	0.667
Baseline	1	0.324	0.388	0.237	0.556	0.824
	2	0.529	0.537	0.559	0.568	0.667
Apoorv	1	0.025	0.021	0.038	0.025	0.275
Rajat	1	0.352	0.337	0.540	0.369	0.718
Results considering the *Unofficial* relevance assessments						
Team name	Run id	Scenarios (F_1)				
		A1	A2	B1	B2	C2
UAEM	1	0.507	0.503	0.613	0.534	N/A
	3	0.246	0.226	0.324	0.255	0.500
UAM-C	1	0.467	0.437	0.616	0.458	0.824
	2	0.746	0.055	0.021	0.028	0.111
	3	**0.797**	**0.769**	**0.847**	**0.829**	**1.000**
DCU	1	0.570	0.534	0.643	0.576	0.596
	2	0.693	0.662	0.727	0.686	0.636
	3	0.658	0.657	0.702	0.700	0.667
Baseline	1	0.301	0.369	0.237	0.545	0.824
	2	0.519	0.496	0.559	0.562	0.667
Apoorv	1	0.027	0.023	0.038	0.026	0.275
Rajat	1	0.384	0.376	0.560	0.404	0.718

behaviour is obtained when the *unofficial* relevance assessments are considered. Notice that for this particular experiment, the best performance across scenarios is obtained by the *Baselines* except for the C1 task, where once again the `UAM-C` got the best performance.

As explained before, both baselines and the `UAEM` methods are based on *NLP-related* strategies for identifying re-use cases. On the contrary, the approach followed by the `UAM-C` considers some features that are not content related (claimed as structural features in [23]). Thus, it seems like NLP-related approaches are good enough for small C/C++ source code files, whereas for more elaborated programs (complex tasks), structural features allow to accurately identify cases of source code re-use.

For the Java scenario, `UAM-C` has achieved the best performance with a balance between precision and recall using both, *official* and *unofficial* relevance assessments. According to the obtained results, the combination of all their proposed similarity (lexical, structural and stylistic) measures using a supervised decision tree has been decisive. However, as can be seen in Table 5 and Table 6 their second run has been affected by the same phenomenon than in the C/C++ scenario: it retrieves more than 10K re-used source code pairs, affecting precision values. For the three runs submitted by `DCU`, a good general performance is obtained, see also Table 6. Particularly for `DCU` second run, the addition of the bag-of-words for selecting nodes from the AST slightly improved the performance of the first run. Whereas that `DCU` third run, where the bag-of-words is not employed for selecting nodes within the AST, it slightly decreases their results.

In general, different approaches were applied to solve the problem of source code re-use detection. As it is possible to notice in Table 5, the two submitted runs from `UAEM` were able to retrieve all the re-used source code pairs (high recall). But their rule introduced for retrieving less obvious re-used cases in run 3 had a negative impact on the perfor-

Table 5: Overall results corresponding to the C/C++ programming languages. Results are reported considering both *official* and *unofficial* relevance assessments.

Results considering the *Official* relevance assessments				
Team name	Run id	Evaluation metrics		
		F_1	P	R
UAEM	1	**0.440**	0.282	1.000
	3	0.387	0.240	1.000
UAM-C	1	0.013	0.006	1.000
	2	0.010	0.005	0.950
	3	0.013	0.006	0.997
Baseline	1	0.190	0.350	0.130
	2	0.295	0.258	0.345
Apoorv	1	0.022	0.011	0.543
Rajat	1	0.129	0.077	0.404

Results considering the *Unofficial* relevance assessments				
Team name	Run id	Evaluation metrics		
		F_1	P	R
UAEM	1	0.196	0.109	1.000
	3	0.169	0.092	1.000
UAM-C	1	0.005	0.002	1.000
	2	0.004	0.002	0.935
	3	0.005	0.002	1.000
Baseline	1	0.238	0.242	0.234
	2	**0.300**	0.193	0.669
Apoorv	1	0.014	0.007	0.903
Rajat	1	0.126	0.068	0.927

mance in terms of precision and, therefore, in F_1. For the `UAM-C` results, these 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 (*keywords*) and taken into account the number of functions according to the C language. As known, C++ language includes new characteristics such as classes and methods and also includes new reserved words (*e.g. cin* or *cout*).

As can be seen in Figures 1 and 2, proposed approaches tend to achieve higher F_1 score when the difficulty of the problem is greater, *i.e.,* source codes files are larger. You might think that easier tasks require less lines of code, thus less evidence of re-use exists within such files. This makes easier tasks a more challenging scenarios to detect re-use of source code.

Accordingly, the best performing system was the string-matching based model [15] for C/C++ programming languages, while the best method for Java was the combination of lexical, structural and stylistic features proposed in [22], in both cases using the *official* relevance assessments. Nevertheless, when considering the *unofficial* relevant assessments, Baseline 2 model, based on the comparison of character 3-grams, obtained the best performing model for C/C++ scenario, meanwhile `UAM-C` approach remains as the best performing for Java.

Table 6: Overall results corresponding to the Java programming languages. Results are reported considering both *official* and *unofficial* relevance assessments.

Results considering the *Official* relevance assessments				
Team name	Run id	Evaluation metrics		
		F_1	P	R
UAEM	1	0.556	0.385	1.000
	3	0.273	0.158	1.000
UAM-C	1	0.517	0.349	1.000
	2	0.037	0.019	0.928
	3	**0.807**	0.691	0.968
DCU	1	0.602	0.432	0.995
	2	0.692	0.530	0.995
	3	0.680	0.515	1.000
Baseline	1	0.380	0.542	0.293
	2	0.556	0.547	0.712
Apoorv	1	0.029	0.015	0.811
Rajat	1	0.418	0.301	0.680

Results considering the *Unofficial* relevance assessments				
Team name	Run id	Evaluation metrics		
		F_1	P	R
UAEM	1	0.552	0.381	1.000
	3	0.271	0.157	1.000
UAM-C	1	0.513	0.345	1.000
	2	0.037	0.019	0.927
	3	**0.821**	0.701	0.991
DCU	1	0.688	0.525	0.995
	2	0.585	0.418	0.973
	3	0.676	0.510	1.000
Baseline	1	0.371	0.525	0.286
	2	0.544	0.445	0.700
Apoorv	1	0.031	0.016	0.855
Rajat	1	0.447	0.321	0.732

8. REMARKS AND FUTURE WORK

In this paper we have presented the results 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 five teams submitted 17 runs. Two teams submitted their runs after the event but have been considered in the

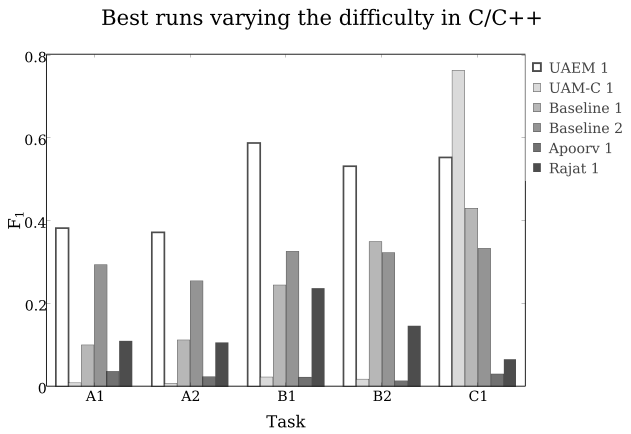

Figure 1: Comparison of the best runs across tasks in C/C++ programming language.

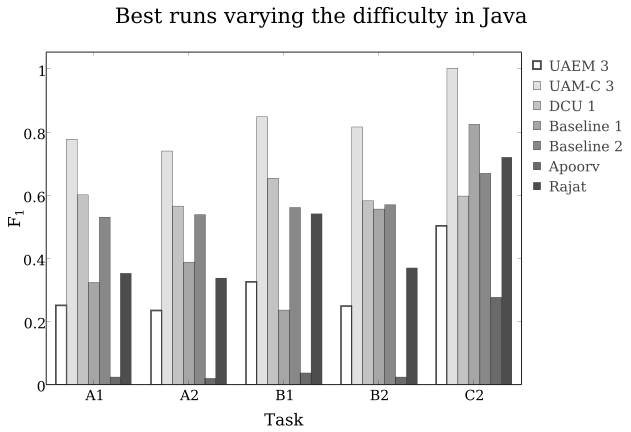

Figure 2: Comparison of the best runs across tasks in Java programming language.

posterior analysis. We summarised the followed approaches by each of the participant systems and presented the evaluation of submitted runs along with their 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 (10 *vs.* 7 runs). We considered two scenarios for the evaluation: *(i)* Calculating the relevance assessments considering just the submitted runs during the track; and *(ii)* Calculating the relevance assessments using all the submitted runs after the competition. `UAM-C` achieved the best results in Java programming language in both scenarios. Nevertheless, the team that achieved the best results in C/C++ in SOCO track was the `UAEM`. Considering the runs submitted after the track, the Baseline 2 achieves the best $F1$ score by means of its character 3-grams 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.

9. 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.

10. REFERENCES

[1] C. Arwin and S. Tahaghoghi. Plagiarism detection across programming languages. *Proceedings of the 29th Australian Computer Science Conference, Australian Computer Society*, 48:277–286, 2006.

[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] M. Chilowicz, E. Duris, and G. Roussel. Syntax tree fingerprinting for source code similarity detection. In *Program Comprehension, 2009. ICPC '09. IEEE 17th International Conference on*, pages 243–247, May 2009.

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

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

[6] B. Cui, J. Li, T. Guo, J. Wang, and D. Ma. Code comparison system based on abstract syntax tree. In *Broadband Network and Multimedia Technology (IC-BNMT), 3rd IEEE International Conference on*, pages 668–673, Oct 2010.

[7] 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.

[8] Fire, editor. *FIRE 2014 Working Notes. Sixth International Workshop of the Forum for Information Retrieval Evaluation, Bangalore, India, 5–7 December*, 2014.

[9] J. L. Fleiss. Measuring nominal scale agreement among many raters. *Psychological bulletin*, 76(5):378, 1971.

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

[11] E. Flores, A. Barrón-Cedeño, P. Rosso, and L. Moreno. DeSoCoRe: Detecting source code re-use across programming languages. In *Proceedings of the 2012 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies: Demonstration Session*, NAACL-HLT, pages 1–4. Association for Computational Linguistics, 2012.

[12] E. Flores, A. Barrón-Cedeño, P. Rosso, and L. Moreno. Towards the Detection of Cross-Language Source Code Reuse. *Proceedings of 16th International Conference on Applications of Natural Language to Information Systems, NLDB-2011, Springer-Verlag, LNCS(6716)*, pages 250–253, 2011.

[13] E. Flores, M. Ibarra-Romero, L. Moreno, G. Sidorov, and P. Rosso. Modelos de recuperación de información basados en n-gramas aplicados a la reutilización de código fuente. In *Proc. 3rd Spanish Conf. on Information Retrieval*, pages 185–188, 2014.

[14] D. Ganguly and G. J. Jones. Dcu@ fire-2014: an information retrieval approach for source code plagiarism detection. In Fire [8].

[15] R. García-Hernández and Y. Lendeneva. Identification of similar source codes based on longest common substrings. In Fire [8].

[16] M. Joy and M. Luck. Plagiarism in programming assignments. *Education, IEEE Transactions on*, 42(2):129–133, May 1999.

[17] A. Marcus, A. Sergeyev, V. Rajlich, and J. Maletic. An information retrieval approach to concept location in source code. In *Reverse Engineering, 2004. Proceedings. 11th Working Conference on*, pages 214–223, Nov 2004.

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

[19] M. Potthast, M. Hagen, A. Beyer, M. Busse, M. Tippmann, P. Rosso, and B. Stein. Overview of the 6th international competition on plagiarism detection. In L. Cappellato, N. Ferro, M. Halvey, and W. Kraaij, editors, *Working Notes for CLEF 2014 Conference, Sheffield, UK, September 15-18, 2014.*, volume 1180 of *CEUR Workshop Proceedings*, pages 845–876. CEUR-WS.org, 2014.

[20] L. Prechelt, G. Malpohl, and M. Philippsen. Finding plagiarisms among a set of programs with JPlag. *Journal of Universal Computer Science*, 8(11):1016–1038, 2002.

[21] I. Rahal and C. Wielga. Source code plagiarism detection using biological string similarity algorithms. *Journal of Information & Knowledge Management*, 13(3), 2014.

[22] A. Ramírez-de-la Cruz, G. Ramírez-de-la Rosa, C. Sánchez-Sánchez, W. A. Luna-Ramírez, H. Jiménez-Salazar, and C. Rodríguez-Lucatero. Uam@soco 2014: Detection of source code reuse by means of combining different types of representations. In Fire [8].

[23] F. Rosales, A. García, S. Rodríguez, J. L. Pedraza, R. Méndez, and M. M. Nieto. Detection of plagiarism in programming assignments. *IEEE Transactions on Education*, 51(2):174–183, 2008.

[24] K. Sparck and C. van Rijsbergen. 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.

[25] G. Whale. Software metrics and plagiarism detection. *Journal of Systems and Software*, 13(2):131–138, 1990.

APPENDIX

A. ADDITIONAL DATA

The following tables show in detail the results obtained by all participant systems at scenario level in terms of F_1, P and R using both *official* and *unofficial* relevance assessments.

Table 7: Results per task considering the official relevance assessments for C/C++ programming languages.

Participant	Run	C														
		A1			A2			B1			B2			C1		
		F1	P	R	F1	P	R	F1	P	R	F1	P	R	F1	P	R
UAEM	1	**0.382**	0.236	1.000	**0.372**	0.229	1.000	**0.587**	0.415	1.000	**0.531**	0.361	1.000	0.552	0.381	1.000
	2	0.321	0.191	1.000	0.309	0.183	1.000	0.581	0.410	1.000	0.521	0.352	1.000	0.485	0.320	1.000
UAM-C	1	0.010	0.005	1.000	0.009	0.005	1.000	0.024	0.012	1.000	0.019	0.009	1.000	0.762	0.615	1.000
	2	0.008	0.004	0.929	0.008	0.004	0.930	0.020	0.010	1.000	0.007	0.003	0.930	**0.800**	0.667	1.000
	3	0.010	0.005	1.000	0.009	0.005	1.000	0.024	0.012	1.000	0.019	0.009	1.000	0.737	0.636	0.875
baseline	1	0.101	0.300	0.061	0.113	0.300	0.070	0.245	0.650	0.151	0.349	0.550	0.256	0.429	0.300	0.750
baseline	2	0.294	0.247	0.364	0.255	0.214	0.314	0.326	0.298	0.360	0.323	0.300	0.349	0.333	0.500	0.250
Apoorv	1	0.035	0.019	0.273	0.023	0.012	0.698	0.022	0.011	0.651	0.014	0.007	0.628	0.029	0.015	0.625
Rajat	1	0.110	0.064	0.394	0.106	0.061	0.407	0.236	0.164	0.419	0.146	0.091	0.372	0.066	0.035	0.500

Table 8: Results per task with the relevant judgements considering all the participants included the post-competition participants in C.

Participant	Run	C														
		A1			A2			B1			B2			C1		
		F1	P	R	F1	P	R	F1	P	R	F1	P	R	F1	P	R
UAEM	1	0.125	0.067	1.000	0.184	0.101	1.000	0.303	0.179	1.000	0.250	0.143	1.000	0.320	0.190	1.000
	2	0.103	0.054	1.000	0.150	0.081	1.000	0.300	0.176	1.000	0.245	0.139	1.000	0.276	0.160	1.000
UAM-C	1	0.003	0.001	1.000	0.004	0.002	1.000	0.011	0.005	1.000	0.007	0.004	1.000	0.471	0.308	1.000
	2	0.002	0.001	0.964	0.003	0.002	0.868	0.009	0.004	1.000	0.002	0.001	0.882	0.500	0.333	1.000
	3	0.003	0.001	1.000	0.004	0.002	1.000	0.011	0.005	1.000	0.007	0.004	1.000	**0.533**	0.364	1.000
baseline	1	0.125	0.150	0.107	0.069	0.100	0.053	**0.386**	0.550	0.297	**0.486**	0.450	0.529	0.333	0.200	1.000
baseline	2	**0.264**	0.158	0.821	**0.256**	0.167	0.553	0.340	0.231	0.649	0.388	0.260	0.765	0.500	0.500	0.500
Apoorv	1	0.023	0.012	0.607	0.015	0.007	1.000	0.014	0.070	0.973	0.009	0.004	1.000	0.024	0.012	1.000
Rajat	1	0.081	0.042	0.929	0.115	0.061	0.921	0.273	0.160	0.946	0.155	0.085	0.882	0.068	0.035	1.000

Table 9: Results per task with the relevant judgements considering only the SOCO participants in Java.

Participant	Run	Java														
		A1			A2			B1			B2			C2		
		F1	P	R	F1	P	R	F1	P	R	F1	P	R	F1	P	R
UAEM	1	0.514	0.346	1.000	0.519	0.351	1.000	0.613	0.442	1.000	0.523	0.354	1.000	0.000	0.000	0.000
	2	0.250	0.143	1.000	0.234	0.133	1.000	0.324	0.193	1.000	0.248	0.142	1.000	0.500	0.333	1.000
UAM-C	1	0.474	0.310	1.000	0.452	0.292	1.000	0.616	0.445	1.000	0.447	0.288	1.000	0.824	0.700	1.000
	2	0.755	0.607	1.000	0.058	0.030	0.957	0.021	0.010	0.973	0.027	0.014	1.000	0.111	0.091	0.143
	3	**0.776**	0.650	0.963	**0.739**	0.611	0.936	**0.847**	0.742	0.986	**0.815**	0.702	0.971	**1.000**	1.000	1.000
DCU	1	0.600	0.429	1.000	0.564	0.397	0.979	0.652	0.483	1.000	0.581	0.410	1.000	0.596	0.424	1.000
	2	0.701	0.540	1.000	0.681	0.516	1.000	0.727	0.576	0.986	0.673	0.507	1.000	0.636	0.467	1.000
	3	0.667	0.500	1.000	0.676	0.511	1.000	0.702	0.541	1.000	0.687	0.523	1.000	0.667	0.500	1.000
baseline	1	0.324	0.600	0.222	0.388	0.650	0.277	0.237	0.550	0.151	0.556	0.750	0.441	0.824	0.700	1.000
baseline	2	0.529	0.439	0.667	0.537	0.434	0.702	0.559	0.472	0.685	0.568	0.463	0.735	0.667	0.500	1.000
Apoorv	1	0.025	0.013	0.759	0.021	0.011	0.702	0.038	0.019	0.863	0.025	0.013	0.853	0.275	0.159	1.000
Rajat	1	0.352	0.241	0.648	0.337	0.232	0.617	0.540	0.425	0.740	0.369	0.275	0.559	0.718	0.560	1.000

Table 10: Results per task with the relevant judgements considering all the participants included the post-competition participants in Java.

Participant	Run	Java														
		A1			A2			B1			B2			C2		
		F1	P	R	F1	P	R	F1	P	R	F1	P	R	F1	P	R
UAEM	1	0.507	0.340	1.000	0.503	0.336	1.000	0.613	0.442	1.000	0.534	0.365	1.000	x	x	x
	2	0.246	0.140	1.000	0.226	0.127	1.000	0.324	0.193	1.000	0.255	0.146	1.000	0.500	0.333	1.000
UAM-C	1	0.467	0.305	1.000	0.437	0.28	1.000	0.616	0.445	1.000	0.458	0.297	1.000	0.824	0.700	1.000
	2	0.746	0.596	1.000	0.055	0.029	0.956	0.021	0.010	0.973	0.028	0.014	1.000	0.111	0.091	0.143
	3	**0.797**	0.662	1.000	**0.769**	0.625	1.000	**0.847**	0.742	0.986	**0.829**	0.723	0.971	**1.000**	1.000	1.000
DCU	1	0.570	0.405	0.962	0.534	0.371	0.956	0.643	0.477	0.986	0.576	0.410	0.971	0.596	0.424	1.000
	2	0.693	0.530	1.000	0.662	0.495	1.000	0.727	0.576	0.986	0.686	0.522	1.000	0.636	0.467	1.000
	3	0.658	0.491	1.000	0.657	0.489	1.000	0.702	0.541	1.000	0.700	0.538	1.000	0.667	0.500	1.000
baseline	1	0.301	0.550	0.208	0.369	0.600	0.267	0.237	0.550	0.151	0.545	0.750	0.429	0.824	0.700	1.000
baseline	2	0.519	0.427	0.660	0.496	0.395	0.667	0.559	0.472	0.685	0.562	0.463	0.714	0.667	0.500	1.000
Apoorv	1	0.027	0.014	0.830	0.023	0.012	0.800	0.038	0.020	0.877	0.026	0.013	0.857	0.275	0.159	1.000
Rajat	1	0.384	0.262	0.717	0.376	0.256	0.711	0.560	0.441	0.767	0.404	0.304	0.600	0.718	0.560	1.000