Uncovering Source Code Reuse in Large-Scale Academic Environments

Enrique Flores[a], Alberto Barrón-Cedeño[b], Lidia Moreno[a], Paolo Rosso[a]

[a]Universitat Politècnica de València, Spain

E-mail: {eflores, lmoreno, prosso}@dsic.upv.es

[b]Talp Research Center, Universitat Politècnica de Catalunya, Spain

E-mail: albarron@lsi.upc.edu

Abstract

The advent of the Internet has caused an increase in content reuse, including source code. The purpose of this research is to uncover potential cases of source code reuse in large-scale environments. A good example is academia, where massive courses are taught to students who must demonstrate that they have acquired the knowledge. The need of detecting content reuse in quasi real-time encourages the development of automatic systems such as the one described in this paper for source code reuse detection.

Our approach is based on the comparison of programs at character level. It is able to find potential cases of reuse across a huge number of assignments. It achieved better results than JPlag, the most used online system to find similarities among multiple sets of source codes. The most common obfuscation operations we found were changes in identifier names, comments and indentation.

Keywords: Source code reuse, plagiarism detection, authoring tools and methods, interactive learning environments, programming and programming languages

1. Introduction

The huge amount of resources available on the Web facilitates the access (and reuse) of diverse content. Automatic detection is necessary to keep track of reuse and especially unauthorised reuse (e.g. plagiarism). Text reuse is defined as "the situation in which pre-existing written material is consciously used again during the creation of a new text or versions" [1]. In recent years, considerable interest in creating better models for uncovering text reuse has been observed[1]. Nevertheless, source code oriented models have been mostly disregarded, except for a few works ([2]; [3]; [4]; [5]). Source code reuse occurs when a programmer borrows (part of) a program "...authored by someone else and, intentionally or unintentionally, fails to acknowledge it adequately, thus submitting it as her own work" [6].

A recent survey by [7] reveals that source code plagiarism represents *30%* of the instances of plagiarism in academia. Around *63%* of college students admit making their work available to classmates. Programmers may obtain source code from their own previous coding, from a classmate, or from a co-worker. As a result, source code reuse detection studies have been mostly applied to closed groups [8]; [5]; [2]. However, resources are also available on the Web. The Web has enabled the rise of new teaching environments.

1 See for instance the International Competition on Plagiarism Detection (Potthast et al., 2013); http://pan.webis.de

Two good examples are Google Code Jam[2] and Coursera[3]. In these examples, a huge number of students need to solve the same problems to certify their acquired knowledge. In Google Code Jam, more than *20K* contestants participated in the 2012 edition. In Coursera, online lectures are crowd-taught. In its Computer Science courses, programming homework is assigned to thousands of students around the globe (e.g. the 2012 lecture on Natural Language Processing was reported to have over *50K* enrolled students). The growth of online students requires to include in academic environments the use of systems to control and improve their learning [**9**]. Other massive programming competitions that involve high school and college students are the International Olympiads in Informatics[4] and ACM International Collegiate Programming Contest[5]. As in a *traditional* academic context, these massive coding environments disapprove the co-operation between participants when solving the proposed tasks. The use of automatic grading tools for programming assignments improves the learning experience of the students and discourage cheating [**10**]. Automatic grading tools combined with a source code reuse detection system should significantly reduce plagiarism among students.

In this paper we approach the problem of identifying cases of source code reuse in a large collection. We apply our recently proposed model for source code reuse detection [**11**], inspired by previous work on natural language text reuse detection [**12**]. We focus on the most used programming languages in the contest: C++, Java, and Python. All in all, we seek to uncover potential cases of borrowing among *34.3K* programs. To the best of our knowledge, this is the first

2 http://code.google.com/codejam/

3 https://www.coursera.org/

4 http://www.ioinformatics.org/

5 http://icpc.baylor.edu/

time that such a large collection has analysed for reuse. Our experiments show numerous instances of collaboration.

The rest of the paper is organised as follows. In Section 2, we overview different approaches to detecting source code reuse. Section 3 describes our model. In Section 4, we describe the corpora we extracted from the Code Jam 2012 program repository. In Section 5, we report our findings on detecting reuse in this corpus and compare our model against JPlag. Finally, in Section 6 we draw conclusions and propose future research.

2. Related Work

Taking advantage of their formality, programming languages allow for a clear typology of potential modifications when reusing complete programs, methods, or just routines. In [13] Faidhi and Robinson identified seven levels of modification applied to reused source code (Table 1 illustrates):

1. Verbatim copy; i.e., copying code snippets without modification.
2. Changes in comments and indentation.
3. Changes in identifiers of variables, methods, and classes.
4. Changes in the position of declarations or addition of extra constants.
5. Methods merging and splitting.
6. Changes in program statements (e.g. *switch* for *if* or *while* instead of *for*).
7. Deep changes in the program decision logic.

Methods for automatic source code reuse detection are divided into two main approaches [14], [15], [16]: attribute-counting and structure-based. The general idea of attribute-counting models is measuring how similar two programs are in terms of the number of elements contained. For instance: number of

unique/distinct operators and operands; number of execution paths; or average nesting depth of the program among others. The structure (logic sequence) of the program is completely ignored. Examples of this approach are [17], [18], [19].This approach performs well when low-level modifications are applied; i.e., Faidhi and Robinson's levels 0 to 3 [16].

We focus more on the second approach: structure-based, which proved to be more successful than the previous approach [3]. This approach considers the entire program contents when looking to identify common segments or assessing an overall level of similarity. The approach mainly consists of converting the source code into a string of tokens and then making a comparison. Structure-based approaches are divided into three categories: flow-based; vocabulary-based; and free vocabulary.

Flow-based approaches represent the structure of the code and only focus on bifurcation statements [20], [16], [21]. This kind of representation is also applied in software maintenance [22]. Software maintenance tries to detect redundant code (*code clones*) to have clearly separated the functionality from the structure of the source code. Its interest consists of finding similar source code fragments that occur more than once [23]. Flow-based is weak against changes in the bifurcation statements, or the combination/division of methods; such as Faidhi and Robinson's levels 3 and 4.

Vocabulary-based approaches represent the code using only certain types of tokens, i.e., reserved words or delimiters. The authors in [4], [8] and [24] show programs submitted by students on the basis of the longest non-overlapping common sequences of reserved words; whereas [5] look at the length of such common sequences. Superficial changes could become more difficult to find for related sequences between two source codes (for example, changing reserved

words, adding extra constants, changing types, merging or splitting methods; Faidhi and Robinson's levels 2-5).

An approach proposed by [2] is shown to be robust against changes in statements and some changes in decision logic (i.e., Faidhi and Robinson's level 5 and 6). By compiling the programs to generate intermediate code, they remove some common changes made by students; such as changes in statements, the addition of useless constants, etc. The resulting code is split into *n*-grams of instruction blocks and these are compared with the Okapi BM25 similarity function [25]. However, this approach is not language-independent, as it needs a common compilation framework that includes the required languages.

Finally, free vocabulary approaches take into account the entire source code, including reserved words, identifiers, and comments. The approach [26] for detecting plagiarism in programming classes involves representing different segments with hashes, which are compared using the Winnowing algorithm (originally proposed for natural language text reuse detection by [27]). Whereas this model is highly efficient, it is weak against editions: modifying one single character in a string dramatically alters its resulting hash value, causing potential cases of reuse to go undetected. However, this approach remains extremely efficient. Our approach belongs to the free vocabulary family and it is described in depth in the next section.

3. Character *n*-Grams for Source Code Reuse

Our detection approach is adapted from models for mono- and cross-language plagiarism detection in natural language texts [28]. The assumption is that programming languages and natural languages can be considered in the same way and both can be treated as an array of characters. The model is designed to

compare all the programs in a large pair-wise collection, making it ideal for searching in scenarios with previously unknown cases of reuse, such as online teaching and contest scenarios. Figure 1 shows a simple representation of its architecture, which is divided into three main modules:

1. Pre-processing. Line feeds, tabs, and spaces are removed and characters case-folded. Symbols duplicated more than six times (e.g. “*******”, “///////”) were cut down to six characters to reduce the noise these symbols introduce in the detection.
2. Term selection and weighting. Code is split into character *n*-grams; i.e., contiguous overlapping sequences of *n* characters. In particular, we use *3*-grams, as we empirically observed that this value produces the best retrieval results [**11**]. Weighting is then computed by simple term frequency (tf) ([**29**], page 117), making it collection independent. The resulting terms are then stored in an inverted index ([**30**], page 109).
3. Similarity estimation. Similarity between codes is estimated using the cosine measure, ranged in [0,1]:

$$\cos(d, d') = \frac{\sum_{t \in d \cap d'} tf_{t,d} tf_{t,d'}}{\sqrt{\sum_{t \in d} tf_{t,d}^2 \sum_{t \in d'} tf_{t,d'}^2}} \quad (1)$$

where *d* and *d*' are two programs in the collection and *t* represents terms; i.e., the character *3*-grams. Pre-processing tries to counteract the vocabulary changes that programmers introduce to conceal reuse. For example, changes to case-sensitive characters in variables, or modifications to the visual structure of the code (by adding line breaks, tabs and spaces), are commonly made to disguise reuse. This kind of change corresponds to the second and third levels of [**13**] (cf. Section 2).

In the feature extraction module, *3*-grams are extracted from the resulting character string. A vector representation of the source code is obtained by weighting the resulting *3*-grams on the basis of normalised *tf.* Spatial information of the *3*-grams is lost, thus preventing the system being sensitive to one of the typical operations to hide reuse: function and operand shifting, first and third level showed in Section 2.

In the document comparison module, all the programs are compared against each other pair-wise, by means of the cosine similarity measure. The output of this module is a similarity ranking that includes every program pair in the collection. The user can then explore the top pairs to uncover potential cases of reuse.

Our decisions are supported in our empirical analysis of a small collection of student assignments following a lecture on multi-agent systems offered in a Computer Science MSc degree [**11**]. We proposed some variants considering parts of the source code to improve the detection of changes made by the programmer. These changes are focused on detecting levels 0 to 4 indicated by [**13**]. We considered the following sections of the original source code for experimentation: (a) source code with comments; (b) source code without comments; and (c) reserved words only. The best results were obtained with (a). We used a character *n*-gram with different *n* values, and found three to be the best option. Therefore, character *3*-grams seem to represent a good characterisation for code in this task, as it is also for natural language texts [**31**].

We compared our approach, which we will refer to from this point as SoCo-C3G, against two models: a sliding-window model and the well-known JPlag tool [**32**]. We tested these models using a collection of *79* C programs from student assignments. These assignments were produced on a course on secure electronic

commerce. This test collection contains *27* reused source code pairs. P-R curves in Figure 2 show that the precision of JPlag starts dropping at very low levels of recall. The maximum F_1 values obtained are *0.64* for SoCo-C3G, *0.4* for JPlag, and *0.55* for the sliding-window model. In terms of AUC, SoCo-C3G clearly outperforms the other two models by achieving a value of *0.70*, versus *0.57* and *0.37* of the others. These results show that SoCo-C3G performs better than a well-known tool such as JPlag. This experiment lead us to consider the same configuration in the current framework.

As this model does not require lexical and structural knowledge about the language, it can be applied to any source code regardless of the programming language used.

4. Corpus

Due to privacy and ethical issues, it is practically impossible to find a freely available corpus of academic programming assignments (e.g. online programming courses) for plagiarism analysis. To the best of our knowledge, one of the few available source code collections that resembles a large-scale corpus of academic programs is Google Code Jam. Therefore, we focused our research work on this massive evaluation platform.

In the *2012* edition of the Google Code Jam contest, *20.6K* contestants from *149* regions participated, working on one of *73* different programming languages. The objective in the contest was to reach the final test stage after a number of rounds. Our analysis focused on the programs submitted in the first round: Qualification. Four problems were proposed in this stage: (A) Speaking in Tongues, (B) Dancing with the Googlers, (C) Recycled Numbers, and (D) Hall of Mirrors. Qualification to further stages requires providing correct solutions to the

proposed problems in the allotted time. A problem can have different levels of difficulty (i.e., larger input data, larger restrictions). Problems B, C, and D have small and large versions. Although it is expected that the same program would solve both problems, the increase in the size of the input data may force re-coding the program to make it more efficient. Participants do not need to solve all the tasks to advance to the next round, but must surpass a minimum quality score.

Problem A had the lowest difficulty of the four. We dismissed it because the data and source code files were included in the same repositories without a clear distinction (something that could introduce noise in the calculation of code similarities). Problems B and C are considered of medium difficulty, whereas problem D is the most difficult. Moreover, problems B-D have two complexity levels: small and large[6].

Figure 3 shows the number of participating codes in each task compared to the number of solved codes. For the easiest tasks (i.e., B_s, B_l, and C_s), most submitted programs were considered adequate; a range of *89%-95%* of success. Conversely, for more complex tasks (i.e., C_l, D_s and D_l) the success rate dropped significantly to *63%-71%.* As expected, the greater the complexity the lower the success rate. This supports the hypothesis that the intention of many contestants is to obtain the minimum score to advance to the next round without trying to solve the most complex tasks.

We focused on the three most used programming languages in the competition: C++, Java, and Python. The total number of programs submitted in these three languages is *34.3K (84.7%* of the overall amount). Some figures are included in

6 The programs collection is freely available at https://code.google.com/codejam/contest/1460488/scoreboard?c=1460488\#vf=1

Table 2. Problems B and C require fewer tokens, whereas the most difficult problem (D) requires the largest number.

5. Experimental Results and Discussion

This section presents two experiments. The aim of experiment 5.1 is to analyse the similarity distributions in three languages. The goal of experiment 5.2 is to detect potential cases of reuse in the competition. In both experiments the programs are exhaustively compared pair-wise; a demanding processing task. For instance, only for problem B more than *34.6* million comparisons of source code pairs were computed.

5.1 Analysis of Similarity Ranges

In this experiment we aim to detect which similarity intervals accumulate among the source code pairs and observe whether these intervals differ for each programming language. Systems like the approach described in Section 3 have to face a large number of source codes that solve the same problem and need to distinguish between high similar parts of the code. These similarity intervals can help establish a threshold for distinguishing between potentially reused and non-reused source code pairs.

Figure 4 shows the similarity distribution of source code pairs for each problem and programming language. For mid-complexity problems, such as B and C, most code pairs in Java have a similarity in the range *0.4-0.6.* For the most complex, problem D, most codes are in the *0.2-0.3* similarity range. The solutions to the easier problems tend to be more similar to each other. Programs written in Python show the same behaviour. The most similar codes are in the range *0.2-0.4* for problems B and C, while for problem D they are in the range *0.1-0.3.* The same behaviour can be appreciated for C++ but to a smaller degree: for

problems B and C most program pairs are in the range *0.2-0.5,* while for problem D they are between *0.2-0.4.*

The trend shows that the accumulation of similar source codes decreases as the complexity of the problem increases. The highest similarity values occur between programs written in Java, followed by C++ and Python. Hence, two codes written in Java may share more common snippets, without necessarily implying reuse. This information is particularly useful to establish decision thresholds between reuse and chance matching for the different programming languages.

5.2 Looking for Instances of Source Code Reuse

The aim of this experiment is to check whether cases of source code reuse occurred in the contest. SoCo-C3G was applied to compute the similarity between *58.6* million pairs from *34.3K* programs distributed by language and task.

After obtaining the similarity-based ranking for each programming language and task, we retrieved the top *20* pairs and presented them to a fluent programmer in the three languages for manual analysis (i.e. *360* instances: *20* pairs x *6* tasks x *3* programming languages). The purpose of such manual analysis was to uncover cases of reuse.

Table 3 shows the number of potentially reused cases we detected. It includes the total number of reused cases by task and programming language (top-*20* pairs). It is worth noting that the harder the task, the fewer the cases of reuse: most reuse cases are in the simpler tasks (B_s, B_l, and C_s). In general, we found that reuse occurrence was lower in Python than in C++ and Java. There was a major difference between tasks C_s and C_l (roughly the same task, but with differing complexity) in Python, which can be explained by the higher computational time requirements of the more complex version. As mentioned previously, we believe

that contestants had to code a more efficient program to accomplish within the time limit for submitting C_I (something that was probably not necessary for the other two languages). We do not discard the possibility that a manual exploration of pairs beyond the top-*20* would uncover more cases of reuse.

We compared the performance of SoCo-C3G on the *6* tasks against JPlag [4]. As JPlag does not support Python, we compare source codes written in C++ and Java only. Table 4 shows the number of potentially reused cases detected by JPlag. SoCo-C3G managed to detect *37* more cases reuse than JPlag.

When source codes are larger (i.e. the harder tasks), both models performed comparably. Larger codes contain more information (e.g. functions, style of programming, etc.) which allow for a better characterisation of the codes. When facing shorter —relatively more similar— codes, JPlag did not manage to distinguish whether two source codes were reused from each other or not. Figure 5 shows the performance of JPlag and SoCo-C3G in terms of number of reused source code pairs detected. It is worth noting that our model detected more cases than JPlag when facing the most difficult (i.e., short) cases to be detected.

We performed a revision of the detected reused pairs in order to classify the levels of source code modifications discussed in Section 2. Table 5 shows the resulting distribution. Note that the reused pairs may belong to more than one level of modification (a frequent occurrence). All the cases include at least one verbatim-copied fragment (level 0). More than *65%* of the pairs contain changes in identifiers (level 2). Changes in comments and indentation occur less often: roughly *50%* (level 1). In this type of scenario, programmers do not tend to comment on their source codes. Moreover, they have to respect an indentation in Python. These reasons cause level 1 to occur less often than level 2.

Higher levels of modification were found less often. On one hand, if they exist, these cases are the hardest to detect, as the similarity between two fragments becomes extremely low. On the other hand, whether they imply relevant reuse instances remains an open question. Differentiating between cases with high-level modifications and independently generated code was nearly impossible, even for a human reviewer.

6. Conclusions and Future Work

Our research work aims at producing source code reuse detection models useful in large-scale programming environments. We analysed the implementations of six different challenges in the three most widely used languages in the contest (C++, Java, and Python) extracted from a Google Code Jam contest. The total number of programs considered was *34.3K*; resulting in *58.6* million pair-wise comparisons.

Our system for source code reuse detection is programming-language independent. SoCo-C3G is based on computing similarities at character level (especially character *3*-grams); an idea borrowed from the analysis of natural language text reuse. The similarities were computed with the well-known cosine measurement.

Our experiments showed an inverse correlation between the complexity of the contest tasks and the similarity between the programs submitted: the simpler the program, the larger the number of common snippets and the instances of reuse. Furthermore, the similarity ranges in the different languages vary: the most similar code pieces are found within the programs written in Java. This evidence can be taken into account to establish a threshold for retrieving good reuse candidates.

For each challenge and programming language we retrieved the *20* most similar program pairs *(360* in total) and gave them to reviewer for manual inspection. The reviewer found *216* cases of potential reuse (i.e. roughly *60*%). Most cases were found in programs written in Java, closely followed by C++.

SoCo-C3G and JPlag performed in a similar way when detecting reused source codes in the most complex tasks. However, our model has achieved better results, especially when the source codes are highly similar and there is a low variability. SoCo-C3G is programming-language independent, an advantage respect to tools like JPlag.

Most fragments in the detected cases were reused by verbatim copying. Among the obfuscated fragments, the most common operations applied by the programmers implied a change in the identifiers names, comments, and indentation. Therefore, the development of new source code reuse detection models, as described in this paper is important in order to detect obfuscation operations and improve assessment in academic environments.

Our future work aims to explore potential cases of reuse across programming languages. This is a challenging problem because of the inherent language differences and the fuzzy frontiers between reuse, inspiration, and coincidental matching between programs in different languages.

Captions

Table 1. Instances of the seven different levels of alteration applied when reusing source code (levels as proposed by [13]).

Table 2. Corpus statistics of the Qualification Round of Google Code Jam 2012.

Table 3. Amount of reused pairs among the top-*20* pairs using our model. Brackets include similarity ranges of the top-*20* pairs.

Table 4. Amount of reuse cases among the top-*20* pairs using JPlag. Brackets include similarity ranges of the top-*20* pairs.

Table 5. Distribution of Faidhi and Robinson modification operations found among the uncovered cases of reuse (percentages).

Figure 1. System architecture: character *n*-grams approach for source code reuse.

Figure 2. Precision-recall curves comparing SoCo-C3G, a sliding-window model and JPlag. In general, SoCo-C3G outperforms the sliding-window and JPlag models.

Figure 3. Amount of programs per task: submitted versus correct. The right-hand-side scale applies for problem D, as participation was significantly lower than for the rest.

Figure 4. Distribution of pair-wise similarity per problem and programming language.

Figure 5. Difference between SoCo-C3G and JPlag in terms of number of detected source code pairs.

References

1. P. Clough, "Measuring text re-use in the news industry," *Copyright and piracy: An interdisciplinary critique*, L. Bently, J. Davis and J. C. Ginsburg (Editors), Cambridge University Press, 2010.
2. C. Arwin and S. Tahaghoghi, "Plagiarism detection across programming languages," *Proc. 29th Australasian Computer Science Conf.* , vol. 48, 2006, pp. 277-286.
3. M. B. Menai and N. S. Al-Hassoun, "Similarity detection in java programming assignments," *Proc. 5th Int. Conf. on Computer Science and Education*, 2010, pp. 356-361.
4. L. Prechelt, G. Malpohl and M. Philippsen, *Finding plagiarisms among a set of programs with jplag*, Journal of Universal Computer Science **8** (2002), no. 11, 1016-1038.
5. 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** (2008), no. 2, 174-183.
6. G. Cosma and M. Joy, *Towards a definition of source-code plagiarism*, IEEE Transactions on Education **51** (2008), no. 2, 195-200.
7. D. Chuda, P. Navrat, B. Kovacova and P. Humay, *The issue of (software) plagiarism: A student view*, IEEE Transactions on Education **55** (2012), no. 1, 22-28.
8. M. J. Wise, *Yap3, improved detection of similarities in computer program and other texts*, ACM SIGCSE Bulletin **28** (1996), no. 1, 130-134.
9. J. Y. Kuo and F. C. Huang, *Code analyzer for an online course management system*, Journal of Systems and Software **83** (2010), no. 12, 2478-2486.
10. D. Spinellis, P. Zaharias and A. Vrechopoulos, *Coping with plagiarism and grading load: Randomized programming assignments and reflective grading*, Computer Applications in Engineering Education **15** (2007), no. 2, 113-123.
11. E. Flores, A. Barrón-Cedeño, P. Rosso and L. Moreno, "Towards the detection of cross-language source code reuse," *Proc. 16th Int. Conf. on Applications of Natural Language to Information Systems, Springer-Verlag, LNCS(6716)*, 2011, pp. 250-253.
12. P. McNamee and J. Mayfield, *Character n-gram tokenization for european language text retrieval*, Information Retrieval **7** (2004), no. 1/2, 73-97.
13. J. A. W. Faidhi and S. K. Robinson, *An empirical approach for detecting program similarity and plagiarism within a university programming environment*, Computers and Education **11** (1987), no. 1, 11-19.
14. P. Clough, "Plagiarism in natural and programming languages: An overview of current tools and technologies," Internal Report CS-00-05, University of Shefield, UK, 2000.
15. G. W. Hislop, *Analyzing existing software for software reuse*, Journal of Systems and Software **54** (1998), no. 3, 203-215.
16. G. Whale, *Software metrics and plagiarism detection*, Journal of Systems and Software **13** (1990), no. 2, 131-138.

17. M. H. Halstead, *Natural laws controlling algorithm structure?*, SIGPLAN Notices **7** (1972), no. 2, 19-26.
18. T. McCabe, *A complexity measure*, IEEE Trans. Soft. Eng. **2** (1976), no. 4, 308-320.
19. W. Harrison and K. Magel, *A complexity measure based on nesting level*, ACM SIGPLAN Notices **16** (1981), no. 3, 63--74.
20. H. T. Jankowitz, *Detecting plagiarism in student pascal programs*, The Computer Journal **31** (1988), no. 1, 1-8.
21. J. Feng, B. Cui and K. Xia, "A code comparison algorithm based on ast for plagiarism detection," *Proc. 4th International Conference on Emerging Intelligent Data and Web Technologies* 2013.
22. R. Koschke, "Survey of research on software clones," *Duplication, redundancy, and similarity in software*, Dagstuhl Seminar Proceedings, 2007.
23. I. D. Baxter, A. Yahin, L. Moura, M. Sant'Anna and L. Bier, "Clone detection using abstract syntax trees," *Proc. of International Conference on Software Maintenance*, 1998, pp. 368-377.
24. A. M. Bejerano, L. E. García and E. E. Zurek, *Detection of source code similitude in academic environments*, Computer Applicattions in Engineering Education (In press).
25. S. E. Robertson and S. Walker, "Okapi/keenbow at trec-8," *Proc. of TREC*, vol. 8, 1999, pp. 151-162.
26. D. Marinescu, A. Baicoianu and S. Dimitriu, "Software for plagiarism detection in computer source code," *Proc. 7th Int. Conf. on Virtual Learning* 2012, pp. 373-379.
27. S. Schleimer, D. S. Wilkerson and A. Aiken, "Winnowing, local algorithms for document fingerprinting," *Proc. of the Int. Conf. on Management of Data*, ACM Press, 2003, pp. 76-85.
28. M. Potthast, A. Barrón-Cedeño, B. Stein and P. Rosso, *Cross-language plagiarism detection*, Language Resources and Evaluation. Special Issue on Plagiarism and Authorship Analysis **45** (2011), no. 1, 45-62.
29. C. D. Manning, P. Raghavan and H. Schutze, "Introduction to information retrieval," Cambridge University Press, 2008.
30. I. H. Witten, A. Moffat and T. C. Bell, *Managing gigabytes: Compressing and indexing documents and images*, Morgan Kaufmann, 1999.
31. E. Stamatatos, "Intrinsic plagiarism detection using character n-gram profiles," *Proc. 3rd Int. Workshop on Uncovering Plagiarism, Authorship, and Social Software Misuse, PAN'09, CEUR Workshop Proceedings*, vol. 502, 2009, pp. 38-46.
32. E. Flores, "Reutilización de código fuente entre lenguajes de programación," *Dept. of Computer Systems and Computation*, Universitat Politècnica de València, MSc dissertation, 2012.