A cross-platform solution for software plagiarism detection

PhD. eng. Cosmin Strilețchi, PhD. eng. Mircea Vaida
Communications Department
Technical University of Cluj-Napoca
Romania

PhD. eng. Ligia Chiorean, eng. Sorin Popa
Communications Department
Technical University of Cluj-Napoca
Romania

Abstract—**The issue of plagiarism is a much discussed topic in the nowadays academic environment. Since the inspiration sources are very vast and accessible, analyzing the originality can become extremely challenging for a human interpreter and therefore an automated solution is very welcomed. This article presents a cross-platformimplementation that measures the similarities found between various source codes that are supposed to solve a certain specific software problem. The testing environment (groups of students) allows the quantified correlations to serve for increasing the awareness about the importance of originality during the code development process.**

Keywords—software plagiarism; source file interpretation; programming language specificity; source files correlation;

I. Introduction

The software development process involves nowadays analyzing numerous sources of inspiration. Any technology should be very well documented and is very likely to be provided with some code snippets that demonstrate the basic usage principles. It is up to each programmer to consider the available information before beginning to write his own original code.

Having access to numerous sources of inspiration(whether they are official or produced by other programmers) broadens the horizons for any newcomer coder and this is a very positive aspect. The downside of being able to access such vast areas of starting points is that inspiration can easily become plagiarism.

The tendency of copying entire code sequences has been demonstrated by the experience accumulated while managing the students from the Technical University of Cluj-Napoca during the software programming laboratories or while working at individual semester and diploma projects. Some students delivered fully functional software applications whose complexity exceeded their knowledge level. They were able to do this by assembling different pieces of pre-existent codes in order to produce the desired result or by copying a working solution while trying to prove it belongs to them (altered comments, changed variable names, etc.).

In this context, it is imperative to draw a line between the coders that possess a certain piece of code claiming they put their own effort into producing it and the rightful creators of that specific software application. Originality and code proprietary imprints needs to be highly valued in the software development process.

There are some other approaches to this matter [1-2]. Each of the existent solutions has specific strengths, amongst which there should be mentioned: online/offline processing, result accuracy, response time, etc. In this context, we tried to develop a new solution that will fit our specific requirements.Our application needed to be able to analyze codes written in various programming languages and collected in an online environment from different platforms. Also, the way our database is organized (very large repositories with thousands of entries) as well as our specific interest in particular aspects manifested in the source codes filesfully justifies our own approach to code plagiarism.

II. Code Originality versus Code Plagiarism

We consider that a certain source code should meet the following requests in order to be considered original when compared to its competitors:
- the names used for the software entities [3] (constants, variables, functions, methods, classes, packages, etc.) should be as unique as possible; it is very less likely for different coders to use the same identifiers for naming all the software entities;
- the algorithmic solution has to have as few common points as possible; even though the concept might be similar, the actual implementation should differ very much from one programmer to another;

The first aspect mentioned above is quite easy to follow (a straight-forward vector match) once the names of the software entities have been properly identified. The algorithm and its writing style are a little more difficult to interpret.

Since all the source codes are stored in plain-text format, the only possibility that a code writer can use in order to prove he is the rightful owner is the file content itself.

A code file can manifest the following content types:
- *comments*; *the header comments* (typically on more than one line) usually depict the code's owner, the date the program was created and the main ideas about the tasks the code is able to resolve; *the body comments* inserted inside the code (on one or more lines) indicate the purpose of the nearby code;
- *actual code*; in this category can be included the prerequisites used by the code (external libraries or files

includes, macro-definitions, etc.) and various code modules (classes, interfaces, functions, etc.).

A code plagiarist can be defined as a person that has the possession of a certain source file created by somebody else and tries to put his own marks on it in order to prove he is the actual owner of the code. The main techniques used for accomplishing this are mentioned below.

T1. The comments can be modified and inserted in various places. This practice is very common amongst the code copiers and is the first step to be done in order to put their "own mark" on a stolen program. Treated as text, different comments will break the similarity chains and are totally irrelevant from the code writing point of view.

T2. The names of the software entities can be modified; its very easy to modify the names of the variables, functions, classes, etc. in order to prove the ownership over a certain piece of code;

T3. Different character or line spacing and altered indentation may also produce false negatives since the similarity chains are divided by this technique. Usually the automated processes fall for this approach.

T4. The code modularity can be altered. The same piece of code can be separated into different software modules. The same implementation can be divided into several functions/methods/classes and therefore the algorithm may seem different. The reverse technique (modules unifying) creates the same false sensation of code originality. Many human analyzers or automated processes fall into this trap and a false negative is created.

III. THE PROPOSED SOLUTION

The source code development process, as used with our students from the Technical University of Cluj-Napoca, consists in the following steps: a certain problem to be solved is made public to its possible coders / the coders deposit the produced source codes in a specific repository / the evaluation software parses all the source codes and computes a series of similarity scores / the programmers receive the feedback about the scores obtained by their own source codes.

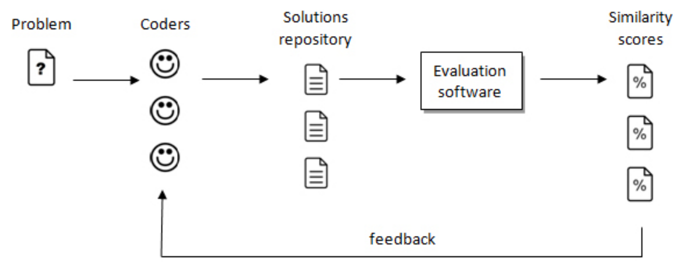

Fig. 1. The evaluation software's role in the source code producing data flow

The entire data flow presented in the image above is implemented as a web distributed application built using the WEB_MATRIX [4] platform.

The coders need to create an account in the application in order to have access to the area from which they can download the problems. Once the proposed tasks are solved, the source codesget uploaded into a server located repository.

The source files consist exclusively of character sequences and therefore the file upload process is controlled in a strict manner so that only text files can be used [5]. File uploading can be easily exploited by malevolent users [6] but limiting the MIME type of the uploaded material and sanitizing the uploaded content [7] resolves this issue.

We will furthermore focus on the implementation of the source code evaluation software.

The evaluation software will have to decide the originality of each source code from the solutions repository, matched against all of its competitors. By parsing the textual input (source files), our application produces various numerical values that quantify the degree of resemblance manifested by each input. The smaller are the obtained matching scores, the more original is considered to be the interpreted code.

A. Input data preprocessing

The preprocessing phase is meant to eliminate all the ballast from the source code file, to normalize the content and to isolate the specific items that are relevant to the actual implementation of the solution proposed by the coder.

Step 1. All the language dependent comments are eliminated. The preprocessing code suppresses all the C/C++, Java, PHP, JavaScript comments. So far, this is the list of programming languages covered by the code analyzer.

Step 2. All the white space characters bigger than 1 blank are eliminated. The new line characters (carriage return and line feed) are also suppressed. False pagination and indentation are no longer a criteria for code shootouts.

By now, all there is left in the source file is pure code. But not all the characters that appear in a source code need to be interpreted in order to highlight the actual written algorithm.

Step 3. All the block code delimiters are taken out. This means that any code sequence marked with '{' or with '}' will lose its separators. They are totally irrelevant for the automated interpretation of the code implementation.

Step 4. All the ; " " and ' ' characters are suppressed. Although the targeted programming languages need the semicolon in order to compile, this character is not needed for the automated evaluation. The same goes for the text sequences and character delimiters (the double and single quotes).

What is left of a source file is a collection of language specific keywords and operators. Also, the file contains the software entities named by the user (constants, variables, etc.). These aspects are very important for detecting the algorithm and also the imprints made by the programmer upon his own code. It can be concluded that the preprocessing phase has served its purpose.

B. Source codes parsing and vectorial representation

It would be much simpler to approach the source codes' content as regular text and to identify the similarities by applying some text comparison algorithms. This way, higher scores will be obtained for longer matched character sequences but the *T4* technique presented in section III suggests that false negatives can be produced.

The parsing phase will emphasize the specificity manifested by the source codes as opposed to regular plain text files. In order to accomplish this, all the operators and keywords that serve as syntax support for a certain programming language need to be loaded as an external resource. Once the language dependent operators (will be referenced as “language tokens”) and keywords have been loaded, a vectorial representation of the source file is created.

A first vector (*v1*) is populated, on each position having each language specific token and the number of its occurrences. The second vector contains all the language tokens (*v2*), extracted from the source code, in the order of their appearance. The third vectorwill store solely the language keywords (*v3*), in their appearance order. Another vector will contain solely the language operators (*v4*), in their appearance order.

During the process described above, all the items discovered in the source file are eliminated. The remaining content is used to populate two vectors of user defined names (*v5* containing constants, variables, functions/methods, classes, interfaces, packages) and values for variables or constants (*v6*);

At this point, the source code is parsed entirely and there is no syntax item left untreated.

C. Vectorial scoring

Once a certain source file and its competitors have been parsed into their vectorial representations the comparison phase can begin.

If a source code is analyzed against *n* competitor files, *n* sets of scores will be obtained.

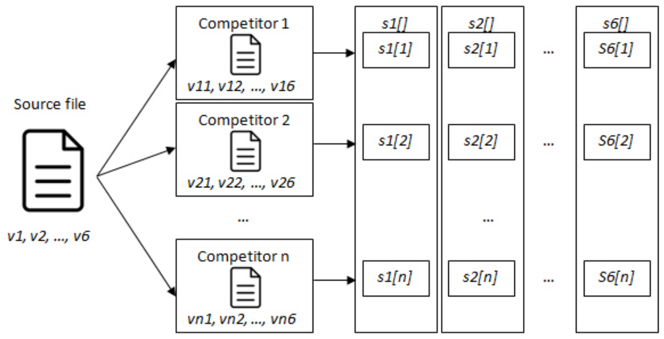

Fig. 2. The scores vectors obtained by analyzing a source file against *n* competitor codes

As a notation, if *v1* represents the first vector associated to the targeted file, *v11…v1n* denote the equivalent vectors associated to the competitor source codes.

Scores 1 (*s1[]*), one-dimensional array. The associated counters of each item in *v1* are matched against the corresponding values in *v11…v1n.*The values of *s1* are initially equalized with the sum of all the counters from *v1* and are decreased with the difference between the corresponding values found in *v11…v1n*.

Scores2 (*s2[]*), one-dimensional array. The values associated to each item located in *v2*are compared to the values in*v21…v2n.*The values from*s2* are initially 0 and are increased with the length of each matched sequence of items.

The comparison method is a variant of sliding window [8]. The original algorithm has been improved in order to mark and count any matching found between the compared vectors. First, the fixed size of the sliding window has been made adaptive and the starting condition is that the number of coincidences should be at least 5. The values from *s2* are greater as the matched vectorial chains are longer. Secondly, *s2*’s values are increased if different chunks of *v2* are to be found in the corresponding *v21…v2n* so the matching algorithm won’t stop at the first occurrence of a match.

Considering the aspects mentioned above, the vector matching algorithm gains robustness against:
- code fragmentation (functions, etc.); the plagiarist divides the original stolen implementation into smaller pieces of code;
- code concatenation; the original code is written in smaller functional blocks and the plagiarist unifies them into bigger code modules;

The minimum width of the sliding window has been established considering the following aspects:
- a value < 5 leads to identifying false positives; a sequence of 2-3 keywords and operators is also insufficient for writing any piece of code;
- a small functional block (for example a function / method or a macro-function) can be written with this amount of language specific tokens;
- a value greater than 5 can skip some chunks of code and this leads to the possibility of obtaining false negatives;

Scores 3(*s3[]*) and *Scores 4* (*s4[]*) are computed in the same manner as *s2*, only that they consider the vectors *v3* and *v4*.

Score 5 (*s5[]*) and *Score 6* (*s6[]*) have their values starting at 0 and are increased each time any single item from *v5*, respectively *v6* is to be found in its competitors *s51…s5n*, respectively *s61…s6n*.

D. Scores reporting and interpretation

Considering the values obtained in the scoring arrays *s1…s6*, a final set of scores *S[]* will be computed for each source file. On each position of the one-dimensional array *S* will be inserted the overall score obtained by weighted summing the individual scores *s1…s6*. Each weight *wk*, $k \in [1, 6]$ is a subunitary value and denotes the importance in the final result of the associated vector.

$$\sum_{k=1}^{6} wk = 1 \tag{1}$$

The weights ordered from the greatest value (importance) to the lowest one have the following positions: *w2*, *w3*,*w4*,*w5*,*w6* and*w1*.

$$S[j] \ = \sum_{k=1}^{6} sj[k] \ * wk, \mathrm{j} \in [0, \mathrm{n}\text{-}1] \tag{2}$$

The overall similarity measure of each file will be obtained by calculating the average value of the the values stored in the *S[]* array.

$$S = \left(\sum_{j=0}^{n-1} S[j]\right) / n \quad (3)$$

The ***scores reporting*** process will produce a textual CSV (Comma Separated Values) output.Each source file will have associated a scoring file that will contain the overall obtained score S and n additional rows, each row containingthe file against which the current similarity score has been calculated, the current similarity score *S[j]* and the individual scores *sj[k]*, $k \in [1, 6]$.

The ***scores interpretation*** is related to the web interface that controls the entire data input and output. The feedback received by each programmer (according to the data flow presented in Fig. 1.) considers a series of thresholds that will trigger different alarms that depend on the individual scores obtained.

IV. Obtained Results

The source files compared were produced by the students in the 1-st and 3-rd year of studies at Electronics, Telecommunications and Information Technology from the Technical University of Cluj-Napoca. The web platform that served for managing the entire data flow between the tutors and the students was built using the WEB_MATRIX framework [4].

Each study year had a total number of 12 weekly activities (laboratories), each one having an average of 7 problems proposed for solving. The solutions produced for a certain laboratory were stored in distinct directories on the server that hosted the entire application.

TABLE I. The Amount of Processed Data

Study year	**Students**	**Laboratories**	**Problems/ laboratory**	**Source files to be processed**
1	400	12	7	33600
3	160	12	7	13440

Each source file was composed of 1300 - 2000 characters grouped in approximately 185 - 285 words averaging 7 characters/word. A full analysis and report for a file that has to be matched against 400 competitor codes usually takes between 0.7 – 1.3 seconds, depending on the code complexity. Our first tests have been performed on an Intel Xeon CPU at 2GHz, 8 GB RAM, 64 bit operating system. This led to the following processing time intervals:
- approx. 2800 seconds (approx. 46 minutes) / laboratory (approx. 552 minutes for all 33600 codes from all laboratories) for students in the1-st year;
- approx. 1120 seconds (approx. 18 minutes) / laboratory (approx. 216 minutes for all 13440 codes from all laboratories) for students in the 3-rd year;

We intend to consider other processing methods using big data concepts in order to improve our systems' response time. Anyway, the processing time is not as important as the accuracy of the interpretation.

V. Conclusions and Future Development

The software tool presented in this paper detects the similarities found between source files written in (but not limited to) C/C++, Java, PHP and JavaScript programming languages. Besides the main purpose of detecting the source code plagiarism, it serves also at increasing the awareness about the importance of own individual work while resolving a certain assignment.

The application is developed and tested for being used during the evaluation process of the individual assignments of the students that learn programming languages at the Technical University of Cluj-Napoca. Using the developed product, the marks that quantify the individual work can now be more realistic since a certain software program developed by a student is not evaluated only by its presence and correctness but also from the similarity point of view.

At this point, the scores obtained by each source code written by each student serve only as an orientation factor (40%) that is used for adjusting the marks associated to the practical activity. The effort invested in assigning a mark for a student belongs mainly to the tutor (professor) that makes use of the integrated software environment for collecting the students' work and the software plagiarism tool for obtaining the similarity scores.

We plan to extend this application by including specific code compilers for each supported programming language. This will give an automatic feedback about the correctness of the code. On numerous occasions we encountered code files uploaded by the students that were either incomplete or with easy-to-spot compiler errors.

Also, the application can be integrated into an intelligent platform able to manage the data from a private academic cloud, so it will propose an automatically calculated mark that will take into account the amount of work done by each student and the similarity scores obtained. The tutor will only have to review and adjust (if necessary) the results produced by the entire platform.

References

[1] https://theory.stanford.edu/~aiken/moss/, accessed on 20.05.2016.

[2] https://www.researchgate.net/publication/220328285_An_Approach_to_Source-Code_Plagiarism_Detection_and_Investigation_Using_Latent_Semantic_Analysis, accessed on 01.06.2016.

[3] http://www.oracle.com/technetwork/java/codeconventions-135099.html#367, accessed on 30.05.2016.

[4] Cosmin Striletchi – "WEB_MATRIX - A secured framework for developing business-to-client web applications", 11th International Conference on Intelligent Engineering Systems, July 2007, Budapest, pp. 1-4

[5] https://www.w3.org/Protocols/rfc1341/7_2_Multipart.html, accessed on 06.03.2016.

[6] Bogdan Calin – "Why File Upload Forms are a Major Security Threat", https://www.acunetix.com/websitesecurity/upload-forms-threat/, accessed on 03.03.2016.

[7] Phillip Tellis – "Keeping Web Users Safe By Sanitizing Input Data", 11.01.2011, https://www.smashingmagazine.com/2011/01/keeping-web-users-safe-by-sanitizing-input-data/, accessed on 15.05.2016

[8] http://algorithmsandme.in/2014/03/heaps-sliding-window-algorithm/, accessed on 07.03.2016