Evaluation of similarity metrics for programming code plagiarism detection method

Vedran Juričić
Department of Information Sciences
Faculty of humanities and social sciences
University of Zagreb
I. Lučića 3, 10000 Zagreb, Croatia
{ vedran.juricic@gmail.com }

Abstract. *This paper shortly presents source code plagiarism detection method based on the low-level language. The similarity or distance metric that is used to calculate similarity coefficient between two source files has great impact on method's performance and results. This paper analyzes precision and recall of four most commonly used metrics, Levenstein distance, Cosine similarity, N-Gram similarity and Greedy String Tilling. Testing is based on various test cases that represent the most frequent code modification techniques.*

Keywords. Plagiarism detection, similarity, source code, similarity metric

1 Code similarity method

This paper presents a method for detecting code similarity for .Net programming languages. Instead of comparing original, source code, method compares projects and source files, based on their Intermediate Language code.

All .Net languages are generally compiled twice before finally executed on the target platform. First compiler is language specific and compiles source code to low-level language called Common Intermediate Language. CIL code is language, processor and platform independent, and in order to execute it, code must be compiled once again, to the target machine code. This is done by using Just-In-Time compiler, which is a component of .Net Framework software package.

Method that analyzes similarity is based on above mentioned CIL code, not the original source code. That way, presented method can be used to compare any language that can be compiled into Common Intermediate Language. In other words, any language that has a .Net compiler can be analyzed and compared. Also, proposed method can be used to compare source files that are written in different .Net languages. For example, it can compare two classes or entire applications, one written in C# and other written in Visual Basic .Net. The only requirement is that source code must not have syntax errors, that is, source code must be able to compile.

In general, all existing source code similarity methods and algorithms have two passes in their comparison. First phase is preprocessing phase, in which comments are removed, letters are converted to lower case, procedure and function blocks are identified, etc. Because some of these are language specific, method must have separate and different first phase for each programming language it supports.

None of this is necessary to perform in proposed method: comments do not appear in compiled assembly, identifiers are reduced to local variables, and function blocks to list of intermediate language instructions.

1.1 Method phases

Method has two phases, compilation and comparison phase. Compilation phase uses appropriate compiler to generate one assembly file for each folder specified in the list of input folders. The language of generated assembly is intermediate language, but its format is not readable and suitable for analysis, so it must be converted into text format. This is done by using disassembler, which loads an assembly containing intermediate language code and generates a text file.

This text file is the actual input file for comparison. It contains certain lines of code that are not relevant for analysis: metadata and module information, comments (generated by disassembler, not developer), stack related data, etc. Those lines are removed and not included in further analysis. Text file containing disassembler code is parsed line by line and lines are verified if they satisfy predefined patterns, so that only important lines are preserved for the second phase. Additionally, every line is transformed, so that it contains only the CIL instruction.

Second phase takes two text files with filtered and parsed lines, and compares them by using one of the existing string comparison methods. Comparison methods are changed so that they can be used with CIL instructions. Instead of analyzing a string (as a sequence of characters), methods are modified so that they analyze a sequence of CIL instructions. The example is shown in the Table 1.

Table 1. Sequence of CIL instructions

A	B	C	B	C	D	E
nop	ldc	stloc	Ldc	stloc	ret	ldarg

Each instruction is taken as one character, so that their sequence form a string, so the string representation of the above example is: ABCBCDE.

By using one of the similarity metrics, method calculates the similarity of two input strings that correspond to the source file similarity. Used metrics are described in the following paragraph.

2 Distance and similarity metrics

Author's initial similarity detection algorithm used Levenstein method for its similarity metrics. Approach deals effectively with single string differences by signaling insertion or deletion. However, algorithm is order preserving, so transformed substrings generate numerous single line differences rather than being seen as a block move

Direct consequence of this limitation is the sensibility to intentional insertions of source code that could be made to conceal original source code and affect plagiarism detection mechanisms. Rearranging blocks of source code, like changing the order of methods and method contents, also have an effect to similarity score.

Proposed method could show better results if it used other similarity measure, that is, distance metric. This chapter describes above mentioned, Levenstein method, and introduces three most common similarity metrics: cosine similarity, n-gram similarity and greedy string tilling.

2.1 Levenstein distance

Levenstein distance is the edit distance function, where the distance is given simply as the minimum edit distance which transforms one string to another. Distance between strings is based on the cost of all transformations necessary to generate one string from another. Possible transformations include copying character from one string to another, deletion, insertion and substitution. Copying transformation actually means that the characters are equal at the observed position, and cost for such transformation is 0. Cost of all other transformations is 1, because characters are not equal at the observed position, so in order to make input strings equal at that position, the character is either deleted, inserted or replaced with suitable character.

Strings are written in form of a table, or matrix, where one string is written horizontally, and other vertically. Cost that is written in a table cell is the minimum cost of its left, top or diagonal neighbors increased by the cost of observed cell transformation.

When cost for each necessary transformation has been identified, distance between two strings is determined as the number in the last row and last column in the matrix.

Table 2. Levenstein distance example

	C	**O**	**D**	**E**
C	0	1	2	3
O	1	0	1	2
D	2	1	0	1
I	3	2	1	1
N	4	3	2	2
G	5	4	3	<u>3</u>

Table above shows the calculation of a Levenstein distance for the example strings, *code* and *coding*. Result distance is three, as it is the number in the 7th row and 5th column. It means that three transformations are necessary to transform string *code* to string *coding*, one substitution and two insertions or deletions.

Similarity between strings is based on the calculated distance, by the following equations:

$$S_1 = 1 - \frac{D}{N_1}$$
$$S_2 = 1 - \frac{D}{N_2} \qquad (1)$$
$$S = Max\ (S_1, S_2)$$

In eq. 1, S_1 is the similarity calculated using Levenstein distance D and the length of the first string, N_1. This is also calculated for similarity S_2 by using length of the second string N_2. Finally, similarity of observed strings is the greater of those two values. In the example given in Table 1, similarity has value of 0.5.

2.2 Cosine similarity

Cosine similarity is a vector based similarity measure, which requires the input string to be transformed into vector space so that the Euclidean cosine rule can be used to determine similarity. The dimension of vector space corresponds to the number of different string elements, that is the number of characters.

First step in forming a vector for observed strings is the identification of possible dimensions. All observed vectors have the equal number of dimensions, and dimension corresponds to the character. If string does not contain a character, its vector has value 0 at the corresponding dimension. If string contains a character, the number of matching characters is associated to the corresponding dimension.

Table 2 shows the example of forming vector for string *code* and *coding*.

Table 3. Cosine similarity example

	C	O	D	E	I	N	G
a	1	1	1	1	0	0	0
b	1	1	1	0	1	1	1

Cosine value for the observed vectors is calculated by dividing their scalar product with norms.

$$\cos(\theta) = \frac{\vec{x}\,\vec{y}}{|\vec{x}||\vec{y}|} \tag{2}$$

Cosine for example shown in Table 2, for strings *code* and *coding*, is 0.61. Although this value can be used as a similarity value, better results are achieved when using normalized vector projection. Using eq. 3, the calculated similarity for string *code* and *coding* is 0.75.

$$\begin{gathered} S_1 = \frac{\cos(\theta)|\vec{y}|}{|\vec{x}|} \\ S_2 = \frac{\cos(\theta)|\vec{x}|}{|\vec{y}|} \\ S = Max\ (S_1, S_2) \end{gathered} \tag{3}$$

2.3 N-Gram similarity

N-Gram similarity method is based on converting the input document or text to sequence of n words (n-grams). It assumes that documents that are written separately have small number of overlapping n-grams. Method has many variants that differ in calculating similarity and forming n-grams. One of the known models based on n-gram similarity is Ferret's model that uses trigrams to calculate similarity.

Following example demonstrates forming trigrams for given two sentences:

This is my first sentence. This is my second sentence.

Table 4. N-Gram similarity example

Trigrams #1	Trigrams #2
This is an	This is an
Is my first	Is my second
My first sentence	My second sentence

The similarity is calculated by dividing the number of n-grams that appear in both documents with average number of formed n-grams.

$$S = \frac{2 \times\ count\ (matching\ ngrams)}{N_1 + N_2} \tag{4}$$

2.4 Greedy String Tilling

Greedy String Tilling is an algorithm that takes two strings and finds the longest match. It is commonly used to compare DNA sequences and protein chains, but has been modified to support comparison of textual data. The algorithm has worst case complexity O(n^3), but with running Karp-Rabin matching has an experimentally derived average complexity close to linear.

Author used simplified variant of this algorithm, because performance will not be taken into evaluation, only the similarity values. The most important definitions and code fragments are presented in the following paragraphs.

A *maximal-match* is where a substring P_p of the pattern string starting at p, matches, element by element, a substring T_t of the text string starting at t. The match is assumed to be as long as possible, i.e. until a non-match or end-of-string are encountered, or until one of the elements is found to be *marked*.

A *tile* is a permanent and unique (one-to-one) association of a substring from P with a matching substring from T. In the process of forming a tile from a maximal-match, tokens of the two substrings are *marked*, and thereby become unavailable for further matches.

A *minimum-match-length* is defined such that maximal-matches (and hence tiles) below this length are ignored. The minimum-match-length can be 1, but in general has greater value. Minimum match length that author used for calculation was 5.

```
search-length s = initial-search-length
repeat
  Lmax = scanpattern(s)
     if Lmax > 2 x s then
       s = Lmax
     else
       markstrings(s)
  if s > 2 x minimum_match_length then
      s = s div 2
  else if s > minimum_match_length then
      s = minimum_match_length
  else stop = true
until stop
```

Above code presents the original algorithm that finds the longest match between two strings. For the purpose of this paper, it has been modified so that it finds all the possible matches that are longer than parameter minimum_match_length. That way it is possible to find all overlaps, and calculate string similarity, by the following equation.

$$S = \frac{\sum length\ (match_i)}{\min(N1, N2)} \qquad (5)$$

Eq. 5. shows that similarity is calculated by dividing the length of all matches with the minimum length of observed matches.

3 Test method and results

The goal of testing is demonstration of algorithms' sensitivity and behavior on different code modification techniques. One of the most popular modification techniques is delocalization, which means reorganization of independent code statements. For example, it is possible to change the order of class members, variable declaration or branch statements.

Unnecessary code insertions comprise statements that do not affect the relevant behavior of a code fragment. It includes insertions of unnecessary variables, class members and other statements, and adding unnecessary method parameters.

Replacing existing code statements with their equivalents does not affect their behavior or work but can have significant impact of visual appearance and some code plagiarism detection techniques. This includes replacing *if-then-else* block with *case* block, replacing for *loop* with *while* loop, changing values of constants and variable types.

Generalization comprises differences in the level of generalization of source code; variable types are replaced (where possible) with more abstract types. For example, replacing *List<int>* with *ArrayList<int>*.

3.1 Test cases and evaluation measures

Author defined 50 test cases that cover all scenarios mentioned in above paragraph, but they are categorized differently, to facilitate and speed up testing.

Variable test category includes tests that check algorithm's behavior when changing the variable names, types, assigned constant values and location of their declaration. Property category tests various property definition styles, changing their type, name and returning values.

Syntax contains small number of test cases, but tests some of the most common variations with variables, including usage of various existing methods for converting one variable type to another. Method contains cases that test changing method name, returning type and various parameter reordering, insertions and deletions.

Loops category contains test cases that check various loop replacements and definitions. Class category includes cases that test changing class name, namespace, and reordering and renaming of class members.

Table 5. Test cases

Category	Test cases
Variable	8
Property	8
Syntax	3
Method	12
Loops	12
Class	7
Total	**50**

Test cases were inspected manually by author, and results of manual comparison were written in a 50x50 matrix whose rows and columns correspond to test cases. Value in every cell is the similarity between two test cases (row represents one test case, and column other test case). Value can be zero, which means that algorithm should not mark observed test cases as equal, and one, which means test cases should be treated as similar or equal. This matrix is a reference matrix; it is used to compare values to those obtained by algorithms, and that is used to evaluate algorithms' behavior.

In order to evaluate algorithms' behavior and sensitivity to different types of code modification techniques, author used common evaluation measures: precision and recall. Precision is also called a measure of exactness or fidelity, and recall is a measure of completeness.

Precision is defined as a fraction of correctly categorized test cases divided by the number of test cases claimed to be similar. Recall is defined as fraction of correctly categorized test cases divided by the number of test cases manually categorized as similar.

Because algorithms can be easily configured so that they show very good precision or good recall, those two measures cannot be considered separately. F measure is defined as a harmonic mean of precision and recall, so that both measures are equally represented. We will analyze algorithms' precision and recall separately, and then, their F measures.

3.2 Results

Analysis was made so that every test case was compared to all other test cases. Similarity values were written in a 50x50 matrix, in a similar way they were written when comparing test cases manually. The only difference is that the cells do not have values zero and one only, but they contain exact similarity values calculated by an algorithm. Values are in the range from zero to one.

In order to compare similarity matrix obtained by algorithm to the reference matrix, values obtained by the algorithm must be converted to zero or one. Zero value means that algorithm has not detected similarity, and value one means that it has. All obtained values above certain threshold are converted to one; otherwise they are converter to zero.

Author tested the relation between threshold value and evaluation measures that it affects. This enabled the author to identify the best values for precision, recall and F measure. Results for each algorithm are presented in the graphs below. Graphs display precision (p), recall (r) and F measure (F) in relation to threshold which is displayed in horizontal axis.

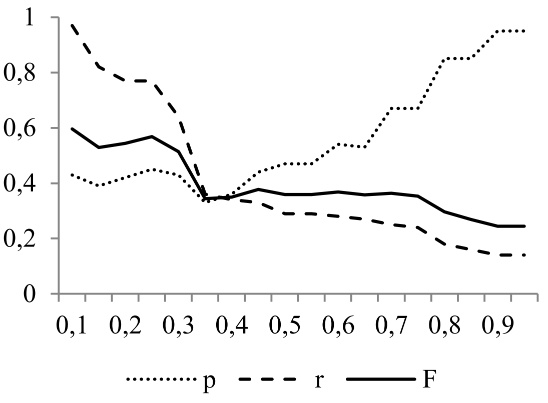

Figure 1. Evaluating Levenstein distance

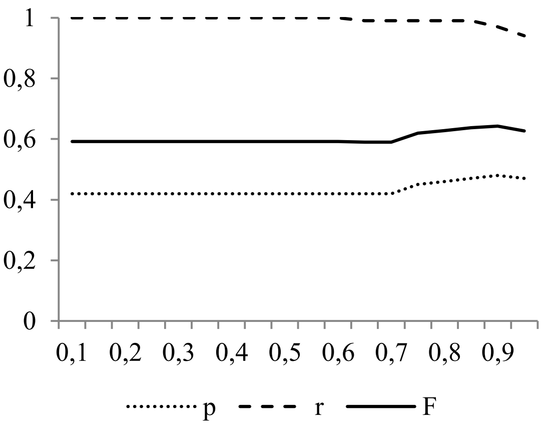

Figure 2. Evaluating Cosine similarity

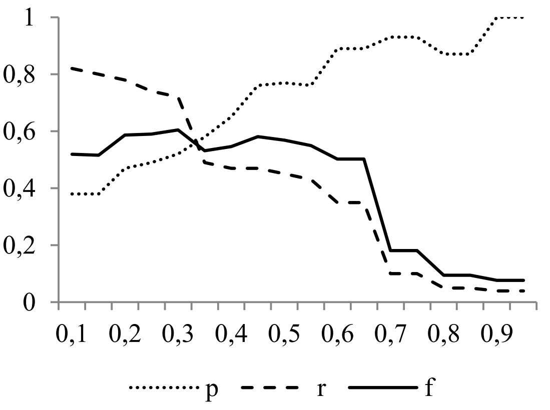

Figure 3. Evaluating N-Gram similarity

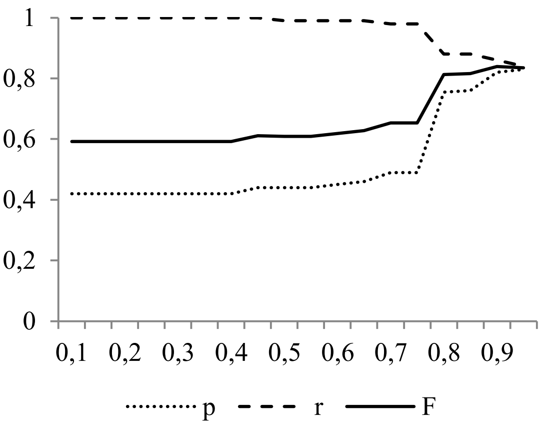

Figure 4. Evaluating Greedy string tilling

As it can be read from graphs, Levenstein distance shows the worst results, it has the lowest F measure of all algorithms regardless of the observed threshold and the recall rapidly falls as the threshold rises. The best value for F measure is 0.59, when threshold is 0.1.

Cosine similarity has very good recall that is greater or equal to 0.9, but has very low precision. Threshold has no effect on precision and recall when lower than 0.7. The best F measure is 0.64, when threshold has value 0.9.

N-Gram similarity has the best precision of all algorithms, reaching 1 when threshold is greater or equal to 0.9. But its recall is very low which results in very low F measure. The best F measure is 0.6 and is reached when threshold is 0.3.

Greedy string tilling shows the best results. It has very high recall, and precision that rises when threshold is greater than 0.8. The best F measure value is 0.84 when threshold is 0.9.

Algorithms' best F measures are shown in fig. 5.

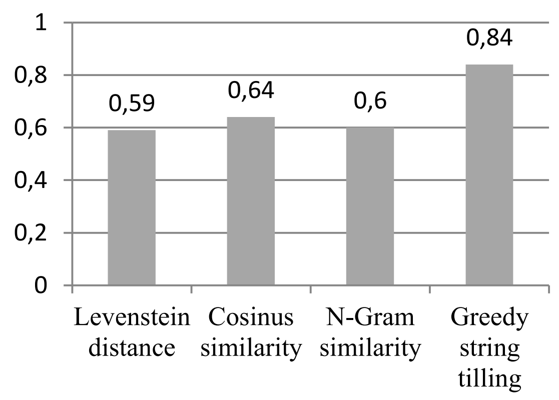

Figure 5. Identified F measures

Levenstein distance and N-Gram similarity are order preserving algorithms and because they are based on the sequences of CIL instructions, have very low recall. As a consequence, both are very sensitive to insertions and rearrangements of code, and are unable to find all the relevant matches.

Cosine similarity is not ordered preserving, and is immune to various types of code transformations, including intentional insertions and rearrangements of code. On the other hand, because the order has absolutely no impact on results and the small number of different CIL instructions that appear in input files, the algorithm has very low precision.

The algorithm that showed the best results is greedy string tilling that combines the best of other observed algorithms. It is ordered preserving, but locally. That means that it searches for longest substring in each pass, but because passes are independent, algorithm is able to find common parts regardless of their location.

4 Conclusion

This paper has presented a method for detecting similarity and potential plagiarisms in programming code by converting it to the low level language. In order to find the most suitable algorithm for calculating similarity, author tested four similarity metrics: Levenstein distance, Cosine similarity, N-gram similarity and Greedy string tilling.

As it is determined by analyzing their behavior on 50 predefined test cases, the best method for similarity detection is Greedy string tilling, as it showed the highest F measure, which means that this algorithm has the best ratio of precision and recall.

References

[1.] Chapman S. SimMetrics, Open source library of Similarity Metrics. 2006. http://staffwww.dcs.shef.ac.uk/people/S.Chapman/stringmetrics.html

[2.] Clough P, Plagiarism in natural and programming languages: an overview of current tools and technologies. 2000. ftp://www.dlsi.ua.es/people /armando/maria/Plagiarism.rtf

[3.] Cohen W.W, Ravikumar P, Fienberg S.E. A Comparison of String Distance Metrics for Name-Matching Tasks. http://www.cs.cmu.edu/ ~wcohen/postscript/ijcai-ws-2003.pdf

[4.] Common Intermediate Language. 17.12.2010. http://en.wikipedia.org/wiki/ Common_Intermediate_Language

[5.] Hage J, Rademaker P, Vugt N. A comparison of plagiarism detection tools. 2010. http://www.cs.uu.nl/ research/techreps/repo/CS-2010/2010-015.pdf

[6.] Juergens E, Deissenboeck F, Hummel B. Code Similarities Beyond Copy & Paste. 2010. https://wwwbroy.in.tum.de/~deissenb/ publications/2010_juergens_beyond_clones.pdf

[7.] Kang N, Gelbukh A, Han S. PPChecker: Plagiarism Pattern Checker in Document Copy Detection. 2006. http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.107.8122&rep=rep1&type=pdf

[8.] Kondrak G. N-gram similarity and distance. 2005. http://citeseerx.ist.psu.edu/viewdoc/ download?doi=10.1.1.67.9369&rep=rep1&type=pdf

[9.] Sembok T, Bakar Z.A. Characteristics and Retrieval Effectiveness of n-gram String Similarity Matching on Malay Documents. 2011. http://www.wseas.us/ elibrary/conferences/2011/ Venice/ACACOS/ACACOS-28.pdf

[10.] Wise M.J. String Similarity via Greedy String Tiling and Running Karp−Rabin Matching. 1993. http://vernix.org/marcel /share/RKR_GST.ps