A Fuzzy R Code Similarity Detection Algorithm

Maciej Bartoszuk[1] and Marek Gagolewski[2,3]

[1] Interdisciplinary PhD Studies Program,
Systems Research Institute, Polish Academy of Sciences
m.bartoszuk@phd.ipipan.waw.pl

[2] Systems Research Institute, Polish Academy of Sciences,
ul. Newelska 6, 01-447 Warsaw, Poland,
gagolews@ibspan.waw.pl

[3] Faculty of Mathematics and Information Science, Warsaw University of Technology,
ul. Koszykowa 75, 00-662 Warsaw, Poland

Abstract. R is a programming language and software environment for performing statistical computations and applying data analysis that increasingly gains popularity among practitioners and scientists. In this paper we present a preliminary version of a system to detect pairs of similar R code blocks among a given set of routines, which bases on a proper aggregation of the output of three different $[0, 1]$-valued (fuzzy) proximity degree estimation algorithms. Its analysis on empirical data indicates that the system may in future be successfully applied in practice in order e.g. to detect plagiarism among students' homework submissions or to perform an analysis of code recycling or code cloning in R's open source packages repositories.

Keywords: R, antiplagiarism detection, code cloning, fuzzy proximity relations, aggregation.

1 Introduction

The R [16] programming language and environment is used by many practitioners and scientists in the fields of statistical computing, data analysis, data mining, machine learning and bioinformatics. One of the notable features of R is the availability of a centralized archive of software packages called CRAN – the Comprehensive R Archive Network. Although it is not the only source of extensions, it is currently the largest one, featuring 5505 packages as of May 3, 2014. This repository is a stock of very interesting data, which may provide a good test bed for modern soft computing, data mining, and aggregation methods. For example, some of its aspects can be examined by using the impact functions aiming to measure the performance of packages' authors, see [6], not only by means of the software quality (indicated e.g. by the number of dependencies between packages or their downloads) but also their creators' productivity.

To perform sensible analyses, we need to cleanse the data set. For example, some experts hypothesize that a considerable number of contributors treat open

source-ness too liberally and do not "cite" the packages providing required facilities, i.e. while developing a package, they sometimes do not state that its code formally depends on some facilities provided by a third-party library. Instead, they just copy-paste the code needed, especially when its size is small. In order to detect such situations, reliable code similarity detection algorithms are needed.

Moreover, it may be observed that R is being more and more eagerly taught at universities. In order to guarantee the high quality of the education processes, automated methods for plagiarism detection, e.g. in students' homework submissions, are of high importance.

The very nature of the R language is quite different from the other ones. Although R's syntax resembles that of C/C++ to some degree, it is a functional language with its own unique features. It may be observed (see Sec. 4) that existing plagiarism detection software, like `MOSS` [1] or `JPlag` [13], fails to identify similarities between R functions' source codes correctly. Thus, the aim of this paper is to present a preliminary version of a tool of interest. It is widely known from machine learning that no single method has perfect performance in every possible case: when dealing with individual heuristic methods one investigates only selected aspects of what he/she thinks plagiarism is in its nature, and does not obtain a "global view" on the subject. Thus, the proposed algorithm bases on a proper aggregation of (currently) three different fuzzy proximity degree estimation procedures (two based on the literature and one is our own proposal) in order to obtain a wider perspective of the data set. Such a synthesis is quite challenging, as different methods may give incomparable estimates that should be calibrated prior to their aggregation. An empirical analysis performed on an exemplary benchmark set indicates that our approach is highly promising and definitely worth further research.

The paper is structured as follows. Sec. 2 describes 3 fuzzy proximity measures which may be used to compare two functions' source codes. Sec. 3 discusses the choice of an aggregation method that shall be used to combine the output of the aforementioned measures. In Sec. 4 we present an empirical study of the algorithm's discrimination performance. Finally, Sec. 5 concludes the paper.

2 Three code similarity measures

Assume we are given a set of n functions' source codes $\mathcal{F} = \{f_1, \ldots, f_n\}$, where f_i is a character string, i.e. $f_i \in \bigcup_{k=1}^{\infty} \Sigma^k$, where Σ is a set of e.g. ASCII-encoded characters. Each f_i should be properly normalized by i.a. removing unnecessary comments and redundant white spaces, as well as by applying the same indentation style. In R, if `f` represents source code (character vector), this may be easily done by calling `f <- deparse(parse(text=f))`.

We are interested in creating a $[0,1]$-valued (fuzzy) proximity relation, cf. e.g. [5], defined by a membership function $\mu : \mathcal{F}^2 \to [0,1]$ such that it is at least:

- reflexive ($\mu(f_i, f_i) = 1$),
- symmetric ($\mu(f_i, f_j) = \mu(f_j, f_i)$).

Intuitively, we have $\mu(f_i, f_j) = 0$ iff f_i, f_j are entirely distinct (according to some criterion), $\mu(f_i, f_j) = 1$ iff they are identical, and immediate values of $\mu(f_i, f_j)$ describe the degree of their "partial proximity". As this is a preliminary study, we omit the discussion on the transitivity (formally, T-transitivity for some t-norm T) of μ and leave it for further research.

Measuring code similarity is always a task that bases on some heuristics. Below we present three different methods μ_1, μ_2, μ_3 to compare two R functions. μ_1 computes the Levenshtein distance [10] between source codes. This is a perfect method to detect verbatim copies, but small modifications (like changing names of variables) should also be easily revealed. Further on, μ_2 bases on our own proposal that takes into account the counts of the number of calls to other functions as well as the names of these functions. This approach is potentially fruitful, as each R statement corresponds to a call to some function. For example:

```
x <- y * z
```

is equivalent to

```
'<-'(x,
     '*'(y, z)
     )
```

and

```
for (i in 1:10)
{ x <- x+i }
```

is in fact

```
'for'(i,
     ':'(1, 10),
     '{'(
         '<-'(x,
              '+'(x, i)
) ) )
```

For instance, a function which calls `sqrt()` twice and `pnorm()` five times is certainly different from the one that calls `readLines()` three times. Last method, μ_3, is based on tokens' analysis, i.e. utilizes the code syntax tree. Here two token sequences are compared by the well-known "greedy string tiling" algorithm, cf. [18].

2.1 Levenshtein distance

The first proximity relation is based on the Levenshtein distance [10] between two character strings. Intuitively, the Levenshtein distance between two sequences is the minimum number of single-character edits (insertions, deletions, substitutions) required to obtain one string from the other. More formally, the Levenshtein distance between two strings a, b is given by $\mathrm{lev}_{a,b}(|a|, |b|)$ where:

$$\mathrm{lev}_{a,b}(i,j) = \begin{cases} \max(i,j) & \text{if} \min(i,j) = 0, \\ \min \begin{cases} \mathrm{lev}_{a,b}(i-1,j) + 1 \\ \mathrm{lev}_{a,b}(i,j-1) + 1 \\ \mathrm{lev}_{a,b}(i-1,j-1) + \mathbf{I}_{(a_i \neq b_j)} \end{cases} & \text{otherwise,} \end{cases}$$

where $\mathbf{I}_{(a_i \neq b_j)} = 1$ iff $a_i \neq b_j$ and 0 otherwise, and $|a|$ is the length of a.

A fast implementation of the Levenshtein distance uses a dynamic programming technique. Here we rely on the `adist()` function in the *utils* package for R.

Please cite this paper as: Bartoszuk M., Gagolewski M., *A fuzzy R code similarity detection algorithm*, In: Laurent A. et al. (Eds.), *Information Processing and Management of Uncertainty in Knowledge-Based Systems*, Part III, (*CCIS* **444**), Springer, 2014, pp. 21–30.

Of course, we get $\mathrm{lev}_{a,b}(i,j) = 0$ if the strings are equal. As we require the proximity degree to be in the interval $[0,1]$, we ought to normalize the result somehow. In our case, we assume:

$$\mu_1(f_i, f_j) = 1 - \frac{\mathrm{lev}_{f_i,f_j}(|f_i|, |f_j|)}{\max(|f_i|, |f_j|)}.$$

It is easily seen that for a pair of identical strings we obtain value of 1. On the other hand, for "abc" and "defghi" we get 0, as $\mathrm{lev}_{\text{“abc”},\text{“defghi”}}(3,6) = 6$.

2.2 Function calls counts

If a function calls some mathematical routines many times and another one rather more often utilizes some data processing methods, we may presume that these two functions are rather not similar. On the other hand, if two functions call `sort()` twice each, then we cannot be certain that there are the same. We see that this heuristic may provide us with a "necessary", but not a "sufficient" proximity estimate. Note that – as it has been stated earlier on – R is a programming language in which function calls play a central role.

Let $\mathcal{R}$ denote the set of names of all possible R functions and $c_i(g)$ be equal to the number of calls of $g \in \mathcal{R}$ within f_i. Our second proximity estimation method is defined by:

$$\mu_2(f_i, f_j) = \frac{\sum_{g\in\mathcal{R}} 2\,(c_i(g) \wedge c_j(g))}{\sum_{g\in\mathcal{R}} (c_i(g) + c_j(g))}.$$

2.3 Longest common token substrings

The idea of the third algorithm has been inspired by the method presented in [14], cf. also [2]. It is quite invulnerable – at least in theory – to such methods used by plagiarists as changing the variable names, swapping two large fragments of code or modifications of numeric constants. It operates in two phases. First, we parse all the functions that are to be compared and convert them into token strings. Transformation $T : \mathcal{F} \to \mathcal{F}'$ gives a token string f_i' calculated from f_i. Then we compare the token strings in pairs to estimate the similarity of each pair. This task is based on finding the longest common token substrings.

Conversion of functions' source codes to token strings. As a rule, a tokenization should be performed in such a way that it depicts the highly abstract "essence" of a function (which is not easy to alter by a plagiarist). In the R language, creation of token strings is very easy: the `getParseData()` function from the *utils* package takes some parsed code as an argument and returns i.a. the information we are looking for. Interestingly, each token is represented internally by an integer number, and a token string is in fact an integer vector.

Example 1. Consider the following source code.

```
1  f <- function(x)
2  {
3     stopifnot(is.numeric(x))
4     y <- sum(x)
5     y
6  }
```

The tokenization procedure results in:

```
1  expr, SYMBOL, expr, LEFT_ASSIGN, expr, FUNCTION, '(',
      SYMBOL_FORMALS, ')', expr,
2  '{',
3  expr, SYMBOL_FUNCTION_CALL, expr, '(', expr,
      SYMBOL_FUNCTION_CALL, expr, '(', SYMBOL, expr, ')', ')'
4  expr, SYMBOL, expr, LEFT_ASSIGN, expr, SYMBOL_FUNCTION_CALL
       expr, '(', SYMBOL, expr, ')',
5  SYMBOL, expr,
6  '}'
```

Comparison of two token strings. The "greedy string tiling" algorithm, as described in [18], is a method used to compare the similarity of two token strings. Interestingly, such a method is also used in `JPlag` [13].

The pseudo-code of the algorithm is presented below. The $\oplus$ operator in line 14 means that we add a match to a matches set if and only if it does not overlap with one of the matches already in this set. The triple match(a, b, l) denotes an association between corresponding substrings of A and B, starting at positions A_a and B_b respectively, of length l.

```
1  Greedy_String_Tiling(String A, String B) {
2     tiles = {};
3     do {
4        maxmatch = MinimumMatchLength;
5        matches = {};
6        Forall unmarked tokens A_a in A {
7           Forall unmarked tokens B_b in B {
8              j=0;
9              while (A_{a+j}==B_{b+j} &&
10                       unmarked(A_{a+j}) &&
11                       unmarked(B_{b+j}))
12                 j++;
13             if (j==maxmatch)
14                 matches = matches ⊕ match(a,b,j);
15             else if (j > maxmatch){
16                 matches = {match(a,b,j)};
17                 maxmatch = j;
18             }
```

```
19                 }
20             }
21             Forall match(a,b,maxmatch) ∈ matches {
22                 for j=0...(maxmatch−1){
23                     mark(A_{a+j});
24                     mark(B_{b+j});
25                 }
26                 tiles = tiles ∪ match(a,b,maxmatch);
27             }
28         } while(maxmatch > MinimumMatchLength);
29         return tiles;
30 }
```

The last task consists of computing a proximity degree of two token strings f_i' and f_j'. In [14], the following formula was proposed:

$$\mu_3(f_i', f_j') = \frac{2\text{coverage}(\textit{tiles})}{|f_i'| + |f_j'|},$$

where

$$\text{coverage}(\textit{tiles}) = \sum_{\text{match}(a,b,\textit{length}) \in \textit{tiles}} \textit{length},$$

and $\textit{tiles} = \text{Greedy_String_Tiling}(f_i', f_j')$.

3 Aggregation and defuzzification of proximity measures

Of course, each of the three proximity relations described above may be represented as a square matrix $M^{(k)}$, where $m_{ij}^{(k)}$ equals to $\mu_k(f_i, f_j)$, $k = 1, 2, 3$. When they are obtained, special attention should be paid to their proper aggregation, and then defuzzification, i.e. projection to $\{0, 1\}$.

The most straightforward approach could consist of determining

$$\mu(f_i, f_j) = A\big(\mu_1(f_i, f_j), \mu_2(f_i, f_j), \mu_3(f_i, f_j)\big)$$

using an aggregation function $A : [0, 1]^3 \to [0, 1]$, i.e. a function at least fulfilling the following conditions: (a) $A(0, 0, 0) = 0$, (b) $A(1, 1, 1) = 1$, (c) A is a nondecreasing in each variable, cf. [7]. For example, one could study the family of weighted arithmetic means, given by 3 free parameters a_1, a_2, a_3:

$$\mu(f_i, f_j) = a_1\mu_1(f_i, f_j) + a_2\mu_2(f_i, f_j) + a_3\mu_3(f_i, f_j),$$

where $a_1 + a_2 + a_3 = 1$ and $a_1, a_2, a_3 \geq 0$. After computing μ, it should be defuzzified in order to obtain a crisp proximity relation P. This may be done by setting $P(f_i, f_j) = \mathbf{I}_{\mu(f_i, f_j) \geq \alpha}$ for some $\alpha \in [0, 1]$.

However, we may note that the values of μ_1, μ_2, and μ_3 are not necessarily comparable – they are not of the same order of magnitude. In other words,

$\mu_1(f_i, f_j) = 0.6$ may not describe the same "objective" proximity degree as $\mu_2(f_i, f_j) = 0.6$. This is illustrated in Fig. 1a: in our exemplary empirical analysis μ_2 seems to give much larger values than the other relations. Thus, their values should be normalized somehow before aggregation. Nevertheless, in our case we will apply a different approach based on logistic regression, which is described in the next section.

4 Experimental results

In order to verify the proposed method, we have created a benchmark set of R functions. The data set consists of students' homework submissions (8 different tasks). A few of them were evidently classified by the tutors as cases of plagiarisms (which was then confirmed by the very students). Moreover, we have added some straightforward modifications of the analyzed routines, e.g. ones with just the variable names changed or those with a more "expanded" forms, like:

```
1 sort_list <- function(x, f) {
2     o <- order(unlist(lapply(x, f)))
3     x[o]
4 }
```

being decomposed to:

```
1 sort_list <- function(x, f) {
2     v1 <- lapply(x, f)
3     v2 <- unlist(v1)
4     o  <- order(v2)
5     x[o]
6 }
```

In result, we obtained a set of 58 functions. 138 pairs, i.e. ca. 4%, are classified a priori as cases of plagiarisms. The whole procedure was computed within just a few seconds on a PC with Intel Core i5-2500K CPU 3.30GHz and 8 GB of RAM (the greedy string tiling was implemented in C++). Notably, similar timings have been obtained by analyzing the same data set with `MOSS` [1] and `JPlag` [13].

Figure 1bcd depicts the relationships between the three proximity measures. The measures are strongly positively correlated. The Spearman ρ is equal to about 0.8 for (μ_1, μ_2) and (μ_1, μ_3) and 0.75 for (μ_2, μ_3). Moreover from Fig. 1a we see that the plagiarism and non-plagiarism case cannot be discriminated so easily with just one measure. The best solution was obtained for α-cut of the form $\mathbf{I}_{\mu_1(f_i,f_j)\geq 0.38}$ (5 false positives (**FP**), i.e. cases in which we classify as plagiarisms non-similar pairs of functions, and 16 false negatives (**FN**), i.e. cases in which we fail to detect plagiarisms), $\mathbf{I}_{\mu_2(f_i,f_j)\geq 0.65}$ (6 and 8, respectively), and $\mathbf{I}_{\mu_3(f_i,f_j)\geq 0.41}$ (15 and 33, respectively). Thus, the intuition behind combining multiple proximity relations seems to be valid.

In order to discriminate between the observations from two classes we may e.g. apply discrimination analysis. A fitted logistic regression model which gives 3 false positives and 7 false negatives is of the form:

$$P(f_i, f_j) = \mathbf{I}\left(\frac{\exp\big(-46.9 + 26\mu_1(f_i, f_j) + 59.1\mu_2(f_i, f_j) - 5.5\mu_3(f_i, f_j)\big)}{1 + \exp\big(-46.9 + 26\mu_1(f_i, f_j) + 59.1\mu_2(f_i, f_j) - 5.5\mu_3(f_i, f_j)\big)} \geq 0.5\right).$$

The performance of our method was compared with `JPlag` and `MOSS`, see Table 1 for a summary. As none of them has built-in support for R syntax, we analyzed two versions of our benchmark set: one "as is" and the other one normalized with `deparse()` and `parse()`. Moreover, we tried to determine which language (`-l`) flag leads to best result in case of `MOSS`. Quite interestingly, the functional language Haskell gave the smallest number of incorrect hits. On the other hand, even though `JPlag` supports Java, C/C++, Scheme, and C#, it treated R sources as plain text (note that μ_3, however, uses a `JPlag`-like approach). In this case, the user may decide which similarity level pair of function should be displayed. Here we have chosen a very liberal 10% level and a more conservative 40% level.

5 Conclusions

The preliminary version of our plagiarism detection system seems to be quite fast and accurate. As far as the analyzed data set is concerned, it correctly classified most of the suspicious similarities. Moreover, the system is – at least theoretically – not vulnerable to typical attacks, like changing names of variables, swapping code places and function composition/decomposition.

Of course, there is still much to be done to make our algorithm more reliable. First of all, we should investigate some kind of transitivity notion in our proximity relation. This is needed to detect fuzzy cliques of functions (e.g. in order to detect groups of plagiarists). Moreover, we could think of a more valid normalization of the relations' values or applying different modern data mining techniques for data classification.

Even more, there are still many other well-known groups of methods which can be aggregated. For example, a plagiarism-detection method based on the Program Dependency Graph (PDG) [4, 11, 15] represents the so-called control and data dependencies between the statements in a program. This approach aims to be immune to changing variables' names, swapping small fragments of code, inserting or deleting single lines of code and substituting `while` loops for their equivalents.

There is still an open question if the Levenshtein distance is the most proper plain text metric. There are many other measures, such as the Damerau-Levenshtein distance [3], Hamming distance [8], Jaro-Winkler distance [17], Lee distance [9] which should be examined, see also [12] for a survey on the topic.

Lastly, we should collect and analyze a more extensive data set of R functions, so that our method could be calibrated more properly, perhaps even with a form of human feedback.

Acknowledgments. M. Bartoszuk acknowledges the support by the European Union from resources of the European Social Fund, Project "Information technologies: Research and their interdisciplinary applications", agreement UDA-POKL.04.01.01-00-051/10-00 via the Interdisciplinary PhD Studies Program.

References

1. Aiken, A.: `MOSS` (Measure of software similarity) plagiarism detection system. http://theory.stanford.edu/∼aiken/moss/
2. Chilowicz, M., Duris, E., Roussel, G.: Viewing functions as token sequences to highlight similarities in source code. Science of Computer Programming 78, 1871–1891 (2013)
3. Damerau, F.J.: A technique for computer detection and correction of spelling errors. Communications of the ACM 7(3), 171–176 (1964)
4. Ferrante, J., Ottenstein, K.J., Warren, J.D.: The program dependence graph and its use in optimization. ACM Trans. Program Lang. Syst. 9(3), 319–349 (1987)
5. Fodor, J., Roubens, M.: Fuzzy Preference Modelling and Multicriteria Decision Support. Springer-Verlag (1994)
6. Gagolewski, M., Grzegorzewski, P.: Possibilistic analysis of arity-monotonic aggregation operators and its relation to bibliometric impact assessment of individuals. International Journal of Approximate Reasoning 52(9), 1312–1324 (2011)
7. Grabisch, M., Marichal, J.L., Mesiar, R., Pap, E.: Aggregation functions. Cambridge University Press (2009)
8. Hamming, R.W.: Error detecting and error correcting codes. Bell System Technical Journal 29(2), 147–160 (1950)
9. Lee, C.Y.: Some properties of nonbinary error-correcting codes. IRE Transactions on Information Theory 4(2), 77–82 (1958)
10. Levenshtein, I.: Binary codes capable of correcting deletions, insertions, and reversals. Soviet Physics Doklady 10(8), 707–710 (1966)
11. Liu, Ch., Chen, C., Han, J., Yu, P.S.: `GPLAG`: Detection of Software Plagiarism by Program Dependence Graph Analysis. In: Proc. 12th ACM SIGKDD Intl. Conf. Knowledge Discovery and Data Mining (KDD'06), 872–881 (2006)
12. Navarro, G.: A guided tour to approximate string matching. ACM Computing Surveys 33(1), 3188 (2001)
13. Prechelt, L., Malpohl, G., Philippsen, M.: Finding plagiarisms among a set of programs with `JPlag`. Journal of Universal Computer Science 8(11), 1016–1038 (2002)
14. Prechelt, L., Malpohl, G., Phlippsen, M.: `JPlag`: Finding plagiarisms among a set of programs. Tech. rep. (2000)
15. Qu, W., Jia, Y., Jiang, M.: Pattern mining of cloned codes in software systems. Information Sciences 259 (2014), 544–554
16. R Core Team: R: A Language and Environment for Statistical Computing. R Foundation for Statistical Computing, Vienna, Austria (2014), http://www.R-project.org/
17. Winkler, W.E.: String Comparator Metrics and Enhanced Decision Rules in the Fellegi-Sunter Model of Record Linkage. Proc. Section on Survey Research Methods (ASA): 354–359 (1990)
18. Wise, M.J.: String similarity via greedy string tiling and running Karp-Rabin matching. Tech. rep., Dept. of Computer Science, University of Sydney (1993)

Please cite this paper as: Bartoszuk M., Gagolewski M., *A fuzzy R code similarity detection algorithm*, In: Laurent A. et al. (Eds.), *Information Processing and Management of Uncertainty in Knowledge-Based Systems*, Part III, (*CCIS* **444**), Springer, 2014, pp. 21–30.

Table 1. Performance of the systems considered.

	FP	**FN**	**t [s]**
Proposed approach	3	7	3
`JPlag` - > 0.1 - plain text - original code	39	55	≈ 10
`JPlag` - > 0.1 - plain text - normalized code	63	57	≈ 10
`JPlag` - > 0.4 - plain text - original code	0	107	≈ 10
`JPlag` - > 0.4 - plain text - normalized code	0	110	≈ 10
`MOSS` - plain text - normalized code	31	89	≈ 3
`MOSS` - plain text - original code	29	84	≈ 3
`MOSS` - C - original code	1	94	≈ 3
`MOSS` - C - normalized code	1	94	≈ 3
`MOSS` - Haskell - normalized code	22	66	≈ 3
`MOSS` - Haskell - original code	14	72	≈ 3

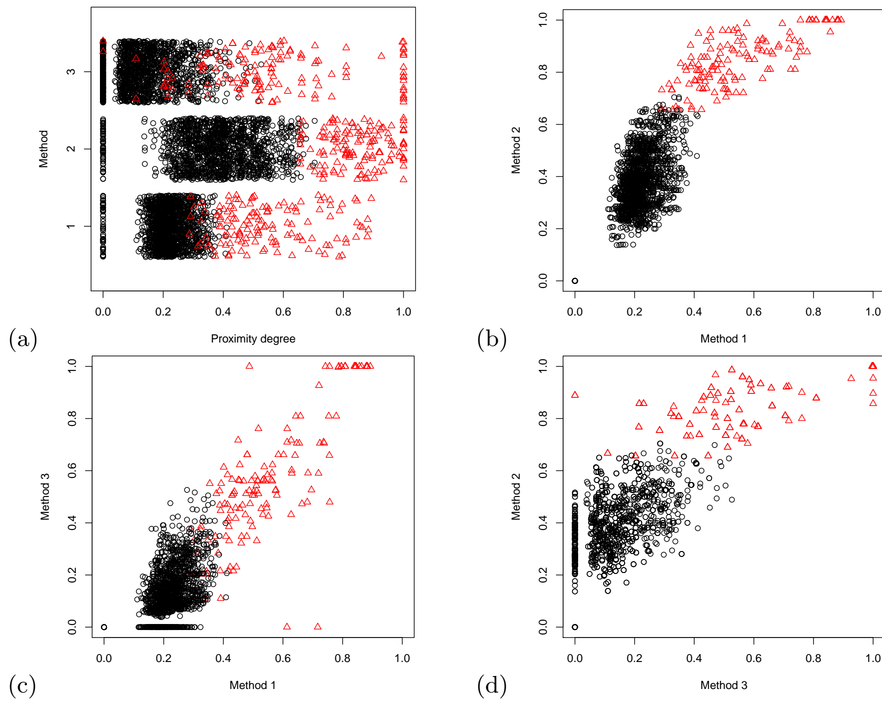

Fig. 1. Values of proximity measures μ_1, μ_2, μ_3 for non-similar pairs (circles) and plagiarisms (triangles).