T-norms or T-conorms? How to Aggregate Similarity Degrees for Plagiarism Detection

Maciej Bartoszuk[a,*], Marek Gagolewski[b,c]

[a] *Warsaw University of Technology, Faculty of Mathematics and Information Science, ul. Koszykowa 75, 00-662 Warsaw, Poland*
[b] *Deakin University, School of Information Technology, Geelong, VIC 3220, Australia*
[c] *Systems Research Institute, Polish Academy of Sciences, ul. Newelska 6, 01-447 Warsaw, Poland*

Abstract

Making correct decisions as to whether code chunks should be considered similar becomes increasingly important in software design and education and not only can improve the quality of computer programs, but also help assure the integrity of student assessments. In this paper we test numerous source code similarity detection tools on pairs of code fragments written in the data science-oriented functional programming language R. Contrary to mainstream approaches, instead of considering symmetric measures of "how much code chunks A and B are similar to each other", we propose and study the nonsymmetric degrees of inclusion "to what extent A is a subset of B" and "to what degree B is included in A". Overall, t-norms yield better precision (how many suspicious pairs are actually similar), t-conorms maximise recall (how many similar pairs are successfully retrieved), and custom aggregation functions fitted to training data provide a good balance between the two. Also, we find that program dependence graph-based methods tend to outperform those relying on normalised source code text, tokens, and names of functions invoked.

Keywords: Fuzzy logic connectives, Similarity aggregation, Decision making, Data-driven optimisation, R language

1. Introduction

With the growth in complexity and volume of source code repositories, there comes the pressing necessity of their automatic processing [25, 37, 40, 46, 47]. Amongst the research topics involving such types of activities, the

*Corresponding author; email: maciej.bartoszuk@mini.pw.edu.pl detection of similar code fragments (code clones) is particularly prevalent. If we had a reliable procedure of this kind at hand, we would be able to indicate duplicated lines that should be turned into stand-alone functions or methods. Moreover, whenever honesty and integrity of student assignments is to be assured, a teacher could be supported in the arduous process of identifying potential instances of plagiarism.

The number of publications in the aforementioned area is vast. Many approaches have been systematically reviewed in, amongst others, [3, 60, 65]. Most importantly, four main method classes can be distinguished, see also Section 3.

- *Textual* – source code is barely transformed. Typical algorithms [28, 41, 53] in this family use hash functions or string metrics to compare fixed-sized fragments of source codes.
- *Lexical* – lexical analysis performed by a language interpreter/compiler yields a token-based representation of a code chunk. Such a form is believed to be more robust against, e.g., variable renaming. Many popular methods are based on tokens, they mainly differ in ways of finding common token subsequences, see [29, 48, 57].
- *Syntactic* – an abstract syntax tree (AST) is generated. Then, such graphs can be compared directly in a pairwise manner [10, 16] or indirectly, based on some graph metrics [14, 18].
- *Semantic* – static analysis of a program is performed so as to obtain a so-called program dependence graph (PDG), which itself is an extension of an AST. This representation is invariant to reordering of independent code lines. Main approaches to comparing PDGs utilise (sub)graph isomorphism algorithms, such as VF2 [32, 39] or SimilaR [9] (which we discuss below).

Here we shall consider the free (libre) and open source R language [15, 50] ("GNU S"), which is widely used for statistical computing and modelling, data wrangling and analysis as well as machine learning. Notably, the main repository of R extensions, CRAN (*The Comprehensive R Archive Network*), featured ca. 17,000 packages in 2020. Despite its popularity, there has been no comparison of the performance of antiplagiarism tools for the R language. Thus, the first aim of this paper is to answer the question regarding which code representation format (e.g., raw source text, sequences of invoked function names, tokens, PDGs) is the most suitable in this setting. We provide a systematic and methodologically consistent (as per the validation framework discussed in Sections 2.4 and 2.5) comparison between many existing algorithms (upon which systems such as JPlag [48], MOSS [58], GPLAG [39] are build; see Table 1) translated to the R case.

However, we will go beyond the simple comparison of standalone code clone detection methods where a measure of similarity of two code chunks is typically chosen ad hoc and assumed to be symmetric: the degree to which A is similar to B is equal to that to which B is similar to A. It is because our preliminary studies [6–8] suggest that it would be better if a similarity measure was regarded as an inclusion-like relation: how much A is a subset of B and to what extent B is included in A. Then, the two independent values can be carefully aggregated to obtain a single number.

Thus, the second aim of this paper is to propose a comprehensive methodology for identifying and assessing the best way to aggregate the nonsymmetric similarity degrees so as to obtain the best performing code clone detection method, e.g, in terms of maximising precision, recall, and F-measure. We study a broad array of aggregation functions (see [11, 12, 19] and Sections 2.3 and 4.3 for an overview) known from decision making, fuzzy logic, and soft computing, including t-norms, t-conorms, and means.

The remainder of this paper is set out as follows. In Section 2 we introduce the notion of nonsymmetric similarity measures in the context of plagiarism detection as well as different ways in which we can aggregate them into a single number. Then we detail our custom framework that we have employed for the benchmarking of R code clone detection algorithms described in Section 3. Experiment results are given in Section 4. We not only indicate the best-performing aggregation functions chosen amongst a predefined set of operators, but also learn custom ones from data using the notion of B-spline surfaces. We conclude the paper in Section 5.

2. Methodology

2.1. Types of clones

Regardless of the programming language chosen, before studying the similarity detection methods, we need to address a few key issues. First and foremost, we should define what we mean when we say that two code fragments are similar to each other. At first glance, this seems trivial, as most users have an intuitive idea of what plagiarism is. However, in order to be able to *quantify* the degree of similarity, we need to formalise the concepts of concern. One extreme would be to recognise as similar only the chunks that are verbatim copies of each other. On the other end of the spectrum, we could treat two functions as identical whenever they yield the same outputs for each set of arguments, no matter how they are implemented – even if the complexities of the underlying algorithms are entirely different. In particular, [13, 63] distinguish between the four following types of clones:

1. Type-1: Syntactically identical code fragments, up to the differences in white spaces, layout, and comments.

2. Type-2: Syntactically identical code fragments, except for the differences in identifier names, literal values, white spaces, layout, and comments.

3. Type-3: Syntactically similar code fragments that differ at the statement level. Fragments have statements added, modified and/or removed with respect to each other.

4. Type-4: Syntactically dissimilar code fragments that implement the same functionality.

2.2. Symmetric and nonsymmetric similarity measures

Let us denote with $X = \{f_1, f_2, \dots, f_n\}$ the set of functions to compare against each other. Typical approaches to code clone detection can be regarded as sophisticated ways to introduce pairwise similarity measures on X. Nevertheless, there is an implicit assumption that if a code chunk $f_i \in X$ is similar to $f_j \in X$, then a code chunk f_j is similar to f_i to the same extent. It means that all methods can be regarded as functions like $\mu\colon X \times X \to [0, 1]$, such that:

- $\forall_{f_i \in X} \quad \mu(f_i, f_i) = 1$ (upper bound),
- $\forall_{f_i, f_j \in X} \quad \mu(f_i, f_j) = \mu(f_j, f_i)$ (symmetry).

Here, “0” means “totally dissimilar” and “1” denotes “very similar” or even “identical”, with various in-between states possible.

In this paper we proclaim a different approach: to consider a measure of inclusion of one code chunk within the other. This is a nonsymmetric setting where we assume that one fragment might be more similar to another.

As a motivation, let us consider the following example:

Code chunk f_i:

```
s = 0
for (i in x) {s = s + i}
```

Code chunk f_j:

```
s = 0
for (i in x) {s = s + i}
m = 1
for (i in x) {m = m * i}
```

It seems reasonable to yearn for a similarity measure $\tilde{\mu}$ where, say, $\tilde{\mu}(f_i, f_j) = 1$ and $\tilde{\mu}(f_j, f_i) \geq 0.5$. This would enable us to detect the situation where somebody makes a verbatim copy of a fragment of a long and complex function.

All the algorithms considered in this paper will be introduced below in two versions: the symmetric (μ) and the nonsymmetric ($\tilde{\mu}$) one. The nonsymmetric approach gives us one more degree of freedom and hence is more informative. Yet, there are many situations where there is still a need to consider just a single value. This is why in the next subsection we shall review a wide range of binary aggregation functions, including t-norms and t-conorms.

2.3. Aggregating similarity measures

Let f_i, f_j be two source code chunks that we wish to compare. The nonsymmetric similarity degrees that each algorithm outputs, $\tilde{\mu}(f_i, f_j), \tilde{\mu}(f_j, f_i) \in [0, 1]$ need to be combined so as to obtain a single value from the unit interval, i.e., $\hat{\mu}(f_i, f_j) = A(\tilde{\mu}(f_i, f_j), \tilde{\mu}(f_j, f_i))$ for some function $A : [0, 1] \times [0, 1] \to [0, 1]$. It is reasonable to assume that it holds (at least) that:

- $A(0, 0) = 0$, $A(1, 1) = 1$ (boundary conditions),
- $A(x, y) \leq A(x', y')$ for $x \leq x'$ and $y \leq y'$ (monotonicity),
- $A(x, y) = A(y, x)$ (commutativity/symmetry).

We call such a function an (symmetric) aggregation operator (see, e.g., [19]). In this paper we shall consider the following functions, which are the most ubiquitous in decision making and knowledge-based systems [11]:

- $T_m(x, y) = \min(x, y)$ (minimum),
- $T_p(x, y) = xy$ (product),
- $T_Ł(x, y) = \max(0, x + y - 1)$ (Łukasiewicz's t-norm),
- $S_m(x, y) = \max(x, y)$ (maximum; t-conorm dual to T_m),
- $S_p(x, y) = 1 - (1 - x)(1 - y)$ (t-conorm dual to T_p),
- $S_Ł(x, y) = \min(1, x + y)$ (t-conorm dual to $T_Ł$),

- $M(x, y) = \frac{x+y}{2}$ (arithmetic mean),
- $\varphi_{\mathbf{V}}(x, y)$ (a fitted B-spline surface with a control point matrix $\mathbf{V}$).

The first three functions are examples of *t-norms* (triangular norms, see, e.g., [31]). A t-norm is a function $T\colon [0,1]\times[0,1] \to [0,1]$, which can serve as a fuzzy logic "AND" operator, i.e., it has 1 as the identity element: $T(x,1) = x$ for all x.

The next three functions are *t-conorms*, i.e., fuzzy logic "OR" operators having 0 as the identity element.

M is the arithmetic mean which acts as a middle ground between the t-norms and t-conorms: it is commutative, monotone, and idempotent, i.e., $M(x,x) = x$ for all x.

Let $\mathbf{t} = (t_1, \ldots, t_k)$ be a vector of k knots such that $0 < t_i < t_{i+1} < 1$ for all $i = 1, \ldots, k$. For simplicity, we presume that $t_i = 0$ for $i < 1$ and $t_i = 1$ whenever $i > k$.

Moreover, assuming that $\cdot/0 = 0$, we define the *B-spline basis functions* for $i \in \mathbb{N}$, $j = 0, 1, \ldots$ and $x \in [0,1]$ recursively as:

$$\begin{aligned} N^{\mathbf{t}}_{i,j}(x) &= \begin{cases} 1 & \text{if } x \in [t_{i-1}, t_i), \\ 0 & \text{otherwise,} \end{cases} && (j = 0) \\ N^{\mathbf{t}}_{i,j}(x) &= \frac{x - t_{i-1}}{t_{i+j-1} - t_{i-1}} N^{\mathbf{t}}_{i,j-1}(x) + \\ & \quad \frac{t_{i+j} - x}{t_{i+j} - t_i} N^{\mathbf{t}}_{i+1,j-1}(x), && (j > 0) \end{aligned}$$

Then we call $\varphi_{\mathbf{V}}(x,y)$ a *B-spline surface* (see, e.g., [6, 11]), if it is given by:

$$\varphi_{\mathbf{V}}(x,y) = \sum_{\alpha=1}^{\eta} \sum_{\beta=1}^{\eta} v_{\alpha,\beta} N^{\mathbf{t}}_{\alpha-p,p}(x)\, N^{\mathbf{t}}_{\beta-p,p}(y)$$

for some control point matrix $\mathbf{V} = (v_{\alpha,\beta}) \in [0,1]^{\eta\times\eta}$, $\eta = p + k + 1$, where $p \geq 1$ is the B-spline degree and $k \geq 0$ is the number of knots, fulfilling the following constraints:

- $v_{\alpha,\beta} \leq v_{\alpha+1,\beta}$ and $v_{\alpha,\beta} \leq v_{\alpha,\beta+1}$ for every α, β,
- $v_{1,1} = 0$ and $v_{\eta,\eta} = 1$,
- $v_{\alpha,\beta} = v_{\beta,\alpha}$ for every α, β,

which guarantee that φ is then a symmetric aggregation operator.

Aggregation functions are typically chosen based on the set of desirable theoretical properties they fulfil, and which are deemed useful in a given application, e.g., strict monotonicity or Lipschitz continuity, see [12]. In particular, there exist numerous characterisation theorems which state that an aggregation function fulfils properties A, B, and C if and only if it is, say, a weighted arithmetic mean. This of course requires that some expert knowledge is at our disposal – but such we are about to obtain only in the sequel.

There are also some methods for fitting certain parametrised classes of aggregation operators to empirical data, see, e.g., [6, 11, 12]. In this setting, we are interested in identifying a set of coefficients which defines a function that, for some predefined inputs, yields outputs as close (with respect to, for instance, the sum of squared errors) as possible to the reference ones. Maximising a goodness-of-fit criterion or minimising an error measure serves then as a working proof of their optimality when dealing with the problem at hand. We shall employ this approach here too.

The optimal control point matrix $\mathbf{V}$ will be determined for each method separately. Let us assume that we have m pairs of functions to compare. Denote the u-th function pair with (f_i, f_j) and let the nonsymmetric similarities generated by an algorithm be denoted with $x_l^{(u)} = \tilde{\mu}(f_i, f_j)$ and $x_r^{(u)} = \tilde{\mu}(f_j, f_i) \in [0,1]$. Moreover, let $y^{(u)}$ be the true (desired) output, where $y^{(u)} = 1$ if the two functions are mutated versions of each other and 0 if they are dissimilar. Then the optimal $\mathbf{V}$ is a solution to the minimisation problem:

$$\min_{\mathbf{V}} \sum_{u=1}^{m} \left(\sum_{\alpha=1}^{\eta} \sum_{\beta=1}^{\eta} v_{\alpha,\beta} N^{\mathbf{t}}_{\alpha-p,p} \left(x_l^{(u)} \right) N^{\mathbf{t}}_{\beta-p,p} \left(x_r^{(u)} \right) - y^{(u)} \right)^2, \tag{1}$$

with respect to the following constraints: $v_{1,1} = 0$, $v_{\eta,\eta} = 1$, $v_{\alpha,\beta} = v_{\beta,\alpha}$, $v_{\alpha+1,\beta} - v_{\alpha,\beta} \geq 0$, $v_{\alpha,\beta+1} - v_{\alpha,\beta} \geq 0$, for all α, β. Optimal $p \in \{1,2,3\}$ and $k \in \{1,2,\ldots,5\}$ are chosen via exhaustive grid search. Note that in this paper we only consider equidistant knots.

2.4. Testing code clone detection systems

There are two main approaches to testing code clone detection systems in the literature: by using real-world source code repositories (with manually labelled clones) [13, 44, 63] or by creating source code databases with code clones injected/generated/simulated algorithmically [51, 55, 62].

The former approach is rather tedious: most of us do not have the resources to inspect every pair of codes. There are, however, some notable exceptions, including the Java language-oriented BigCloneBench [63], where its creators actually labelled the clones by hand. It is worth stressing that, unless all the pairs of programs are manually inspected, metrics such as *recall* (how many existing clones were mined) cannot be truly assessed. What is more, parameters of a dataset cannot be freely adjusted, e.g., its "difficulty" or size. Also, unfortunately, no such benchmark set exists for R.

In the latter mode, some source code base is only a point of departure. In addition, we need to come up with a list of different attacks or mutators whose aim is to alter code fragments in various ways, see, e.g., [55, 62]. In this paper we follow a similar approach, but with some crucial alterations.

Unlike in [55], we will be mutating each function in possibly many different ways rather than defining a clone as a code chunk modified by means of a single mutation. We believe this approach is more realistic: a student can come up with many alterations to conceal the copying of their classmate's work. What is more, this makes the whole task much more difficult for the algorithmic checkers. While some automatically generated clones might seem artificial, they still perform exactly the same task as the original source.

Also, it is worth noting that a recent paper [51] uses some obfuscators and decompilers to obtain mutated code in the case of Java. The idea is interesting on its own, however, there are no tools of this kind for the R language. What is more, let us emphasise that, contrary to the cited paper, in our approach all parameters governing the benchmark set generation will be controllable.

We will mainly be interested in Type-3 and Type-4 clones as we perceive Type-1 and Type-2 ones as too trivial in real-life scenarios. Moreover, there are already many studies related to them in the literature [4, 13, 20, 42, 52, 56]. Hence, each source code chunk will be normalised prior to comparison (e.g., by removing comments and indentation).

2.5. Proposed framework for benchmarking of methods

R is a functional programming language where *functions per se* play the key role, hence we shall be detecting clones on this very level. This however comes with no loss of generality, as R allows nested functions, therefore any larger code chunks can be embraced within a "super" function, which we can call `main()`, for example.

From a bird's eye view, the framework for benchmarking of methods for code clone detection that we shall use here is designed as follows, see also Fig. 1.

1. Collect mature, well-established and high-quality software packages or libraries and form the repository of base functions, B. In our case, see

Section 2.6 for discussion, this will be a selection of the most prominent packages from the CRAN repository.

2. Sample a number of functions from B without replacement, thus creating a reference set instance $R \subset B$. It is assumed that all the functions in R are *dissimilar* (0) from each other (because B is comprised of high-quality, well-tested, peer-reviewed code).

3. Sample a number of functions from R with replacement, clone each of them and then modify the clones by applying a series of *mutation operators* (attacks) on their individual code lines. The number of applications of the mutation operators can vary. This mimics the behaviour of a dishonest student or a negligent programmer. The independent and the cloned works altogether form a single benchmark set instance, X. We assume that all the functions derived from the same routine should be considered *similar* (1) to each other (and thus form a cluster of similar functions).

Note that the number of possible benchmark data sets X that can be derived this way (by repeating Steps 2 and 3) is very large, thus allowing for a reliable comparison of plagiarism detection algorithms. Moreover, the data set size, proportion of code clones and the intensity of mutators can be strictly controlled, therefore their effect on the algorithms' behaviour can be studied rigorously and objectively. In practice, we shall be sampling a few instances for each parameter set, and computing the aggregated accuracy, precision, or recall. Hence, we will be measuring the performance of each algorithm by

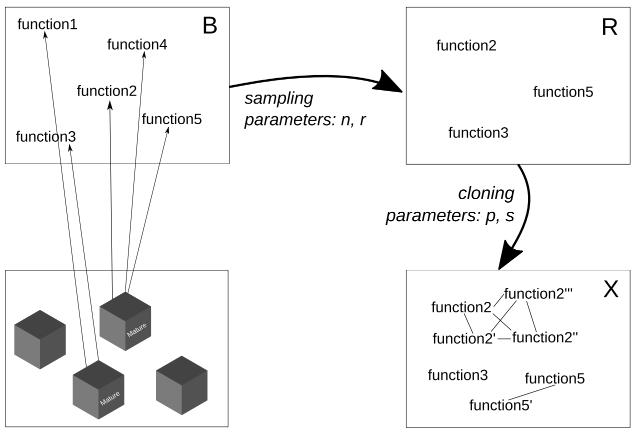

Figure 1: Single benchmark set generation at a glance

means of the standard metrics used for the purpose of evaluation of binary classifiers in statistical and machine learning, see, e.g., [23].

Let us now describe the benchmarking procedure that we propose here in more detail. Assume we are given the base repository B and that we have defined the set of admissible mutators $\mathcal{M}_1, \ldots, \mathcal{M}_k$. There are four parameters that govern the process of generating X.

- sampling parameters (used in Step 2):
  - $n \in \mathbb{N}$ – output sample size, $|X|$;
  - $r \in [0, 1]$ – desired fraction of clones in X;
- cloning parameters (used in Step 3):
  - p – probability of performing each kind of mutation on any vulnerable (i.e., one on which a particular attack can be applied) code line;
  - $s \geq 0$ – parameter of the Zipf distribution controlling how eagerly the same functions are selected for cloning over and over again, see below for details.

The generated benchmark set will consist of n functions. In Step 2, $m = \lfloor (1 - r)n \rfloor$ functions are sampled without replacement from B so as to obtain $R = \{f_1, \ldots, f_m\}$.

The number of clones will thus be equal to $\lceil rn \rceil$. To generate a clone, we select its parent from R in such a way that the likelihood of choosing f_i is given by:

$$\mathfrak{p}(i) = \frac{i^{-s}}{\sum_{j=1}^{m} j^{-s}},$$

which is the probability mass function of the Zipf distribution with exponent s and support $\{1, \ldots m\}$. Such a process is a realisation of the *preferential attachment* or the rich-get-richer rule, see, e.g., [45]. It has frequently been used to describe the occurrences of particular words in texts [38], citations to papers [64], number of hyperlinks to websites [2] and so on. Hence, it is also a natural choice in the case of our domain and leads to benchmark sets where some functions are cloned more eagerly than others. Note that if $s = 0$, the probability of choosing each f_i is the same, hence Zipf's reduces to the discrete uniform distribution. However, the greater the exponent, the more skewed the probability mass. For $s \simeq 1$ we get a heavy-tailed distribution.

In order to mimic a cheating student, we consider each possible mutator $\mathcal{M}_1, \ldots, \mathcal{M}_k$. For the j-th mutation operator, we find all the nodes in the abstract syntax tree of the parent function f_i on which $\mathcal{M}_j$ can be applied (for instance, a change of variable names does not make sense for expressions made up of numeric constants only). An actual mutation of the node happens with some small probability p.

Note that although there are n functions in X, of which $\lceil rn \rceil$ are clones, the number of cloned *pairs* strictly depends on the s parameter as well – all the clones derived from the same parent form a family (cluster) of mutually similar functions. In real-world datasets, we observed that the estimated proportion of cloned pairs is ca. 1–5% (leading to a highly imbalanced problem). Therefore, for the sake of realism, we recommend to fix n and s and then adjust r so as to obtain the desired fraction of similar pairs. Otherwise, for fixed r, the proportion decreases as n increases.

2.6. R as a functional programming language

R [50] is an open source (GNU) version of the S language which originated in the late 1970s as a tool for statistical computing and graphics. Quoting the R project authors themselves[1]:

> *R is an integrated suite of software facilities for data manipulation, calculation and graphical display. It includes*
>
> - *an effective data handling and storage facility,*
> - *a suite of operators for calculations on arrays, in particular matrices,*
> - *a large, coherent, integrated collection of intermediate tools for data analysis,*
> - *graphical facilities for data analysis and display either on-screen or on hardcopy, and*
> - *a well-developed, simple and effective programming language which includes conditionals, loops, user-defined recursive functions and input and output facilities.*

While the R syntax resembles that of the imperative C language, its semantics has been heavily inspired by the functional Scheme, being itself a Lisp dialect. The book by Chambers [15] accompanied by the Scheme-oriented one by Abelson and Sussman [1] can serve as a comprehensive introduction to the core principles of R. Let us just highlight a few of them here.

[1]See `https://www.r-project.org/about.html`, last access 2021-07-06.

- Functions are first-class objects, i.e., functions can be passed as arguments or returned as output values to/from other functions as well as be assigned to variables and stored in data structures (e.g., vectors, matrices).

- Each language construct is in fact a function call, e.g., `2+2` is equivalent to `` `+`(2, 2) `` and `if (cond) { print(":)"); return(7) }` translates to `` `if`(cond, `{`(print(":)"), return(7))) ``. Therefore, even the most complex series of expressions translates to a composition of mutually nested function calls.

- Most functions have no side effects, i.e., they return the same value for the same set of arguments and do not modify their arguments in-place (obviously there are some exceptions to this rule, e.g., random number generation or plotting functions).

2.6.1. Abstract syntax tree vertex types

The *mutation operators* that we shall introduce in Section 2.6.2 are based on an *abstract syntax tree* (AST), which itself is generated during the syntactic analysis of a source code. In an AST, each vertex corresponds to a language expression. In the R language, we distinguish the following expression types:

1. simple:

   (a) constant – a hard-coded value, e.g., `1.0`, `TRUE`, or `"string"`,

   (b) symbol – variable names, e.g., `x` or `sum`,

2. compound:

   (a) function call – an expression of the form $f(a_1, \ldots, a_n)$, where $f, a_1, \ldots, a_n$ are R expressions themselves (simple or compound), denoting a call to f with arguments $a_1, \ldots, a_n$, e.g., `rep(x, 8)` or `(function(x) x^2)(1/exp(y))`.

An AST is a directed acyclic graph such that constants and symbols are its terminal nodes (leaves). Moreover, if a vertex represents a function call, its children denote the arguments passed on input.

Note that in R it is particularly easy to access the language parser programmatically. Moreover, language objects themselves may be manipulated on the fly. A call to `parse(text="source_code")` returns an unevaluated expression. It has also the nice side effect that it removes source code comments, standardises code indentation, etc.

2.6.2. Mutation operators

A mutation operator (*mutator* or *attack*) is a well-defined alteration of a code chunk which does not change its meaning. In other words, the original and the altered expression can be used interchangeably for exactly the same purpose. From our perspective, mutators are modifications of whole parts of abstract syntax trees, replacing particular subtrees with different ones.

Below we provide the list of the most noteworthy mutators that we will use in our study. Many of these mutations are also listed in different papers in the area, e.g., [55, 62], where they are supported by some empirical studies [5, 30] about how developers alter their code, e.g., in their copy-paste routine. While we could not find any study about how students modify code to trick their teachers, we believe students are not different from other professional developers and the aforementioned findings might apply to them as well. Our own teaching experience seems to support these choices too.

While most of the following mutations can be applied to any programming language, *Expanding of nested function calls*, *Memoisation*, and *Applying the pipe operators* are R-language specific.

Changing and aliasing variable names. In this mutation, variable names are replaced with new, randomly generated ones. Only the AST vertices of type *symbol* are vulnerable to this mutator. Quoting R's manual on `make.names`, *"A syntactically valid name consists of letters, numbers and the dot or underline characters and starts with a letter or the dot not followed by a number. Names such as* `".2way"` *are not valid, and neither are the reserved words."*

Example source code before and after this mutation applied on all vulnerable nodes:

```
# before
sum1 <- function(x) {
    s <- 0
    for(element in x) s <- s + element
    s
}

# after
sum2 <- function(ryLr) {
    K4rg <- 0
    for (fTL1 in ryLr) K4rg <- K4rg + fTL1
    K4rg
}
```

Recall that functions are first-class objects. Hence, we allow function calls to be affected by this mutator as well. This is done by introducing an alias to an existing variable name. For instance, after assigning `f <- sum`, we may use `s <- f(x)` instead of `s <- sum(x)` in further spots.

In order to make this mutator more realistic, we do not substitute names of built-in operators and control structures such as `if`, `for`, `while`, `+`, `&`, or `{`.

For example:

```r
# before
isEven1 <- function(number) {
    stopifnot(length(number) == 1)
    number %% 2 == 0
}

# after
isEven2 <- function(number) {
    FLcr <- stopifnot
    tCu4 <- length
    FLcr(tCu4(number) == 1)
    number %% 2 == 0
}
```

Expanding nested function calls. Next mutator is applied in cases where a function call is an argument of another call. A complex, multi-level expression of this kind will be replaced with a series of calls (including the necessary assignments).

For example:

```r
# before
r1 <- function(x, y) {
    sum((x-mean(x))*(y-mean(y)))/
    (sqrt(sum((x-mean(x))^2))*sqrt(sum((y-mean(y))^2)))
}

# after
r2 <- function(x) {
    qz3u32r6 <- mean(x)
    a4f36h4f <- x-qz3u32r6
    ZA3d1rr1 <- mean(y)
    j3f25hab <- y-ZA3d1rr1
    Qvb6h7Ab <- (a4f36h4f)*(j3f25hab)
    nbF2346h <- sum(Qvb6h7Ab)

    VDSdf311 <- sqrt(sum((x-mean(x))^2))  # these can be
        unwound further
    xv3w53df <- sqrt(sum((y-mean(y))^2))
    vcbre654 <- VDSdf311 * xv3w53df

    nbF2346h/vcbre654
}
```

Memoisation. In the above example, we note that, e.g., `x-mean(x)` is called twice. The assumption that functions have no side effects, i.e., calling them on exactly the same arguments always yields the same results, allows us to cache the precomputed value of a complex expression in order to refer to it in further computations.

For example:

```
# r1 above after applying the memoisation mutator
r3 <- function(x, y) {
     xm <- (x-mean(x))
     ym <- (y-mean(y))
     sum(xm*ym)/(sqrt(sum(xm^2))*sqrt(sum(ym^2)))
}
```

Changing the order of independent code lines. There are cases where two expressions can be computed independently, so it does not matter which one is written first. In the next mutation operator, we detect such situations and swap the independent code fragments.

For example:

```
# r3 above after changing the order of independent lines of
    code
r4 <- function(x, y) {
     ym <- (y-mean(y))
     xm <- (x-mean(x))
     sum(xm*ym)/(sqrt(sum(xm^2))*sqrt(sum(ym^2)))
}
```

Changing the order of operands. Another mutation is based on an observation that there are many binary operators that are symmetric, i.e., they return the same value regardless of the order of the arguments given. Therefore, we may safely swap the arguments passed to `+` (vector addition), `*` (multiplication), `&` (vectorised logical conjunction), `|` (disjunction), and `==` (vectorised testing for equality). Also, we can similarly change the order of operands in expressions involving `<=`, `<`, `>=`, and `>` operators, however, the operators themselves must also be substituted with their reflected versions, e.g., `2 < 3` is equivalent to `3 > 2`.

Note that this mutator is not applied in the case of `&&` (scalar logical conjunction) and `||` (scalar disjunction), as both of them rely on lazy evaluation of their inputs.

For example:

```
# r4 above after applying this mutator
r5 <- function(x, y) {
     ym <- (y-mean(y))
     xm <- (x-mean(x))
```

```
    sum(ym*xm)/sqrt(sum(ym^2)))*(sqrt(sum(xm^2))
}
```

Using tautologies to modify logical conditions. Frequently, conditional expressions rely on the testing of various combinations of logical conditions. The next mutation operator alters the nodes corresponding to logical operators: `&`, `&&`, `|`, and `||` based on, amongst others, De Morgan's laws. For instance, we replace `a && b` with `!(!a || !b)` and `a || b` with `!(!a && !b)`. What is more, expressions involving negations of comparisons can be rewritten using the dual relations, e.g., `!(a > b)` becomes `a <= b`.

For instance:

```
# before
get_bmi1 <- function(weight, height) {
    bmi <- weight/height^2
    if (bmi > 18.5 && bmi <= 25)     "normal"
    else                             "other"
}

# after
get_bmi2 <- function(weight, height) {
    bmi <- weight/height^2
    if (!(bmi <= 18.5 || bmi > 25)) "normal"
    else                             "other"
}
```

Changing loop types. We also allow for rewriting `for` (*for-each*) loops in terms of `while`-based expressions. For instance:

```
for (element in sequence) {
    expressions
}
```

is transformed to:

```
iterating_variable <- 1
len <- length(sequence)
while (iterating_variable <= len) {
    # in each expression, every occurrence of
    # `element` is replaced by
    # sequence[[iterating_variable]]
    iterating_variable <- iterating_variable + 1
}
```

The knowledge of frequently applied functions and operators and how do they translate to more primitive forms can increase the usability of a plagiarism detection system as well as be used for defining new code mutations. In particular, R provides numerous map–filter–reduce tools known from other functional programming languages. Functions such as `apply()`, `lapply()`, `sapply()`, etc., are frequently used as a substitute for the imperative-style `for` loops when an iteration over whole collections of objects is needed. For instance:

```
x <- lapply(sequence, function)
```

can be transformed to:

```
out <- vector()
for (element in sequence)
    out[[length(out)+1]] <- function(element)
```

Applying the pipe operators.. We should emphasise the growing popularity of the forward pipe operators, `|>` (since R 4.1) and `%>%` (from the `magrittr` package on CRAN), which resemble, e.g., Haskell's *bind* `>>=` construct. These operators provide their users with a more object-orientated syntax, e.g., instead of writing `mean(split(height, gender))` we can refer to `height |> split(gender) |> mean()`. Therefore, this kind of mutation allows for rewriting nested function calls in terms of piping and vice versa.

Adding ad-hoc code lines. Another mutator is based on an observation that one can mislead some antiplagiarism systems by adding a few meaningless, randomly generated lines of code anywhere within a function's body. Every added line potentially breaks the flow of a sequence of operations, which would otherwise be detected as similar. Token-based antiplagiarism methods are expected to be particularly vulnerable to this mutation.

2.6.3. Package repositories

The Comprehensive R Archive Network (CRAN) is the largest and the most popular peer-reviewed repository of open source R packages, see `https://cran.r-project.org/`. The number of hosted projects grows exponentially; there were ca. 2000 packages in 2010, 3500 in 2012, 6000 in 2014, 9000 in 2016, 12500 in 2018, and as of 2021-07-05 the repository contains 17811 items.

CRAN is an inherent part of the R environment pipeline – new packages can easily be downloaded and installed via a single call to:

```
install.packages("package_name")
```

Due to the widespread adoption of CRAN, new projects tend to stand on the "shoulders of giants" – packages often have multiple dependencies and call many different functions from many libraries. The quality of code, especially in the case of the more frequently used packages that have been under constant development for years, is assured by CRAN's Continuous Integration system as well as the vibrant, engaged open source community. Many libraries are thus perfect candidates for the inclusion in the base function repository B.

On a side note, some R packages can also be found in the Bioconductor repository (`https://www.bioconductor.org/` – tools for biostatistics) as well as software hosting facilities such as GitHub (`https://github.com/`).

3. Algorithms

3.1. Methods for code clone detection

Theoretically, it is possible to generate, in a way described in Section 2, a set of all possible clones of each function in B, by applying all the *mutators* in every possible configuration. An oracle could easily decide whether a given function is a clone of some $f_i \in X$ just by querying such a database. In practice, this is of course computationally intractable. Moreover, real-world datasets might also consist of additional, random mutations of individual code lines. This is why approximate code clone detection algorithms are needed, so that a similarity assessment can be performed using a finite amount of computer resources.

We assume that each $f_i \in X$ is initially expressed as a plain-text source code, i.e., it is a character string. At the preprocessing stage, we should unify the code style. In order to remove comments, redundant whitespaces, apply coherent indentation, etc., we simply call `f_i <- deparse(parse(text=f_i))` in R.

In this paper we shall separate the way code is represented in computer memory from the particular algorithms used to perform the similarity assessment of two given objects.

We distinguish two main code representation schemes: *sequential*, where data is stored in a one-dimensional vector (characters, function calls, and tokens) and *graph*-based (program dependence graph). Sequential representations will be compared by means of the Levenshtein and q-gram distances, greedy string tiling, and the Smith–Waterman algorithm. Furthermore, we will study three different algorithms to compare PDGs: the McGregor and Weisfeiler–Lehman methods as well as the SimilaR algorithm which we have recently implemented in [9].

This gives us 15 different code clone detection methods, some of which are studied for the first time, see Table 1 for a summary.

3.2. Code representation

Let us first discuss the four code representation schemes outlined above.

Table 1: An overview of the methods for code clone detection considered in our study. "–" means that a comparer cannot be applied to a given code representation scheme and "new" denotes a method considered, to the best of our knowledge, for the first time in the literature.

code representation comparer	characters	function calls	tokens	PDG
Levenshtein distance	Simian [21]	new	[35]	–
Smith–Waterman algorithm	new	new	new	–
greedy string tiling	NICAD [54]	new	JPlag [48]	–
q-grams	MOSS [58]	new	MOSS [58]	–
McGregor, VF2 algorithm	–	–	–	GPLAG [39]
Weisfeiler–Lehman algorithm	–	–	–	new
SimilaR	–	–	–	[9]

Characters. The simplest possible representation of a function's preprocessed source code is by means of a plain-text Unicode character string. For instance:

```
function(x) {
    y = x**2
    y
}
```

becomes:

```
"function (x)\n{\n    y <- x^2\n    y\n}"
```

where `\n` denotes the newline character. From now on, we shall denote the set of functions in X represented in this very way as $\mathrm{U}(X)$.

Tokens. As far as antiplagiarism systems are concerned, tokens constitute a popular code representation. Its main advantage is that they omit particular variable names (all variables are assigned the `SYMBOL` token). Instead, they focus on elements of the language grammar.

For example, the aforementioned function becomes:

```
expr, FUNCTION, '(', SYMBOL_FORMALS, ')', expr, '{', equal_
```

```
    assign , SYMBOL , expr , EQ_ASSIGN , expr , SYMBOL , expr , '^',
    NUM_CONST , expr , SYMBOL , expr , '}'
```

In R, obtaining a token sequence is as easy as calling `getParseData(parse (text=as.character(as.expression(code))))$token`. From now on we will denote with $\mathrm{T}(X)$ the set of all tokenised functions in X.

Function calls. Recall that R is a programming language where functions play a key role. Moreover, there exists a central package repository, CRAN, on which R users rely eagerly. Other programmers develop their own projects based upon "building blocks" delivered by packages on CRAN. We can therefore expect two functions serving the same purpose to be referring to similar procedure names (e.g., text processing facilities or matrix algebra operations). Denote with $\mathrm{C}(X)$ the set of function call sequences for each element in X.

For example, the function `r1()` above becomes:

```
/, sum , *, - mean , -, mean , *, sqrt , sum , ^, -, mean , sqrt ,
    sum , ^, -, mean
```

Program Dependence Graph. A Program Dependence Graph (PDG) is the most sophisticated representation used in code clone detection up to date. It was first proposed in [17] and then considered, e.g., in [24, 32, 34, 39, 49].

A PDG models the dependencies between R expressions. Just as in an abstract syntax tree (AST), its vertices correspond to individual expressions. However, each node is assigned a colour, denoting the type of language construct (e.g., a function call or an assignment). Moreover, there are two types of edges. Control dependency edges represent a nesting of function calls, just like in an AST. On the other hand, data dependency edges represent data flow in a program; there is an edge from v_i to v_j if some variable assigned in v_i is used in v_j. The transforming of the source code to the corresponding PDG is described in detail in [22].

The set of all program dependence graphs of functions in X will be denoted with $\mathrm{G}(X)$.

3.3. Sequence comparers

Let us assume we wish to compare two functions f and g represented as characters, tokens, or function call sequences, i.e., $f, g \in U(X)$, $f, g \in T(X)$, or $f, g \in C(X)$, respectively. In this paper we study four different sequence comparers.

Levenshtein distance. The simplest pairwise comparison algorithm uses Levenshtein's string edit distance [36]. Intuitively, the Levenshtein distance between f and g, $\mathrm{lev}(f, g)$, is the minimal number of insertions, deletions, or substitutions required to transform one sequence into another. In our case, we will be mapping the distance into a similarity degree as follows:

$$\mu_1(f, g) = 1 - \frac{\mathrm{lev}(f, g)}{\max(|f|, |g|)} \in [0, 1],$$

$$\tilde{\mu}_1(f, g) = 1 - \frac{\mathrm{lev}(f, g)}{|f|} \in [0, 1],$$

where $|f|$ denotes the length of f. Clearly, $\mu_1(f, g)$ if and only if f and g are identical.

Smith–Waterman algorithm. The Smith–Waterman algorithm [61] is frequently used in bioinformatics for comparing DNA sequences and is based on the so-called local alignment. While global alignment is used mainly to compare sequences of similar lengths, local alignment can be used to detect similar subparts. This method is similar to the Levenshtein distance: some single-item operations are also considered, such as element substitution and gap insertion. It requires defining four parameters: scoring values for the same element $w = s(a, a)$, scores for different elements $d = s(a, b)$ as well as the beginning of the gap (v) and its continuation (u) penalty.

Assuming that $s = \mathrm{sw}(f, g)$ is the Smith–Waterman score, we express the similarity measure as:

$$\mu_2(f, g) = \frac{s}{(|f| + |g|)\, w - s} \in [0, 1].$$

$$\tilde{\mu}_2(f, g) = \frac{s}{|f| \cdot w} \in [0, 1].$$

Greedy string tiling. The greedy string tiling algorithm, described in [67], is used in the token-based JPlag system [48]. First, it finds the longest common substrings. All their elements will no longer be part of any further match. In the consecutive iterations, subsequent, shorter and shorter common substrings are located and further excluded from matching. The minimum match length d is taken into account so as to increase the algorithm's robustness with regards to negligible differences between two sequences.

The output similarity measure is given by:

$$\mu_3(f, g) = \frac{2\,\mathrm{coverage(tiles)}}{|f| + |g|} \in [0, 1],$$

$$\tilde{\mu}_3(f,g) = \frac{\text{coverage(tiles)}}{|f|} \in [0,1],$$

where coverage(tiles) is a sum of lengths of all the detected common substrings.

q-gram distance. For any fixed $q \in \mathbb{N}$, a q-gram of a sequence $(x_1, \dots, x_k)$, see, e.g., [33] is any subsequence $(x_i, \dots, x_{i+q-1})$, $i = 1, \dots, k-q+1$.

Given f and g, denote with Q the set of all their common q-grams. Moreover, let $\hat{f}_i$ and $\hat{g}_i$ denote the number of occurrences of the i-th q-gram of f and g, respectively, $i = 1, \dots, |Q|$. Then the q-gram distance is given by:

$$\mu_4(f,g) = 1 - \frac{\sum_{i=1}^{|Q|} |\hat{f}_i - \hat{g}_i|}{\sum_{i=1}^{|Q|} |\hat{f}_i| + \sum_{i=1}^{|Q|} |\hat{g}_i|} \in [0,1].$$

$$\tilde{\mu}_4(f,g) = \frac{\sum_{i=1}^{|Q|} \min(\hat{f}_i, \hat{q}_i)}{\sum_{i=1}^{|Q|} |\hat{f}_i|} \in [0,1].$$

3.4. PDG comparers

Let us now assume that we wish to compare two program dependence graphs $f, g \in \mathrm{G}(X)$.

McGregor algorithm. The first PDG-based approach uses the McGregor algorithm to find the most common subgraph. Its detailed description can be found in [43]. It is a typical backtracking algorithm which has exponential time complexity. A vertex is considered as common to both graphs only if all its neighbours are matched as well. This makes it very inflexible; even a small difference between the graphs results in a low similarity score.

Assuming that the most common subgraph of f and g is s, we define the corresponding similarity measures as:

$$\mu_5(f,g) = \frac{|s|}{|f| + |g| - |s|},$$

$$\tilde{\mu}_5(f,g) = \frac{|s|}{|f|},$$

where $|s|$ is the number of vertices in s.

A modified Weisfeiler–Lehman algorithm. In [59] proposed was a graph kernel-based algorithm which uses the Weisfeiler–Lehman test for graph isomorphism [66]. Its h parameter controls the number of iterations; with small h we seek small similar subgraphs, while with large h we switch to comparing the graphs in their entirety.

Let $\hat{f}_i^{(j)}$ and $\hat{g}_i^{(j)}$ denote the number of occurrences of the i-th label assigned by the algorithm in the j-th iteration to vertices in f and g, respectively. Denote with L the set of all unique labels. The output similarity measure is given by:

$$\mu_6(f,g) = 1 - \frac{1}{h}\sum_{j=1}^{h} \frac{\sum_{i=1}^{|L|} |\hat{f}_i^{(j)} - \hat{g}_i^{(j)}|}{\sum_{i=1}^{|L|} |\hat{f}_i^{(j)}| + \sum_{i=1}^{|L|} |\hat{g}_i^{(j)}|} \in [0,1].$$

$$\tilde{\mu}_6(f,g) = \frac{1}{h}\sum_{j=1}^{h} \frac{\sum_{i=1}^{|L|} \min\left(\hat{f}_i^{(j)}, \hat{g}_i^{(j)}\right)}{\sum_{i=1}^{|L|} |\hat{f}_i^{(j)}|} \in [0,1].$$

SimilaR algorithm. In [9] we proposed and implemented a modified version of the above algorithm named SimilaR, which is described in very detail therein. As opposed to the original algorithm, here we assign the same label to a pair of vertices also whose neighbourhoods differ, provided that this difference is relatively small.

The formulas for the similarity measures $\mu_7(f,g)$, $\tilde{\mu}_7(f,g) \in [0,1]$ are identical to the ones above.

4. Results

Let us proceed with the empirical comparison of the 15 different methods for code clone detection and the evaluation of ways to aggregate the nonsymmetric similarity measures. First of all, as far as the base function repository B is concerned, we have extracted all the 7089 functions that consist of at least 5 lines of code (the choice is repository-dependent in general; e.g., authors of [13] used chunks at least six lines long, arguing that smaller clones tend to be more spurious; in our case it is 5 that turned out to provide a good balance between the dataset size and soundness) and that are included on the list of 233 most well-known, mature R packages. The inclusion of packages was based on the number of referencing libraries (see `https://www.r-pkg.org/depended`), downloads (`https://www.r-pkg.org/downloaded`) and GitHub stars (`https://www.r-pkg.org/starred`). Apart from these, we used the functions from base and recommended packages, which are included in every distribution of R.

Table 2: Plagiarism detection accuracy for the existing tools.

tool	recall	precision	F-measure
JPlag	0.011 ± 0.004	0.902 ± 0.054	0.021 ± 0.028
MOSS	0.164 ± 0.032	0.982 ± 0.053	0.281 ± 0.031

Many different scenarios were considered so as to test the algorithms in a wide variety of settings; the underlying parameters for benchmark sets generation (see Section 2.5) were assumed as follows: $n \in \{50, 200, 500, 1000\}$, $p \in \{0.05, 0.1, 0.25\}$, and $s \in \{1, 1.5, 2\}$ (compare [2, 64]). The r parameter was set so as to obtain 1, 3, or 5% similar pairs of functions in each case. For each of the 108 distinct possible combinations of the parameters, 10 data set instances, X, were generated.

4.1. Existing multipurpose tools

Let us first perform the comparison of the existing, general-purpose tools. We were initially interested in testing 7 of them: MOSS [58], JPlag [48], NiCad [54], CCFinderX [29], Deckard [27], GPLAG [39], and SourcererCC [57]. Our assumption was that a teacher would like to use these tools to identify potential instances of plagiarism. They would submit a set of assignments as-is and simply fetch the results without any particular pre-processing.

As mentioned in the introduction, none of these tools support R natively. While some of them allow to treat R source code as plain text (JPlag) or they accept it labelled as one written in a different language (Python in the MOSS case), the other ones simply either fail to accept R code (e.g., NiCad, SourcererCC, Deckard, and GPLAG which does not work on languages other than explicitly defined) or refuse to run on modern machines (e.g., CCFinderX).

The results for JPlag and MOSS are presented in Table 2. In each case, we report the medians (over 1080 samples) of *precision*, *recall*, and the *F-measure* as well as their *median absolute deviations* (which are robust metrics – suitable for the skewed distributions that we observe here). The obtained recall, i.e., the fraction of the total number of true clones that were actually retrieved (and hence the F-measure, being the harmonic mean of recall and precision) is very unsatisfactory. While JPlag and MOSS were executed with default parameters, it does not seem likely that tuning the underlying knobs would yield sufficiently different results.

We therefore needed to implement all the algorithms listed in Table 1 and discussed above in a way which was R language-aware.

Table 3: Average execution times [s] of each algorithm, where n is the number of considered.

comparer	representation	$n = 50$	$n = 200$	$n = 500$	$n = 1000$
Levenshtein	characters	100.87	1877.67	10971.12	38072.83
	function calls	1.22	22.01	134.16	464.54
	tokens	32.59	578.67	3505.96	12520.83
Smith–Waterman	characters	1.13	19.87	124.37	289.95
	function calls	0.09	1.41	8.79	35.14
	tokens	0.38	6.60	38.86	151.36
greedy string tiling	characters	70.13	1282.52	4735.50	13287.04
	function calls	0.18	3.34	15.26	50.65
	tokens	33.88	598.93	2594.35	8333.98
q-gram	characters	1.57	30.21	213.42	945.61
	function calls	0.23	4.19	27.62	119.16
	tokens	0.63	11.37	77.34	330.80
McGregor	PDG	4649.68	32867.33	60744.67	∞
Weisfeiler–Lehman	PDG	5.45	41.31	182.69	622.77
SimilaR	PDG	5.47	43.42	172.16	643.74

Before proceeding with the analysis of the performance of the considered methods, let us take a look at the algorithms' execution times which are reported in Table 3. Generally, the algorithms based on function calls are the fastest, mostly because it is the most compact representation of a source code. Overall, the algorithms based of function calls, tokens, and PDGs can be recommended for practical use. Also, note that McGregor's algorithm, due to its exponential time complexity, will be excluded from further consideration.

4.2. Choosing the best aggregation operator

Note that each algorithm outputs a symmetric similarity measure $\mu_i \in [0, 1]$ as well as a pair of nonsymmetric values $\tilde{\mu}_i \in [0, 1]$ which are to be aggregated. In order to treat it as a binary classifier, in each case we must learn the optimal (in terms of the average F-measure) threshold value θ. Function pairs with a similarity measure less than θ are classified as not similar (class 0) and class 1 (similar) is assigned otherwise. The estimated, optimal values of θ are specific to each (representation, comparer, fuzzy logic connective) triple and for practical use they can easily be hardcoded in the method's implementation so that a user will never be bothered with its setting.

In Tables 6, 7, 8, 9, and 10 we present the detailed results regarding the plagiarism detection accuracy for all of the considered code representation schemes, comparers, and aggregation functions described in Section 2.3. In each case, we report the medians (over 1080 samples) of *precision*, *recall*,

Table 4: Mean ranks for the aggregation operators (less is better).

aggregation function	recall	precision	F-measure
symmetric	6.643	2.857	4.429
$T_Ł$	6.714	2.607	5.036
T_m	7.464	2.500	5.143
T_p	5.679	3.679	4.107
M	3.786	5.321	3.179
$S_Ł$	2.357	7.571	6.393
S_m	5.714	8.250	8.429
S_p	3.429	7.250	6.000
$\varphi_{\mathbf{V}}$	3.214	4.964	2.286

and *F-measure* as well as their *median absolute deviations*; note that the two classes (0=dissimilar, 1=similar) are highly imbalanced. The underlined values denote the quality measures that do not differ from the best one significantly (Wilcoxon rank-sum test, significance level $\alpha = 0.05$) in each group of three.

What is more, Table 4 gives the ranks for every aggregation method averaged over all (representation, comparer) pairs. The rank of 1 denotes the best result, while 9 is the worst.

We note what follows:

- The algorithms based on function calls (in the symmetric case) have the lowest recall (sensitivity) rate amongst those operating on sequences, ca. 90%, i.e., the fraction of the total number of true clones that they can actually retrieve. However, they often have very good precision, up to 98%, i.e., the fraction of actual clones in the pairs returned by the algorithm. In other words, while they might ignore some of the actual clones, the ones they identify are more likely to be the true instances of plagiarism. Moreover, this representation is the fastest to compute upon.

- On the other hand, character-based representation (in the symmetric case) gives the highest recall and the lowest precision within the sequential group. In other words, it is the most "suspicious" one. It is also the slowest to compute on.

- As far as sequential representations are still concerned, the F-measure (a commonly used aggregate of precision and recall – the harmonic mean) is maximised with token-based algorithms. Note that two state-of-the-art methods, JPlag and MOSS, are based on comparing tokens with greedy string tiling and q-grams (see Table 1) This is the first time these algorithms have been tested in such a comprehensive manner. It is quite surprising that the same approaches which are used in MOSS and JPlag gave much better results in our own framework. It seems that the language-awareness (i.e., tokens generated for source code) is much more important than one could expect. Possibly there are also some details in original algorithms' implementations (which we have no access to) which affects the results somehow.

- The Smith–Waterman algorithm is the fastest amongst all the methods. Recall that the algorithm is frequently used in bioinformatics. Note that 1-grams of function calls, i.e., the algorithm that only takes into account the difference in the number of common names of functions referred to in the source codes, yields a very high precision. Moreover, its specialised version is fast to execute (results not reported in Table 3). Also, the exact algorithm for computing the most common subgraph (McGregor) is useless because of its high time complexity. Moreover, globally, our new algorithm, SimilaR, outperforms all the other methods in terms of both precision and recall as well as the F-measure. It gives some improvements over the Weisfeiler–Lehman method, which we used in the domain of plagiarism detection for the first time in this very paper. Moreover, its run-times are highly competitive.

- In only two of the 42 (comparer, representation, quality measure) cases, the symmetric approach was the best one, while the nonsymmetric ones were significantly worse: precision calculated for the Levenshtein distance on function calls and the Smith–Waterman algorithm on tokens. There were also 9 groups where the symmetric approach was neither significantly worse than the asymmetric ones nor the other way around. In the remaining 31 instances, the asymmetric cases were significantly better than the symmetric approach.

- From Table 4 we read that t-conorms generally result in better recall (with S_m being an exception). It is understandable: they are generalised "OR" operators, so the final similarity measure is expected to be rather high and more pairs will be classified as similar. Analogously, the t-norms, "AND", maximise precision. Interestingly, the arithmetic mean is the second best aggregation function for the F-measure.

4.3. Finding the closest t-(co)norms to the fitted B-spline surfaces

As the B-spline surfaces have many degrees of freedom, we have also decided to test if the fitted $\varphi_{\mathbf{V}}$s resemble some of the known t-(co)norms (see [31]). In order to do so, we minimised the root mean squared error (RMSE,

L_2) between them. Tested families included the following aggregation functions.

- The family $(T^{SS}_\lambda)_{\lambda\in[-\infty,\infty]}$ of *Schweizer–Sklar t-norms* is given by:

$$T^{SS}_\lambda(x,y) \;=\; \begin{cases} T_m(x,y) & \text{if } \lambda=-\infty,\\ T_p(x,y) & \text{if } \lambda=0,\\ T_d(x,y) & \text{if } \lambda=\infty,\\ (\max((x^\lambda+y^\lambda-1),0))^{\frac{1}{\lambda}} & \text{if } \lambda\in(-\infty,0)\cup(0,\infty), \end{cases}$$

  where T_m is the minimum t-norm, T_p is the product t-norm (both defined previously), and T_d is the drastic t-norm, given by $T_d(x,1)=x$, $T_d(1,y)=y$, and $T_d(x,y)=0$ otherwise.

  The family $(S^{SS}_\lambda)_{\lambda\in[-\infty,\infty]}$ of *Schweizer–Sklar t-conorms* is given by:

$$S^{SS}_\lambda(x,y) \;=\; \begin{cases} S_m(x,y) & \text{if } \lambda=-\infty,\\ S_p(x,y) & \text{if } \lambda=0,\\ S_d(x,y) & \text{if } \lambda=\infty,\\ 1-(\max(((1-x)^\lambda+(1-y)^\lambda-1),0))^{\frac{1}{\lambda}} & \text{if } \lambda\in(-\infty,0)\cup(0,\infty), \end{cases}$$

  where S_m is the maximum t-conorm, S_p is the t-conorm dual to T_p (see above), and S_d is the drastic t-conorm, given by $S_d(x,0)=x$, $S_d(0,y)=y$, and $S_d(x,y)=1$ otherwise.

- The family $(T^F_\lambda)_{\lambda\in[0,\infty]}$ of *Frank t-norms* is given by:

$$T^F_\lambda(x,y) \;=\; \begin{cases} T_m(x,y) & \text{if } \lambda=0,\\ T_p(x,y) & \text{if } \lambda=1,\\ T_d(x,y) & \text{if } \lambda=\infty,\\ \log_\lambda(1+\frac{(\lambda^x-1)(\lambda^y-1)}{\lambda-1}) & \text{otherwise.} \end{cases}$$

  The family $(S^F_\lambda)_{\lambda\in[0,\infty]}$ of *Frank t-conorms* is given by:

$$S^F_\lambda(x,y) \;=\; \begin{cases} S_m(x,y) & \text{if } \lambda=0,\\ S_p(x,y) & \text{if } \lambda=1,\\ S_d(x,y) & \text{if } \lambda=\infty,\\ 1-\log_\lambda(1+\frac{(\lambda^{1-x}-1)(\lambda^{1-y}-1)}{\lambda-1}) & \text{otherwise.} \end{cases}$$

- The family $(T^Y_\lambda)_{\lambda\in[0,\infty]}$ of *Yager t-norms* is given by:

$$T^Y_\lambda(x,y) \;=\; \begin{cases} T_m(x,y) & \text{if } \lambda = 0,\\ T_d(x,y) & \text{if } \lambda = \infty,\\ \max\left(1-((1-x)^\lambda+(1-y)^\lambda)^{\frac{1}{\lambda}},0\right) & \text{otherwise.}\end{cases}$$

  The family $(S^Y_\lambda)_{\lambda\in[0,\infty]}$ of *Yager t-conorms* is given by:

$$S^Y_\lambda(x,y) \;=\; \begin{cases} S_m(x,y) & \text{if } \lambda = 0,\\ S_d(x,y) & \text{if } \lambda = \infty,\\ \min\left((x^\lambda+y^\lambda)^{\frac{1}{\lambda}},1\right) & \text{otherwise.}\end{cases}$$

- The family $(T^D_\lambda)_{\lambda\in[0,\infty]}$ of *Dombi t-norms* is given by:

$$T^D_\lambda(x,y) \;=\; \begin{cases} T_d(x,y) & \text{if } \lambda = 0,\\ T_m(x,y) & \text{if } \lambda = \infty,\\ \frac{1}{1+((\frac{1-x}{x})^\lambda+(\frac{1-y}{y})^\lambda)^{\frac{1}{\lambda}}} & \text{otherwise.}\end{cases}$$

  The family $(S^D_\lambda)_{\lambda\in[0,\infty]}$ of *Dombi t-conorms* is given by:

$$S^D_\lambda(x,y) \;=\; \begin{cases} S_d(x,y) & \text{if } \lambda = 0,\\ S_m(x,y) & \text{if } \lambda = \infty,\\ 1-\frac{1}{1+((\frac{x}{1-x})^\lambda+(\frac{y}{1-y})^\lambda)^{\frac{1}{\lambda}}} & \text{otherwise.}\end{cases}$$

- The family $(T^{SW}_\lambda)_{\lambda\in[-1,\infty]}$ of *Sugeno–Weber t-norms* is given by:

$$T^{SW}_\lambda(x,y) \;=\; \begin{cases} T_d(x,y) & \text{if } \lambda = -1,\\ T_p(x,y) & \text{if } \lambda = \infty,\\ \max\left(\frac{x+y-1+\lambda xy}{1+\lambda},0\right) & \text{otherwise.}\end{cases}$$

  The family $(S^{SW}_\lambda)_{\lambda\in[-1,\infty]}$ of *Sugeno–Weber t-conorms* is given by:

$$S^{SW}_\lambda(x,y) \;=\; \begin{cases} S_d(x,y) & \text{if } \lambda = -1,\\ S_p(x,y) & \text{if } \lambda = \infty,\\ \min\left(x+y+\lambda xy,1\right) & \text{otherwise.}\end{cases}$$

- The family $(T_\lambda^{AA})_{\lambda\in[0,\infty]}$ of *Aczél–Alsina t-norms* is given by:

$$T_\lambda^{AA}(x,y) = \begin{cases} T_d(x,y) & \text{if } \lambda=0, \\ T_m(x,y) & \text{if } \lambda=\infty, \\ e^{-((-\log x)^\lambda+(-\log y)^\lambda)^{\frac{1}{\lambda}}} & \text{otherwise.} \end{cases}$$

The family $(S_\lambda^{AA})_{\lambda\in[0,\infty]}$ of *Aczél–Alsina t-conorms* is given by:

$$S_\lambda^{AA}(x,y) = \begin{cases} S_d(x,y) & \text{if } \lambda=0, \\ S_m(x,y) & \text{if } \lambda=\infty, \\ 1-e^{-((-\log(1-x))^\lambda+(-\log(1-y))^\lambda)^{\frac{1}{\lambda}}} & \text{otherwise.} \end{cases}$$

- power means:

$$P_\lambda(x,y) = \begin{cases} \sqrt{xy} & \text{if } \lambda=0, \\ (\frac{x^\lambda+y^\lambda}{2})^{\frac{1}{\lambda}} & \text{otherwise.} \end{cases}$$

- exponential means:

$$E_\lambda(x,y) = \begin{cases} \frac{x+y}{2} & \text{if } \lambda=0, \\ \frac{1}{\lambda}\log(\frac{e^{x\lambda}+e^{y\lambda}}{2}) & \text{otherwise.} \end{cases}$$

Table 5 shows the best fitted aggregation operators in the sense of minimising the RMSE to $\varphi_{\mathbf{V}}$. Interestingly, the Yager t-conorms yield the smallest error in the vast majority of the cases. We have included the induced recall, precision, and F-measures in Tables 6, 7, 8, 9 and 10. As this time the fitting does not involve the R similarity dataset, the results are sometimes worse that those for the original B-spline surface.

5. Conclusions

In this paper we investigated many approaches to source code clones detection. We indicated the problem with the lack of an in-depth investigation of the relevant tools tailored to R, which is a very high-level, functional programming language. We also thoroughly evaluated our new algorithm, SimilaR [9]. It turned out that this program dependence graph-based method

Table 5: Best fitted aggregation operators.

comparer	representation	best fitted function	RMSE
Levenshtein distance	letters	Yager t-conorm (1.742553)	0.065
	function calls	Yager t-conorm (1.591355)	0.062
	tokens	Yager t-conorm (1.886454)	0.088
Smith–Waterman	letters	Yager t-conorm (0.437134)	0.057
	function calls	Yager t-conorm (1.443944)	0.067
	tokens	Yager t-conorm (1.716247)	0.080
Greedy string tiling	letters	Yager t-conorm (1.573940)	0.054
	function calls	Yager t-conorm (1.682680)	0.111
	tokens	Yager t-conorm (1.433622)	0.065
q-grams	letters	Aczel–Alsina t-conorm (0.390426)	0.059
	function calls	Yager t-conorm (2.702127)	0.133
	tokens	Yager t-conorm (1.534131)	0.083
Weisfeiler–Lehman	PDG	power mean (0.804382)	0.072
SimilaR	PDG	Yager t-conorm (2.660945)	0.121

has the best performance in terms of precision, recall, as well as F-measure. At the same time, its execution time is still acceptable.

Another key contribution was the idea of separating the representation form of a code chunk (characters, function calls, tokens, PDG) from the particular algorithm used to compare the representations with each other. This way, we could study the impact each component has on the code clone detection quality. What is more, as seen in Table 1, we assessed some (representation, comparer) pairs that had not been considered in the literature before.

Most importantly, we thoroughly studied a new approach to measuring the similarity of programs. Instead of treating it as a symmetric relation, we decided to formulate it as a measure of inclusion. However, as many applications require a single value on the output anyway, we evaluated many aggregation operators. It turns out that t-conorms maximise recall, while t-norms increase precision. The arithmetic mean is a good way to obtain a high F-measure.

Our approach hence provides a user with an additional degree of freedom; they can choose the appropriate aggregation method depending on their needs (e.g., if merely a quick pre-filtering of potential clones is needed or if one wants to be sure that there is sufficiently strong evidence that the identified cases are indeed instances of plagiarism).

Future work will involve the translation of our study to other programming languages, such as Python, Java, C, Julia, or Haskell. Note that in some lower-level languages (e.g., C), function calls might not necessarily constitute an informative representation of source code. Also, program dependence graphs may be much larger. The fine-tuning of the model parameters is computationally expensive and of course should be re-done for every language from scratch. However, this paper indicates that it might be worth the effort: the obtained results are indeed significantly better than when relying on fixed, ad hoc transformations. Also note that once the best aggregation methods are identified (in the lengthy learning phase), they may be hard-coded within the production environment: predictions, which are of interest to end users, are generated effortlessly.

Note that we have been approximating the fitted B-spline surfaces (which represent arbitrary symmetric aggregation functions) by parametrised t-norms and t-conorms (such as the Yager or Schweizer–Sklar ones). It would be interesting to determine the fit quality of particular classes of t-operators when parameters are learned directly from data (without the B-surface as a proxy).

Some parametrised classes of averaging functions (generalised means) could also be studied, for example, the power, exponential, and other quasi-arithmetic means.

Also, the fitting of the surfaces themselves, i.e., $\varphi_{\mathbf{V}}$, was performed by minimising the sum of squared error. This is computationally convenient, because it can be expressed as a quadratic programming problem. Nevertheless, other error measures such as cross-entropy could also be studied. Moreover, the use of aggregation operators for a “yes–maybe–no” type classification (compare [26]) in the context of plagiarism detection seems interesting.

Acknowledgements

This research was supported by the Australian Research Council Discovery Project ARC DP210100227 as well as the PL-Grid Infrastructure.

Conflict of interest

The authors declare that they have no conflict of interest.

References

[1] H. Abelson, G.J. Sussman, Structure and Interpretation of Computer Programs, MIT Press, Cambridge, MA, USA, 1996.

[2] L.A. Adamic, B.A. Huberman, Zipf’s law and the Internet, Glottometrics 3 (2002) 143–150.

[3] Q.U. Ain, W.H. Butt, M.W. Anwar, F. Azam, B. Maqbool, A systematic review on code clone detection, IEEE Access 7 (2019) 86121–86144.

[4] A.T. Ali, H.M. Abdulla, V. Snasel, Overview and comparison of plagiarism detection tools, in: Proceedings of the Dateso 2011: Annual International Workshop on Databases, Texts, Specifications and Objects, 2011, pp. 161–172.

[5] M. Balint, R. Marinescu, T. Girba, How developers copy, in: 14th IEEE International Conference on Program Comprehension (ICPC'06), 2006, pp. 56–68.

[6] M. Bartoszuk, G. Beliakov, M. Gagolewski, S. James, Fitting aggregation functions to data: Part II – Idempotization, in: J.P. Carvalho, et al. (Eds.), Information Processing and Management of Uncertainty in Knowledge-Based Systems, Springer International Publishing, 2016, pp. 780–789.

[7] M. Bartoszuk, M. Gagolewski, Detecting similarity of R functions via a fusion of multiple heuristic methods, in: Proc. IFSA-EUSFLAT, 2015, pp. 419–426.

[8] M. Bartoszuk, M. Gagolewski, Binary aggregation functions in software plagiarism detection, in: 2017 IEEE International Conference on Fuzzy Systems (FUZZ-IEEE), 2017, pp. 1–6.

[9] M. Bartoszuk, M. Gagolewski, SimilaR: R code clone and plagiarism detection, The R Journal 12 (2020) 367–385.

[10] I.D. Baxter, A. Yahin, L. Moura, M. Sant'Anna, L. Bier, Clone detection using abstract syntax trees, in: Proceedings of the International Conference on Software Maintenance, ICSM '98, IEEE Computer Society, Washington, DC, USA, 1998, pp. 368–378.

[11] G. Beliakov, H. Bustince, T. Calvo, A practical guide to averaging functions, Springer, 2016.

[12] G. Beliakov, A. Pradera, T. Calvo, Aggregation functions: A guide for practitioners, Springer-Verlag, 2007.

[13] S. Bellon, R. Koschke, G. Antoniol, J. Krinke, E. Merlo, Comparison and evaluation of clone detection tools, IEEE Transactions on Software Engineering 33 (2007) 577–591.

[14] D.K. Chae, J. Ha, S.W. Kim, B. Kang, E.G. Im, S. Park, Credible, resilient, and scalable detection of software plagiarism using authority histograms, Knowledge-Based Systems 95 (2016) 114–124.

[15] J. Chambers, Programming with Data, Springer, 1998.

[16] R. Falke, P. Frenzel, R. Koschke, Empirical evaluation of clone detection using syntax suffix trees, Empirical Software Engineering 13 (2008) 601–643.

[17] J. Ferrante, K.J. Ottenstein, J.D. Warren, The program dependence graph and its use in optimization, ACM Transactions on Programming Languages and Systems 9 (1987) 319–349.

[18] D. Fu, Y. Xu, H. Yu, B. Yang, WASTK: A weighted abstract syntax tree kernel method for source code plagiarism detection, Scientific Programming 2017 (2017) 7809047.

[19] M. Grabisch, J.L. Marichal, R. Mesiar, E. Pap, Aggregation functions, Cambridge University Press, 2009.

[20] J. Hage, P. Rademaker, N. van Vugt, Plagiarism detection for Java: A tool comparison, in: Computer Science Education Research Conference, CSERC '11, Open Universiteit, Heerlen, 2011, pp. 33–46.

[21] S. Harris, Simian – similarity analyser, 2021. https://www.harukizaemon.com/simian/.

[22] M.J. Harrold, B. Malloy, G. Rothermel, Efficient Construction of Program Dependence Graphs, Technical Report, ACM International Symposium on Software Testing and Analysis, 1993.

[23] T. Hastie, R. Tibshirani, J. Friedman, The Elements of Statistical Learning: Data Mining, Inference, and Prediction, Springer, 2009.

[24] S. Horwitz, T. Reps, Efficient comparison of program slices, Acta Informatica 28 (1991) 713–732.

[25] J. Hryszko, L. Madeyski, Assessment of the Software Defect Prediction Cost Effectiveness in an Industrial Project, in: L. Madeyski, et al. (Eds.), Software Engineering: Challenges and Solutions, volume 504 of *Advances in Intelligent Systems and Computing*, Springer, 2017, pp. 77–90.

[26] M. Hudec, E. Mináriková, R. Mesiar, A. Saranti, A. Holzinger, Classification by ordinal sums of conjunctive and disjunctive functions for explainable AI and interpretable machine learning solutions, Knowledge-Based Systems 220 (2021) 106916.

[27] L. Jiang, G. Misherghi, Z. Su, S. Glondu, Deckard: Scalable and accurate tree-based detection of code clones, in: Proceedings of the 29th International Conference on Software Engineering, ICSE '07, IEEE Computer Society, Washington, DC, USA, 2007, pp. 96–105.

[28] J.H. Johnson, Identifying redundancy in source code using fingerprints, in: Proceedings of the 1993 Conference of the Centre for Advanced Studies on Collaborative Research: Software Engineering – Volume 1, CASCON '93, IBM Press, 1993, pp. 171–183.

[29] T. Kamiya, S. Kusumoto, K. Inoue, CCFinder: A multilinguistic token-based code clone detection system for large scale source code, IEEE Transactions on Software Engineering 28 (2002) 654–670.

[30] M. Kim, L. Bergman, T. Lau, D. Notkin, An ethnographic study of copy and paste programming practices in OOPL, in: Proc. 2004 International Symposium on Empirical Software Engineering, 2004, pp. 83–92.

[31] E.P. Klement, R. Mesiar, E. Pap, Triangular norms. Position paper II: General constructions and parametrized families, Fuzzy Sets and Systems 145 (2004) 411–438.

[32] R. Komondoor, S. Horwitz, Using slicing to identify duplication in source code, in: Proceedings of the 8th International Symposium on Static Analysis, 2001, pp. 40–56.

[33] G. Kondrak, N-gram similarity and distance, in: Proceedings of the 12th International Conference on String Processing and Information Retrieval, 2005, pp. 115–126.

[34] J. Krinke, Identifying similar code with program dependence graphs, in: Proceedings of the Eighth Working Conference on Reverse Engineering, 2001, pp. 301–307.

[35] T.M. Lavoie, E. Merlo, Levenshtein edit distance-based type III clone detection using metric trees, Technical Report, EPM-RT-2011-01, 2011.

[36] V. Levenshtein, Binary Codes Capable of Correcting Deletions, Insertions and Reversals, Soviet Physics Doklady 10 (1966) 707.

[37] Q. Li, Q. Hu, Y. Qi, S. Qi, X. Liu, P. Gao, Semi-supervised two-phase familial analysis of Android malware with normalized graph embedding, Knowledge-Based Systems 218 (2021) 106802.

[38] W. Li, Random texts exhibit Zipf's-law-like word frequency distribution, IEEE Transactions on Information Theory 38 (1992) 1842–1845.

[39] C. Liu, C. Chen, J. Han, P.S. Yu, GPLAG: Detection of software plagiarism by program dependence graph analysis, in: Proceedings of the 12th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining, KDD '06, 2006, pp. 872–881.

[40] J. Lu, X. Sun, B. Li, L. Bo, T. Zhang, BEAT: Considering question types for bug question answering via templates, Knowledge-Based Systems 225 (2021) 107098.

[41] U. Manber, Finding similar files in a large file system, in: USENIX Winter 1994 Technical Conference, 1994, pp. 1–10.

[42] V.T. Martins, D. Fonte, P.R. Henriques, D. da Cruz, Plagiarism Detection: A Tool Survey and Comparison, in: M.J.V. Pereira, et al. (Eds.), 3rd Symposium on Languages, Applications and Technologies, OpenAccess Series in Informatics (OASIcs), Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, Dagstuhl, Germany, 2014, pp. 143–158.

[43] J.J. McGregor, Backtrack search algorithm and the maximal common subgraph problem, in: Software | Practice and Experience 12, 1982, pp. 23–34.

[44] H. Murakami, Y. Higo, S. Kusumoto, A dataset of clone references with gaps, in: Proceedings of the 11th Working Conference on Mining Software Repositories, 2014, pp. 412–415.

[45] M. Newman, Power laws, Pareto distributions and Zipf's law, Contemporary Physics 46 (2005) 323–351.

[46] S.K. Pandey, A.K. Tripathi, BCV-Predictor: A bug count vector predictor of a successive version of the software system, Knowledge-Based Systems 197 (2020) 105924.

[47] T. Panker, N. Nissim, Leveraging malicious behavior traces from volatile memory using machine learning methods for trusted unknown malware detection in Linux cloud environments, Knowledge-Based Systems 226 (2021) 107095.

[48] L. Prechelt, G. Malpohl, M. Philippsen, JPlag: Finding plagiarisms among a set of programs, Technical Report, University of Karlsruhe, Department of Informatics, 2000.

[49] W. Qu, Y. Jia, M. Jiang, Pattern mining of cloned codes in software systems, Information Sciences 259 (2014) 544–554.

[50] R Core Team, R: A language and environment for statistical computing, R Foundation for Statistical Computing, Vienna, Austria, 2021. R-project.org.

[51] C. Ragkhitwetsagul, J. Krinke, D. Clark, A comparison of code similarity analysers, Empirical Software Engineering 23 (2018) 2464–2519.

[52] D. Rattan, R. Bhatia, M. Singh, Software clone detection: A systematic review, Information and Software Technology 55 (2013) 1165–1199.

[53] M. Rieger, Effective clone detection without language barriers, Ph.D. thesis, University of Bern, Switzerland, 2005.

[54] C.K. Roy, J.R. Cordy, NICAD: Accurate detection of near-miss intentional clones using flexible pretty-printing and code normalization, in: 2008 16th IEEE International Conference on Program Comprehension, 2008, pp. 172–181.

[55] C.K. Roy, J.R. Cordy, A mutation/injection-based automatic framework for evaluating code clone detection tools, in: 2009 International Conference on Software Testing, Verification, and Validation Workshops, 2009, pp. 157–166.

[56] C.K. Roy, J.R. Cordy, R. Koschke, Comparison and evaluation of code clone detection techniques and tools: A qualitative approach, Science of Computer Programming 74 (2009) 470–495.

[57] H. Sajnani, V. Saini, J. Svajlenko, C.K. Roy, C.V. Lopes, SourcererCC: Scaling code clone detection to big-code, in: 2016 IEEE/ACM 38th International Conference on Software Engineering (ICSE), 2016, pp. 1157–1168.

[58] S. Schleimer, D.S. Wilkerson, A. Aiken, Winnowing: Local algorithms for document fingerprinting, in: Proceedings of the 2003 ACM SIGMOD International Conference on Management of Data, SIGMOD '03, 2003, pp. 76–85.

[59] N. Shervashidze, P. Schweitzer, E.J. van Leeuwen, K. Mehlhorn, K.M. Borgwardt, Weisfeiler-Lehman graph kernels, Journal of Machine Learning Research 12 (2011) 2539–2561.

[60] G. Shobha, A. Rana, V. Kansal, S. Tanwar, Code clone detection—a systematic review, in: Emerging Technologies in Data Mining and Information Security, Springer Singapore, 2021, pp. 645–655.

[61] T. Smith, M. Waterman, Identification of common molecular subsequences, Journal of Molecular Biology 147 (1981) 195–197.

[62] J. Svajlenko, C. Roy, The mutation and injection framework: Evaluating clone detection tools with mutation analysis, IEEE Transactions on Software Engineering 47 (2021) 1060–1087.

[63] J. Svajlenko, C.K. Roy, BigCloneEval: A clone detection tool evaluation framework with BigCloneBench, in: 2016 IEEE International Conference on Software Maintenance and Evolution (ICSME), 2016, pp. 596–600.

[64] C. Tsallis, M.P. de Albuquerque, Are citations of scientific papers a case of nonextensivity?, The European Physical Journal B-Condensed Matter and Complex Systems 13 (2000) 777–780.

[65] A. Walker, T. Cerny, E. Song, Open-source tools and benchmarks for code-clone detection: Past, present, and future trends, SIGAPP Appl. Comput. Rev. 19 (2019) 28–39.

[66] B. Weisfeiler, A.A. Lehman, A reduction of a graph to a canonical form and an algebra arising during this reduction, Nauchno-Technicheskaya Informatsia 2 (1968).

[67] M.J. Wise, String similarity via greedy string tiling and running Karp-Rabin matching, Technical Report, Dept. of CS, University of Sydney, 1993.

Table 6: Plagiarism detection performance for the Levenshtein distance.

representation	aggregation function	parameters	θ	recall	precision	F-measure
characters	symmetric		0.30	0.941 ± 0.116	0.941 ± 0.052	$\mathbf{0.928 \pm 0.077}$
	$T_Ł$		0.01	0.784 ± 0.287	$\mathbf{0.979 \pm 0.063}$	0.866 ± 0.258
	T_m		0.31	0.937 ± 0.135	0.936 ± 0.059	$\mathbf{0.922 \pm 0.093}$
	T_p		0.11	0.942 ± 0.136	0.926 ± 0.065	0.918 ± 0.096
	M		0.33	$\mathbf{0.950 \pm 0.131}$	0.919 ± 0.067	0.917 ± 0.094
	$S_Ł$		0.67	$\mathbf{0.948 \pm 0.136}$	0.922 ± 0.067	0.918 ± 0.097
	S_m		0.44	0.909 ± 0.225	0.928 ± 0.074	0.908 ± 0.174
	S_p		0.58	0.941 ± 0.152	0.925 ± 0.065	0.917 ± 0.108
	$\varphi_{\mathbf{V}}$	$p = 2, k = 1$	0.34	$\mathbf{0.946 \pm 0.112}$	0.910 ± 0.074	0.912 ± 0.083
	Yager t-conorm (1.742553)		0.44	$\mathbf{0.949 \pm 0.105}$	0.904 ± 0.075	0.910 ± 0.079
function calls	symmetric		0.45	0.892 ± 0.150	$\mathbf{0.961 \pm 0.035}$	$\mathbf{0.917 \pm 0.099}$
	$T_Ł$		0.01	0.910 ± 0.199	0.939 ± 0.053	$\mathbf{0.918 \pm 0.148}$
	T_m		0.44	0.900 ± 0.158	0.954 ± 0.041	$\mathbf{0.919 \pm 0.107}$
	T_p		0.24	0.899 ± 0.189	0.950 ± 0.044	$\mathbf{0.919 \pm 0.136}$
	M		0.49	$\mathbf{0.926 \pm 0.181}$	0.929 ± 0.055	$\mathbf{0.919 \pm 0.129}$
	$S_Ł$		0.98	$\mathbf{0.926 \pm 0.181}$	0.929 ± 0.055	$\mathbf{0.919 \pm 0.129}$
	S_m		0.61	0.814 ± 0.263	0.845 ± 0.150	0.812 ± 0.223
	S_p		0.77	0.910 ± 0.213	0.921 ± 0.073	0.901 ± 0.163
	$\varphi_{\mathbf{V}}$	$p = 2, k = 2$	0.82	0.915 ± 0.185	0.956 ± 0.044	$\mathbf{0.922 \pm 0.129}$
	Yager t-conorm (1.591355)		0.80	0.887 ± 0.217	$\mathbf{0.961 \pm 0.046}$	0.912 ± 0.161
tokens	symmetric		0.63	0.909 ± 0.120	$\mathbf{0.966 \pm 0.032}$	$\mathbf{0.929 \pm 0.077}$
	$T_Ł$		0.29	0.916 ± 0.119	0.960 ± 0.036	$\mathbf{0.931 \pm 0.077}$
	T_m		0.63	0.899 ± 0.126	$\mathbf{0.968 \pm 0.032}$	$\mathbf{0.927 \pm 0.081}$
	T_p		0.41	0.915 ± 0.118	0.959 ± 0.037	$\mathbf{0.930 \pm 0.075}$
	M		0.64	0.922 ± 0.114	0.955 ± 0.038	$\mathbf{0.931 \pm 0.074}$
	$S_Ł$		1.00	$\mathbf{0.989 \pm 0.040}$	0.142 ± 0.047	0.247 ± 0.069
	S_m		0.71	0.916 ± 0.180	0.914 ± 0.064	0.902 ± 0.130
	S_p		0.88	0.941 ± 0.110	0.943 ± 0.047	$\mathbf{0.932 \pm 0.075}$
	$\varphi_{\mathbf{V}}$	$p = 3, k = 3$	0.90	0.931 ± 0.102	0.958 ± 0.039	$\mathbf{0.931 \pm 0.064}$
	Yager t-conorm (1.886454)		0.91	0.932 ± 0.092	0.948 ± 0.044	$\mathbf{0.928 \pm 0.059}$

Table 7: Plagiarism detection performance for the Smith–Waterman algorithm.

representation	aggregation function	parameters	w	d	v	u	θ	recall	precision	F-measure
characters	symmetric		4	1	4	1	0.16	0.934 ± 0.130	0.938 ± 0.053	0.925 ± 0.091
	$T_Ł$		4	1	1	1	0.16	0.647 ± 0.318	$\mathbf{0.982 \pm 0.082}$	0.780 ± 0.299
	T_m		4	1	4	1	0.20	0.940 ± 0.114	0.938 ± 0.053	0.927 ± 0.077
	T_p		4	1	1	1	0.13	0.894 ± 0.186	0.955 ± 0.051	0.911 ± 0.136
	M		4	1	4	1	0.24	$\mathbf{0.984 \pm 0.097}$	0.904 ± 0.067	0.930 ± 0.076
	$S_Ł$		3	1	1	1	0.44	$\mathbf{0.990 \pm 0.101}$	0.893 ± 0.072	0.925 ± 0.081
	S_m		4	1	4	1	0.33	$\mathbf{0.990 \pm 0.159}$	0.902 ± 0.075	0.925 ± 0.117
	S_p		4	1	4	1	0.47	$\mathbf{0.988 \pm 0.101}$	0.902 ± 0.068	0.930 ± 0.079
	$\varphi_{\mathbf{V}}$	$p = 1, k = 4$	4	1	3	1	0.94	$\mathbf{0.987 \pm 0.076}$	0.933 ± 0.061	$\mathbf{0.948 \pm 0.061}$
	Yager t-conorm (0.437134)		4	1	3	1	0.86	$\mathbf{0.989 \pm 0.059}$	0.917 ± 0.070	$\mathbf{0.941 \pm 0.056}$
function calls	symmetric		4	1	1	1	0.31	0.908 ± 0.133	0.951 ± 0.039	0.919 ± 0.087
	$T_Ł$		4	1	1	1	0.13	0.837 ± 0.233	$\mathbf{0.962 \pm 0.044}$	0.892 ± 0.185
	T_m		4	1	1	1	0.43	0.914 ± 0.120	0.942 ± 0.044	0.921 ± 0.079
	T_p		4	1	1	1	0.22	0.923 ± 0.112	0.936 ± 0.045	0.924 ± 0.074
	M		4	1	1	1	0.45	0.959 ± 0.098	0.905 ± 0.058	0.921 ± 0.068
	$S_Ł$		3	1	1	1	0.86	0.943 ± 0.144	0.909 ± 0.061	0.904 ± 0.098
	S_p		4	1	1	1	0.66	0.827 ± 0.253	0.880 ± 0.124	0.842 ± 0.210
	S_m		4	1	1	1	0.70	$\mathbf{0.969 \pm 0.096}$	0.882 ± 0.081	0.898 ± 0.076
	$\varphi_{\mathbf{V}}$	$p = 1, k = 1$	4	1	1	1	0.81	0.946 ± 0.112	0.955 ± 0.040	$\mathbf{0.938 \pm 0.071}$
	Yager t-conorm (1.443944)		4	1	1	1	0.79	0.933 ± 0.156	$\mathbf{0.959 \pm 0.044}$	$\mathbf{0.933 \pm 0.105}$
tokens	symmetric		4	3	4	1	0.39	0.908 ± 0.126	$\mathbf{0.974 \pm 0.027}$	0.938 ± 0.083
	$T_Ł$		4	4	4	1	0.15	0.920 ± 0.113	0.962 ± 0.032	0.939 ± 0.075
	T_m		4	4	4	1	0.50	0.914 ± 0.123	0.971 ± 0.032	0.935 ± 0.080
	T_p		4	4	4	1	0.30	0.926 ± 0.111	0.958 ± 0.035	0.937 ± 0.072
	M		3	3	3	1	0.52	0.950 ± 0.090	0.940 ± 0.048	0.938 ± 0.064
	$S_Ł$		3	4	4	1	0.84	$\mathbf{0.968 \pm 0.076}$	0.920 ± 0.051	0.935 ± 0.056
	S_p		3	4	3	1	0.60	0.957 ± 0.140	0.906 ± 0.065	0.910 ± 0.098
	S_m		4	4	4	1	0.81	$\mathbf{0.963 \pm 0.081}$	0.931 ± 0.050	0.938 ± 0.059
	$\varphi_{\mathbf{V}}$	$p = 2, k = 2$	3	4	4	1	0.80	$\mathbf{0.961 \pm 0.082}$	0.965 ± 0.035	$\mathbf{0.954 \pm 0.051}$
	Yager t-conorm(1.716247)		3	4	4	1	0.73	$\mathbf{0.961 \pm 0.081}$	0.968 ± 0.034	$\mathbf{0.951 \pm 0.054}$

Table 8: Plagiarism detection performance for greedy string tiling.

representation	aggregation function	parameters	d	θ	recall	precision	F-measure
characters	symmetric		4	0.26	0.945 ± 0.087	0.950 ± 0.057	0.938 ± 0.063
	$T_Ł$		2	0.23	0.885 ± 0.174	$\mathbf{0.958 \pm 0.051}$	0.908 ± 0.127
	T_m		7	0.21	0.951 ± 0.080	0.941 ± 0.063	0.934 ± 0.062
	T_p		3	0.25	0.931 ± 0.116	$\mathbf{0.962 \pm 0.048}$	0.933 ± 0.079
	M		6	0.44	0.953 ± 0.111	$\mathbf{0.955 \pm 0.050}$	0.946 ± 0.081
	$S_Ł$		7	0.78	0.978 ± 0.073	0.930 ± 0.062	0.943 ± 0.062
	S_m		7	0.51	$\mathbf{0.991 \pm 0.129}$	0.904 ± 0.074	0.928 ± 0.099
	S_p		7	0.71	0.961 ± 0.122	0.944 ± 0.056	0.945 ± 0.091
	$\varphi_{\mathbf{V}}$	$p = 2, k = 1$	7	0.56	$\mathbf{0.996 \pm 0.067}$	0.939 ± 0.061	$\mathbf{0.956 \pm 0.060}$
	Yager t-conorm (1.573940)		7	0.56	$\mathbf{0.994 \pm 0.056}$	0.938 ± 0.063	$\mathbf{0.955 \pm 0.054}$
function calls	symmetric		1	0.49	0.909 ± 0.107	0.970 ± 0.039	0.933 ± 0.071
	$T_Ł$		1	0.33	0.930 ± 0.092	0.961 ± 0.042	0.937 ± 0.062
	T_m		1	0.65	0.870 ± 0.131	$\mathbf{0.984 \pm 0.029}$	0.921 ± 0.087
	T_p		1	0.45	0.916 ± 0.103	0.969 ± 0.039	0.936 ± 0.069
	M		1	0.65	0.945 ± 0.078	0.939 ± 0.052	0.935 ± 0.057
	$S_Ł$		4	0.75	0.937 ± 0.217	0.897 ± 0.076	0.896 ± 0.156
	S_m		4	0.52	0.872 ± 0.269	0.839 ± 0.124	0.828 ± 0.210
	S_p		3	0.74	0.929 ± 0.227	0.895 ± 0.096	0.889 ± 0.172
	$\varphi_{\mathbf{V}}$	$p = 1, k = 1$	2	0.91	$\mathbf{0.965 \pm 0.085}$	0.956 ± 0.043	$\mathbf{0.950 \pm 0.055}$
	Yager t-conorm (1.682680)		2	0.85	$\mathbf{0.973 \pm 0.133}$	0.942 ± 0.049	$\mathbf{0.942 \pm 0.087}$
tokens	symmetric		13	0.27	0.945 ± 0.077	0.951 ± 0.042	0.939 ± 0.052
	$T_Ł$		8	0.28	0.925 ± 0.095	$\mathbf{0.964 \pm 0.036}$	0.937 ± 0.063
	T_m		13	0.34	0.929 ± 0.091	0.957 ± 0.042	0.937 ± 0.060
	T_p		11	0.26	0.934 ± 0.088	$\mathbf{0.964 \pm 0.036}$	0.941 ± 0.058
	M		11	0.51	0.974 ± 0.064	0.939 ± 0.049	0.944 ± 0.048
	$S_Ł$		16	0.74	$\mathbf{0.990 \pm 0.074}$	0.917 ± 0.059	0.934 ± 0.055
	S_m		19	0.48	0.944 ± 0.193	0.895 ± 0.075	0.895 ± 0.135
	S_p		14	0.75	0.975 ± 0.128	0.929 ± 0.056	0.930 ± 0.084
	$\varphi_{\mathbf{V}}$	$p = 1, k = 1$	14	0.75	0.984 ± 0.064	$\mathbf{0.964 \pm 0.039}$	$\mathbf{0.961 \pm 0.043}$
	Yager t-conorm (1.433622)		14	0.75	0.981 ± 0.092	$\mathbf{0.962 \pm 0.040}$	$\mathbf{0.957 \pm 0.058}$

Table 9: Plagiarism detection performance for q-grams.

representation	aggregation function	parameters	q	θ	recall	precision	F-measure
characters	symmetric		3	0.41	0.952 ± 0.112	0.955 ± 0.045	0.942 ± 0.075
	$T_Ł$		2	0.14	0.906 ± 0.199	$\mathbf{0.962 \pm 0.047}$	0.923 ± 0.148
	T_m		3	0.32	0.960 ± 0.085	0.946 ± 0.051	0.941 ± 0.059
	T_p		3	0.16	0.965 ± 0.094	0.944 ± 0.052	0.941 ± 0.066
	M		3	0.39	0.978 ± 0.087	0.931 ± 0.057	0.941 ± 0.065
	$S_Ł$		3	0.78	0.978 ± 0.087	0.931 ± 0.057	0.941 ± 0.065
	S_m		7	0.24	$\mathbf{0.999 \pm 0.083}$	0.879 ± 0.078	0.922 ± 0.074
	S_p		4	0.63	0.980 ± 0.108	0.925 ± 0.059	0.938 ± 0.080
	$\varphi_{\mathbf{V}}$	$p = 1, k = 1$	6	0.68	$\mathbf{0.998 \pm 0.049}$	0.917 ± 0.070	$\mathbf{0.947 \pm 0.055}$
	Aczel–Alsina t-conorm (0.390426)		6	0.68	0.991 ± 0.038	0.910 ± 0.073	0.940 ± 0.051
function calls	symmetric		1	0.72	0.925 ± 0.103	$\mathbf{0.981 \pm 0.022}$	0.948 ± 0.065
	$T_Ł$		1	0.39	0.943 ± 0.091	0.974 ± 0.027	0.952 ± 0.058
	T_m		1	0.66	0.910 ± 0.114	$\mathbf{0.982 \pm 0.022}$	0.940 ± 0.073
	T_p		1	0.45	0.948 ± 0.088	0.969 ± 0.030	0.950 ± 0.057
	M		1	0.70	0.943 ± 0.091	0.974 ± 0.027	0.952 ± 0.058
	$S_Ł$		2	0.84	$\mathbf{0.975 \pm 0.119}$	0.918 ± 0.053	0.929 ± 0.078
	S_m		4	0.31	0.780 ± 0.280	0.858 ± 0.129	0.800 ± 0.230
	S_p		2	0.74	0.937 ± 0.197	0.917 ± 0.061	0.905 ± 0.138
	$\varphi_{\mathbf{V}}$	$p = 3, k = 1$	1	0.97	$\mathbf{0.979 \pm 0.062}$	0.957 ± 0.045	$\mathbf{0.956 \pm 0.045}$
	Yager t-conorm (2.702127)		1	0.96	0.964 ± 0.084	0.960 ± 0.045	0.950 ± 0.056
tokens	symmetric		5	0.63	0.938 ± 0.095	$\mathbf{0.973 \pm 0.026}$	0.950 ± 0.060
	$T_Ł$		4	0.33	0.951 ± 0.083	0.970 ± 0.030	0.953 ± 0.054
	T_m		8	0.42	0.931 ± 0.102	$\mathbf{0.974 \pm 0.024}$	0.948 ± 0.065
	T_p		4	0.44	0.942 ± 0.089	$\mathbf{0.974 \pm 0.028}$	0.951 ± 0.057
	M		7	0.51	0.973 ± 0.073	0.947 ± 0.038	0.952 ± 0.049
	$S_Ł$		8	0.89	$\mathbf{0.981 \pm 0.076}$	0.938 ± 0.043	0.948 ± 0.052
	S_m		16	0.32	0.943 ± 0.173	0.870 ± 0.078	0.886 ± 0.124
	S_p		11	0.60	0.976 ± 0.118	0.918 ± 0.056	0.928 ± 0.081
	$\varphi_{\mathbf{V}}$	$p = 1, k = 1$	8	0.78	$\mathbf{0.980 \pm 0.069}$	0.967 ± 0.032	$\mathbf{0.964 \pm 0.045}$
	Yager t-conorm (1.534131)		8	0.81	0.957 ± 0.134	0.966 ± 0.035	0.948 ± 0.088

Table 10: Plagiarism detection performance for the PDG-based algorithms.

comparer	aggregation function	parameters	h	θ	recall	precision	F-measure
Weisfeiler–Lehman	symmetric		3	0.71	$\mathbf{0.932 \pm 0.111}$	0.978 ± 0.027	$\mathbf{0.946 \pm 0.071}$
	$T_Ł$		2	0.63	0.923 ± 0.114	$\mathbf{0.984 \pm 0.023}$	$\mathbf{0.946 \pm 0.074}$
	T_m		2	0.75	0.916 ± 0.121	$\mathbf{0.985 \pm 0.022}$	$\mathbf{0.943 \pm 0.078}$
	T_p		2	0.58	$\mathbf{0.936 \pm 0.109}$	0.977 ± 0.028	$\mathbf{0.948 \pm 0.069}$
	M		2	0.82	0.923 ± 0.114	0.984 ± 0.023	$\mathbf{0.946 \pm 0.074}$
	$S_Ł$		20	0.85	0.916 ± 0.148	0.581 ± 0.150	0.692 ± 0.135
	S_m		3	0.84	0.881 ± 0.146	0.679 ± 0.153	0.748 ± 0.133
	S_p		6	0.85	0.923 ± 0.126	0.752 ± 0.139	0.811 ± 0.117
	$\varphi_{\mathbf{V}}$	$p = 1, k = 1$	1	0.89	0.862 ± 0.146	0.978 ± 0.029	0.910 ± 0.098
	power mean (0.804382)		1	0.90	0.873 ± 0.145	0.975 ± 0.032	0.911 ± 0.098
SimilaR	symmetric		2	0.63	0.957 ± 0.089	0.988 ± 0.017	$\mathbf{0.967 \pm 0.055}$
	$T_Ł$		2	0.39	0.956 ± 0.086	0.992 ± 0.014	$\mathbf{0.970 \pm 0.053}$
	T_m		2	0.66	0.941 ± 0.108	$\mathbf{0.993 \pm 0.013}$	0.963 ± 0.068
	T_p		2	0.44	0.957 ± 0.088	0.990 ± 0.016	$\mathbf{0.968 \pm 0.054}$
	M		2	0.71	0.956 ± 0.086	0.992 ± 0.014	$\mathbf{0.970 \pm 0.053}$
	$S_Ł$		6	0.82	$\mathbf{0.988 \pm 0.038}$	0.645 ± 0.116	0.771 ± 0.090
	S_m		2	0.83	0.965 ± 0.068	0.961 ± 0.041	0.955 ± 0.048
	S_p		3	0.87	0.981 ± 0.048	0.946 ± 0.046	0.956 ± 0.040
	$\varphi_{\mathbf{V}}$	$p = 3, k = 1$	2	0.93	0.953 ± 0.076	0.981 ± 0.025	0.961 ± 0.048
	Yager t-conorm (2.660945)		2	0.73	0.919 ± 0.110	0.991 ± 0.017	0.949 ± 0.071