Automated Clone Validation And Classification

Kasper Engelen
kasper.engelen@student.uantwerpen.be

University of Antwerp

Abstract

In the past, much research has been performed on the detection and the nature of code clones. This research has resulted in a number of benchmarks, which are used to evaluate automatic clone detection. In this paper we attempt to automate the construction of such benchmarks, by investigating automatic validation (filtering out false positives) and classification (assigning code clones a type). We do this by constructing an automatic tool, and evaluating it using an existing benchmark. We discovered that automating the validation using the chosen technique is not feasible, achieving a maximum precision of 40%. We do show, however, that automatic classification is feasible with precision and recall ranging between 50% and 80%.

1 Introduction

A code clone is defined as a segment of copied or duplicated code. A clone pair consists of two segments, which have been determined to have some similarity between them [18]. There are many causes of code cloning, which may be intentional (i.e. copy-paste) or even unintentional such as two developers accidentally writing the same code [18].

There is a good incentive for detecting clones since code clones may propagate bugs and incur increased maintenance costs [18]. Even in situations in which the cloning is intentional, detection may bring more insights which can be used for long term maintenance of such clones [12].

Clone detection tools may incorrectly mark a code fragment as a clone, which is then called a false positive [14]. The process of deciding whether or not a previously marked code fragment is a false positive is called clone *validation* [14].

Clones belong to one of four clone types, the first three are based on syntactic similarity, while the fourth one is concerned with semantic similarity [18]. Clone *classification* is the process in which a detected clone pair, with an unknown clone type, is analysed and the actual clone type is determined [9].

A primary motivation for automating clone validation and classification is the construction of clone benchmarks. Clone benchmarks are used to evaluate the performance of clone detection tools [18]. Manually constructing such benchmarks is effort intensive [4, 17], and judges can be unreliable and biased [17]. Roy and Cordy [17] note that automatic validation and classification can greatly improve the construction of benchmarks, and efforts towards automation have been undertaken in the past [17, 19]. Additionally, benchmark construction requires a precise and consistent definition [24], which can be made possible by using an automatic tool.

Aside from constructing benchmarks, there are secondary benefits. Li et al. [13] reported that the detection of exact clones can lead to better bug fixing, since the absence of renamed identifiers can indicate negligence on behalf of the programmer. The classification of cloned code segments can also be used to discover re-engineering opportunities [2].

In this paper we propose techniques to automatically validate and classify clones. The validation will be done based on clone density [6], and the clone classification will focus on syntactic similarity. The proposed techniques are not tied to a specific clone detection tool, and are as such tool-agnostic. Some other validation and classification tools are also tool-agnostic [14, 15]. We implemented the proposed techniques in a tool, which was then applied to a benchmarking dataset using different configurations. Finally, we used the obtained results to evaluate the effectiveness of the proposed techniques.

2 Related work

The literature concerning code clones describes multiple clone types [18]. We list the exact definitions of the first three syntactic clone types:

- Type-1: The program text is identical, save for differences in comments, whitespace, or newlines.
- Type-2: The program text may additionally contain differences in identifiers and literals.
- Type-3: The program text may additionally contain inserted or deleted sub-segments, which are called "gaps".

The existing literature concerning the automatisation of clone classification is limited. We list two papers that touch upon the topic.

In Farima et al. [9] an automated clone classifier is discussed. Classification is performed by first removing comments, whitespace, and newlines and then taking the MD5 hash. If the hashes are identical, the clone is determined to be of Type-1. If this is not the case then additionally the identifiers and literals are parameterised, and once again the MD5 hash is taken. If the hashes correspond this time, the clone is decided to be of Type-2. If not, it is assumed to be of Type-3.

Balazinska et al. [3] present a classification tool called SMC. This tool uses a different classification scheme which consists of 18 so-called *categories*. The classification scheme has been designed to aid in the discovery of re-engineering opportunities.

Since the construction of benchmarks is our main motivation, and since current benchmarks can help to construct validation and classification tools [17], we will list some previously constructed clone benchmarks.

One of the first and most notable of the clone benchmarks is the one constructed by Bellon et al. [4]. In Bellon's experiment multiple clone detection tools were used to process a number of C and Java programs. Of the clones reported by these tools, some 2% were manually validated. This manual validation effort took 77 hours.

A bigger benchmark was constructed for the BigCloneBench [22] project. In contrast to Bellon's benchmark, this one was not constructed using clone detectors. Instead, the authors made a selection of common functionalities (file copy, bubble sort, etc) and mined a big data repository for clones of these functionalities. The detected clones were then manually validated and classified. After 216 man-hours, this resulted in 6 million clones. In a later paper [23] the amount of clones had grown to 8 million.

In 2021, van Bladel et al. [24] published a validated and classified clone dataset. This dataset contains clones written in the Java programming language. These were obtained using detection tools, by applying them on test and production code from multiple projects: Apache Commons Math, Google Guava, Elastic Search, the Spring Framework, and Java Design Patterns.

We looked for techniques that can be used to filter out false positives. Some clone detection tools contain a filtering step in which false positives are removed [6, 13, 16].

In Ducasse et al. [6] there is mention of a number of techniques to filter out false positive clones, among which is the *clone density*. This is defined as the number of matched elements in the comparison sequence divided by the total length of the comparison sequence. The motivation is that stretched-out clones with repetitive parts (e.g. a switch-statement) would be filtered out since they have a low density. Using such a density threshold also is not sensitive to the size of the clone.

Other measures exist in the literature, such as introducing a minimum clone size [6, 16, 13], or a max gap-size for Type-3 clones [6, 13]. In this feasibility study we only investigate the clone density. We decided to use this measure since it is not sensitive to the clone size. In contrast to a minimum clone size, this will not filter out any Type-1 or Type-2 clones, since they have a density equal to 1 by definition. We also expect that there is a difference between the densities of Type-3 clones and the densities of the false positives.

In order to come up with techniques that are capable of accurately mapping a clone pair onto a clone type, we also analysed other existing clone detection tools. For each detection tool, we analysed the different aspects [18, 16] of those tools: source unit size, comparison unit size, transformations, and comparison algorithm.

The source unit is the size of the code fragment that is accepted by the classification tool. Such a source unit [18, 16] can have multiple sizes: entire files, classes, methods, individual statements, etc. Rieger [16] defines the source unit as the largest source fragment that can be in a clone relation with another fragment. Multiple benchmarks [17, 24] exist which use the method-level source unit. Since it is our aim to automate the construction of such benchmarks, we decided to use the method-level source unit.

The comparison unit is the smallest entity that is considered by the classification algorithm. The size of the unit determines the comparison granularity [18, 16]. The source unit is made up of comparison units [16]. In Roy and Cordy [20], and in Ducasse et al. [7] a line-based comparison unit is considered. This is also the granularity employed by van Bladel et al. [24] when classifying his benchmark dataset. Smaller, more fine-grained comparison units are also possible. The CCFinder tool [11] works on the level of individual tokens. The motivation being that this allows for the detection of clones with a different textual layout, and that this is very light-weight since it does not require parsing. It is also possible to work with syntax trees, which are then serialised using pre-order [8] or post-order [10] traversal.

Multiple sources [18, 5, 7, 11, 16, 20] report the use of pre-processing techniques, which means that certain transformations are applied to the comparison units before running the comparison algorithm. The definition of the three syntactic clone types [18] mentions the existence of transformations in the source code, such as the renaming of identifiers, variations in comments and whitespace, etc.

The removal of whitespace and comments is applied by most text-based clone detection tools in the literature [18], and is explored used by Ducasse et al. [7] and Rieger [16]. This transformation is relevant since all clone type definitions allow for differences in comments and whitespace.

Another important transformation is the parameterisation or normalisation of identifiers and literals [1, 7, 11, 16, 20]. This means that all identifiers and literals are replaced by a common symbol, causing differences in identifiers and literals to be ignored by the comparison algorithm. Again, this is relevant since the second and third clone type allow for identifiers and literals to be renamed [18]. This technique is explored in-depth by Ducasse et al. [7].

Pretty-printing is a transformation that is applied to the layout of the program text, and is useful when utilising line-based comparison units [5, 16, 20]. The pretty printing technique will insert linebreaks at standardised points [16], thus normalising the layout of the program text. This makes the comparison of clones easier since it removes formatting differences [20]. It makes it also possible to fine-tune the granularity of the line-based comparison unit [5].

The comparison algorithm decides how clone detectors compare code fragments in order to determine whether or not they are clones. There exist many such algorithms, which differ in working and also in complexity. Baker [1], and CCFinder [11] make use of suffix trees. On the other hand, Cordy et al. [5] make use of the Unix *diff* tool. There is also mention of algorithms that solve the longest-common subsequence problem in Balazinska et al. [3], and Yang [26]. The LCS algorithm is very similar to the Unix *diff* tool [20].

The LCS algorithm has the advantages of being simple to implement and to have an acceptable complexity of $O(n^2)$ [2]. The algorithm is also capable of matching two sequences of arbitrary type, as long as the difference between two elements of that type is defined [2].

There are clone detection tools that take a different approach by calculating various metrics about source code fragments. This means that for each such fragment, be it a class or even a single statement, a vector is produced. The produced vectors are then compared in order to detect clones [18]. However, such a metric does not offer any explanation about the nature of the clone [3]. For example, a method-level metric does not offer insight into which statements were cloned and which were not. Due to this lack of information, this paper does not consider metric-based techniques for clone classification.

In the clone detection literature there is also mention of techniques that make use of the Program Dependecy Graph (PDG) [18, 21]. This paper does not consider such techniques since they have high computational complexities [18], and they are geared towards being semantics-aware [21] while this paper does not consider the semantic type-4 clones.

3 Experimental setup

3.1 Automated tool

In this section we will explain the workings of the tool that we developed. The library that was used for tokenisation and parsing was Javaparser [25]. The tool consists of multiple steps that are applied one after another.

Step 1: In order to classify a clone pair, the tool will first read the raw source code of the two code fragments. These source fragments are then converted into a code representation (Section 3.1.1). This results in two sequences of comparison units, one sequence for each fragment.

Step 2: These two sequences are then analysed by the comparison algorithm (Section 3.1.2). This algorithm will try to find matches between elements of the two sequences. The result is a mapping, that maps each element of one sequence to at most one element of the other sequence. Such a match between two elements may be exact or parameterised.

Step 3: The clone pair, which now consists of two sequences of comparison units between which there exists a mapping, is then passed to the validation algorithm (Section 3.1.3). This algorithm then determines, based on the clone density, whether or not the clone is a false positive. If the clone is a false positive, this is reported to the user and the tool stops.

Step 4: If the clone is a valid clone, a classification algorithm (Section 3.1.4) is applied. This algorithm will determine the type of the clone pair. This type is then reported to the user.

3.1.1 Code representation

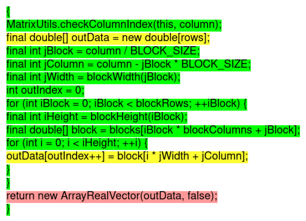
We constructed so called code representations by chaining the different types of comparison units and pre-processing techniques together. This resulted in four such representations:

- Line-based representation: The whitespace and comments are removed from the code, all the identifiers and constants are parameterised, the code is then pretty printed. The resulting code is read line-by line.
- Token-based representation: The code is tokenised, the whitespace and comment tokens are filtered out, and the identifier and constant tokens are parameterised.
- Pre-order-based representation: The code is parsed and an AST is constructed. This AST is then traversed in pre-order. The resulting nodes are then parameterised by replacing the identifier nodes and constants by a universal symbol.
- Post-order-based representation: identical to the pre-order one, except that the AST is traversed in post-order.

3.1.2 Comparison algorithm

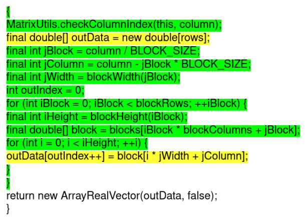
In order to compare two code fragments, we employ an algorithm that is similar to the one used by Balazinska et al. [3], and Yang [26]. Such an algorithm solves the longest common subsequence (LCS) problem [26]. We decided to use this algorithm because of the simplicity, the fact that it accepts sequences of elements, and the property that it finds the minimal amount of differences between two pieces of code.

When using the comparison algorithm in combination with the line-based comparison unit, we applied some additional post-processing. This consisted of selectively filtering some closing braces "}" from the matched lines. In order to determine which braces needed to be filtered, we checked whether they were directly or indirectly adjacent to a non-brace matched line (i.e. an ordinary line). This is to prevent such braces from needlessly stretching out the clone by introducing matches on their own. This is a form of noise reduction that was done to prevent Type-1 and Type-2 clones from being misclassified as Type-3 clones.



```
{
MatrixUtils.checkColumnIndex(this, column);
final double[] outData = new double[rows];
final int jBlock = column / BLOCK_SIZE;
final int jColumn = column - jBlock * BLOCK_SIZE;
final int jWidth = blockWidth(jBlock);
int outIndex = 0;
for (int iBlock = 0; iBlock < blockRows; ++iBlock) {
final int iHeight = blockHeight(iBlock);
final double[] block = blocks[iBlock * blockColumns + jBlock];
for (int i = 0; i < iHeight; ++i) {
outData[outIndex++] = block[i * jWidth + jColumn];
}
}
return new ArrayRealVector(outData, false);
}
```

(a) Before



```
{
MatrixUtils.checkColumnIndex(this, column);
final double[] outData = new double[rows];
final int jBlock = column / BLOCK_SIZE;
final int jColumn = column - jBlock * BLOCK_SIZE;
final int jWidth = blockWidth(jBlock);
int outIndex = 0;
for (int iBlock = 0; iBlock < blockRows; ++iBlock) {
final int iHeight = blockHeight(iBlock);
final double[] block = blocks[iBlock * blockColumns + jBlock];
for (int i = 0; i < iHeight; ++i) {
outData[outIndex++] = block[i * jWidth + jColumn];
}
}
return new ArrayRealVector(outData, false);
}
```

(b) After

Figure 1: An example of a code fragment before and after applying the brace filter.

In Fig. 1 an example of the brace filtering method is illustrated. The code originates from the Apache Commons Math library. In the first illustration (Fig. 1a) the filter was not applied, resulting in a gap on the second last line. The second illustration (Fig. 1b) shows that the last brace is filtered because it is not adjacent to a matched line, and would otherwise introduce a match of its own.

3.1.3 Validation algorithm

The two clone fragments are then separately *validated* in order to filter out false positives produced by clone detection tools. This is done by first determining the clone size, which is the amount of comparison units between the first and last matched unit. Next, the clone density is determined by counting the amount of matched comparison units and dividing by the clone size. This results in the fraction of matched comparison units.

If the clone density of either fragment is below a certain previously defined threshold, the clone is marked as a false positive. If the clone size is zero, it should also be marked as a false positive.

3.1.4 Classification algorithm

Once the clone pair is validated, it still needs to be classified as Type-1, Type-2, or Type-3. Again, we first take the two fragments separately. For each fragment we look at which comparison units are matched. We also look if these matches are exact or parameterised.

If all comparison units between the first and last matched comparison unit form an exact match, the fragment will be marked as Type-1. If there are one or more parameterised matches, it is of Type-2. Finally, if there are non-matched comparison units between the first and last matched unit, the fragment is marked as Type-3. In order to obtain the type of the clone pair, we take the least strict of the types of the two fragments. For example, if one fragment is of Type-1, and the other fragment is of Type-3, the clone pair will be of Type-3.

Note that this corresponds to the exact definition of the three clone types [18]. This approach is also similar to the one applied by van Bladel et al. [24].

3.2 Dataset

In order to benchmark the tool, we used van Bladel's dataset [24]. We decided to use this benchmark since it was readily available in the flexible XML format. The different amount of clones per clone type in the benchmark are listed in Table 1.

Label	Type-1	Type-2	Type-3	FP
Amount	2760	16847	20175	1011

Table 1: The amount of clones present in the dataset, grouped by label.

3.3 Experimental protocol

We conduct experiments by applying the tool to the benchmarking dataset, and evaluating the output of the tool. We ran the tool multiple times, each time with a different setup. In order to obtain different setups we used different types of code representation, and different values for the minimum clone density threshold. The setups comprised all possible combinations of the four code representations, and different clone density thresholds between 0 and 1 with increments of 0.1.

3.4 Evaluation Metrics

In order to evaluate the accuracy of the validation and classification tool, we construct a multi-class confusion matrix. The classes are the three clone types Type-1, Type-2, and Type-3, and the additional False Positive category. This confusion matrix is filled by comparing the output of the tool, with the label stored in the labelled benchmarking dataset.

Based on the multi-class confusion matrix, one binary confusion matrix is constructed for each class. This binary confusion matrix is then used to calculate precision and recall. This allows us to determine the effectiveness of the validation for false positive clones, and the effectiveness of the classification for each clone type.

3.5 Research questions

RQ1 *"Is automated clone validation feasible using clone density and if so, what is the optimal threshold?"*
Motivation Clone detectors sometimes give false positives as their output, it is important to filter out such clones [16, 18]. In the literature [14, 9, 15] other attempts are made to detect such false positives. We want to see if we can use a minimum clone density threshold to detect false positive clones. In order to make good use of such a threshold, we must also find an optimal value for it.

RQ2 *"Is automated clone classification feasible?"*
Motivation We recognise that clone type detection is important, both for researchers and practitioners. We therefore aim to introduce techniques to automate this classification process. We must determine whether or not this is feasible by evaluating the proposed techniques.

RQ3 *"What is the best code representation for automated clone classification?"*
Motivation In this paper we introduce multiple code representations. It is of importance to show the effectiveness of each one, and to determine the one with highest precision and recall.

4 Results

In this section we will review the results of the experiments, and answer our research questions.

4.1 Clone validation

We ran the tool multiple times, using multiple combinations of code representations and density thresholds. The results for the line-based representation can be seen in Fig. 2, the token-based results are plotted in Fig. 3, and Figures 4 and 5 display the pre-order and post-order results respectively. On the left side the precision is plotted, while the right side contains the recall.

We can see that the precision for detecting false positives is low. The precision never reaches higher than 40%, meaning that 60% of the detected false positive clones, actually are not false positives. The recall is at times almost 75%, but this is only the case as as the minimum density goes toward 1. We also see that as the minimum density goes toward 1, the Type-3 precision and recall go toward zero.

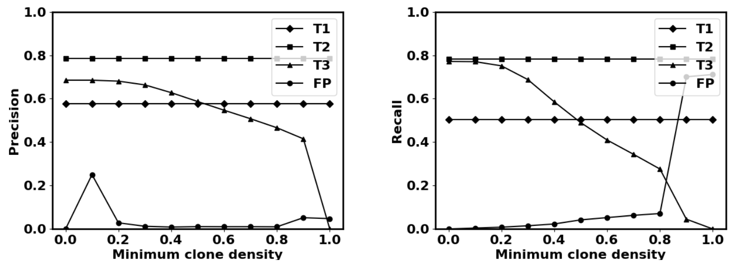


Figure 2: Precision and recall for line-based representation.

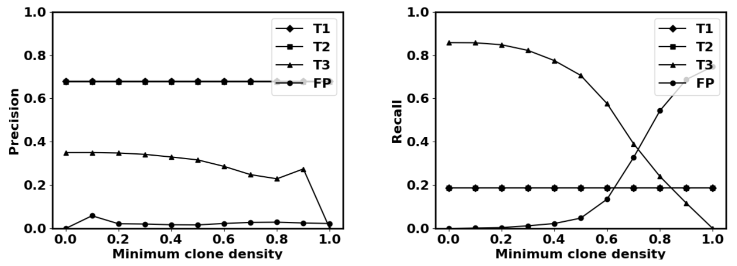


Figure 3: Precision and recall for token-based representation.

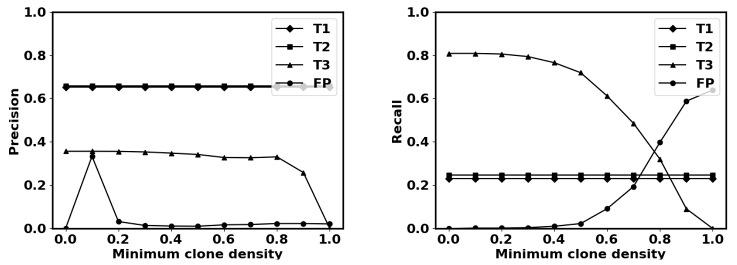


Figure 4: Precision and recall for pre-order tree traversal-based representation.

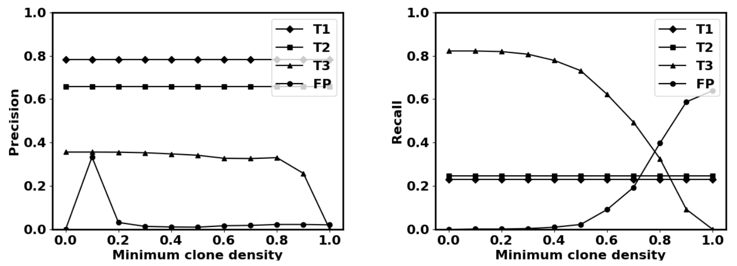


Figure 5: Precision and recall for post-order tree traversal-based representation.

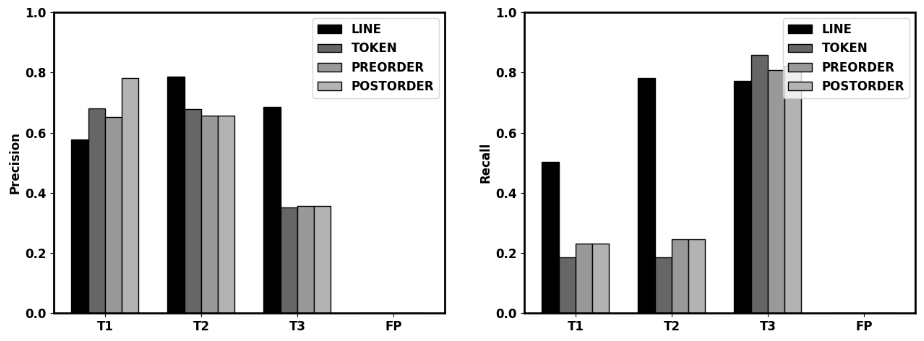


Figure 6: Precision and recall for the different types of comparison unit, per clone type.

The reason why Type-3 recall goes down as the minimum density goes up, can be explained by the fact that there are gaps in Type-3 clones, which leads to lower density in such clones. If the density threshold is increased, less Type-3 clones will meet this threshold and thus be misclassified as false positives. Contrary to our expectation, there is an overlap between the density of Type-3 clones and the density of false positives. The results for Type-1 and Type-2 clones are not affected by the different density settings, since they have a density equal to 1 by definition.

Since the false positive detection offers low precision and recall in general, and since Type-3 recall decreases as the minimum density goes toward 1, we conclude that a minimum density value of 0 is optimal. The unfortunate consequence is that setting the minimum density to 0 will effectively disable the detection of false positives based on clone density.

Since we obtained a maximum precision of 40%, we conclude our findings by answering **RQ1**: clone validation using a clone density threshold is not feasible, and as such the optimal clone density is zero.

4.2 Clone classification

In order to determine whether or not clone classification is feasible, we ran the classification tool using multiple code representations. The clone density threshold was set to zero. Afterwards we calculated the precision and recall per clone type and per code representation. This way we can compare the different code representations. This was then plotted in Fig. 6. The precision is plotted on the left, and recall on the right.

We can see that the results vary, between precision and recall, between clone types, and between code representations. It is also visible that the fluctuations are very limited among the "token-like" code representations, namely the token-based, and tree-based ones. The difference between the "token-like" code representations on one hand, and the line-based code representation on the other hand is remarkable.

We can explain these differences by the fact that the token-based and tree-based representations work on the level of individual identifiers and constants, while the line-based representation works more on the level of complete statements. The latter thus puts identifiers in their context. Additionally all of this is parameterised, leading to identifiers being indistinguishable from each other. This has as a result that the comparison algorithm matches completely unrelated identifiers in different parts of the code segment. Also, this makes it possible that parts of a statement are matched separately with different other statements. All of this introduces gaps that should not be there. These gaps then lead to Type-1 and Type-2 clones being misclassified as Type-3. Which has as a result that recall for Type-1 and Type-2, and precision for Type-3 is lower, as can be seen in the graphs.

We have obtained precision and recall ranging between 50% and 80% for all three clone types, using the line-based code representation. The other code representations have difficulties. The answer to **RQ2** is thus that clone classification is feasible. Additionally, **RQ3** can be answered by saying that the line-based representation is optimal.

5 Conclusions

In this paper we proposed techniques to facilitate the automatic validation and classification of clones. We derived these techniques from existing clone detection literature, and we implemented them in order to evaluate their performance. Their performance was determined by applying them to an existing validated and classified benchmarking dataset. Afterwards we calculated precision and recall, both for the validation and classification of clones.

For the clone validation, we conducted experiments by setting different thresholds for the clone density. The results showed that using such thresholds is not viable, with precision never exceeding 40% and most of the time being lower than 10%. Recall higher than 60% was only attainable with thresholds for which Type-3 precision and recall went towards zero.

The clone classification precision and recall vary between 50% and 80%. The token-based and tree-based representations suffered from problems, due to which Type-1 and Type-2 clones were misclassified as Type-3 clones. This weakened the Type-1 and Type-2 recall, as well as Type-3 precision. We concluded that automated classification is feasible, and that the line-based code representation is optimal.

References

[1] Brenda S. Baker. A theory of parameterized pattern matching: Algorithms and applications. In *Proceedings of the Twenty-Fifth Annual ACM Symposium on Theory of Computing*, STOC '93, page 71–80, New York, NY, USA, 1993. Association for Computing Machinery.

[2] M. Balazinska, E. Merlo, M. Dagenais, B. Lague, and K. Kontogiannis. Advanced clone-analysis to support object-oriented system refactoring. In *Proceedings Seventh Working Conference on Reverse Engineering*, pages 98–107, 2000.

[3] Magdalena Balazinska, Ettore Merlo, Michel Dagenais, and Bruno Lague. Measuring clone based reengineering opportunities. 11 2000.

[4] Stefan Bellon, Rainer Koschke, Giuliano Antoniol, Jens Krinke, and Ettore Merlo. Comparison and evaluation of clone detection tools. *IEEE TSE*, 33(9):577–591, 2007.

[5] James Cordy, Thomas Dean, and Nikita Synytskyy. Practical language-independent detection of near-miss clones. pages 1–12, 01 2004.

[6] Stéphane Ducasse, O. Nierstrasz, and Matthias Rieger. Lightweight detection of duplicated code — a language-independent approach 1. 2004.

[7] Stéphane Ducasse, Oscar Nierstrasz, and Matthias Rieger. On the effectiveness of clone detection by string matching. *Journal of Software Maintenance*, 18:37–58, 01 2006.

[8] Raimar Falke, Pierre Frenzel, and R. Koschke. Empirical evaluation of clone detection using syntax suffix trees. *Empirical Software Engineering*, 13:601–643, 2008.

[9] Farima Farmahinifarahani, Vaibhav Saini, Di Yang, Hitesh Sajnani, and Cristina Lopes. On precision of code clone detection tools. pages 84–94, 02 2019.

[10] Nicolas Juillerat and Béat Hirsbrunner. An algorithm for detecting and removing clones in java code. 01 2006.

[11] Toshihiro Kamiya, Shinji Kusumoto, and Katsuro Inoue. Ccfinder: A multilinguistic token-based code clone detection system for large scale source code. *Software Engineering, IEEE Transactions on*, 28:654– 670, 08 2002.

[12] Cory J. Kapser and Michael W. Godfrey. "cloning considered harmful" considered harmful: Patterns of cloning in software. *Empirical Softw. Engg.*, 13(6):645–692, December 2008.

[13] Zhenmin Li, Shan Lu, Suvda Myagmar, and Y. Zhou. Cp-miner: A tool for finding copy-paste and related bugs in operating system code. In *OSDI*, 2004.

[14] Golam Mostaeen, Banani Roy, Chanchal Roy, Kevin Schneider, and Jeffrey Svajlenko. A machine learning based framework for code clone validation. *Journal of Systems and Software*, 169:110686, 06 2020.

[15] Golam Mostaeen, Jeffrey Svajlenko, Banani Roy, Chanchal K. Roy, and Kevin A. Schneider. Clonecognition: Machine learning based code

clone validation tool. In *Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, ESEC/FSE 2019, page 1105–1109, New York, NY, USA, 2019. Association for Computing Machinery.

[16] Matthias Rieger. Effective clone detection without language barriers. 2005.

[17] C. Roy and J. Cordy. Benchmarks for software clone detection: A ten-year retrospective. *2018 IEEE 25th International Conference on Software Analysis, Evolution and Reengineering (SANER)*, pages 26–37, 2018.

[18] Chanchal Roy and James Cordy. A survey on software clone detection research. *School of Computing TR 2007-541*, 01 2007.

[19] Chanchal Roy and James Cordy. A mutation/injection-based automatic framework for evaluating code clone detection tools. *IEEE International Conference on Software Testing, Verification, and Validation Workshops, ICSTW 2009*, 01 2009.

[20] Chanchal Roy and J.R. Cordy. Nicad: Accurate detection of near-miss intentional clones using flexible pretty-printing and code normalization. pages 172–181, 07 2008.

[21] Chanchal K. Roy, James R. Cordy, and Rainer Koschke. Comparison and evaluation of code clone detection techniques and tools: A qualitative approach. *Science of Computer Programming*, 74(7):470 – 495, 2009.

[22] J. Svajlenko, J. F. Islam, I. Keivanloo, C. K. Roy, and M. M. Mia. Towards a big data curated benchmark of inter-project code clones. In *2014 IEEE International Conference on Software Maintenance and Evolution*, pages 476–480, 2014.

[23] J. Svajlenko and C. K. Roy. Evaluating clone detection tools with bigclonebench. In *2015 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, pages 131–140, 2015.

[24] Brent van Bladel and Serge Demeyer. A comparative study of test code clones and production code clones, Mar 2021.

[25] Danny van Bruggen, Federico Tomassetti, Roger Howell, Malte Langkabel, Nicholas Smith, Artur Bosch, Malte Skoruppa, Cruz Maximilien, ThLeu, Panayiotis, Sebastian Kirsch (@skirsch79), Simon, Johann Beleites, Wim Tibackx, jean pierre L, André Rouél, edefazio, Daan Schipper, Mathiponds, Why you want to know, Ryan Beckett, ptitjes, kotari4u, Marvin Wyrich, Ricardo Morais, Maarten Coene, bresai, Implex1v, and Bernhard Haumacher. javaparser/javaparser: Release javaparser- parent-3.16.1, May 2020.

[26] Wuu Yang. Identifying syntactic differences between two programs. *Softw. Pract. Exper.*, 21(7):739–755, June 1991.