Code Clone Benchmarks Overview

TIJANA VISLAVSKI and GORDANA RAKIĆ, University of Novi Sad

Traditionally, when a new code clone detection tool is developed, few well-known and popular benchmarks are being used to evaluate the results that are achieved. These benchmarks have typically been created by cross-running several state-of-the-art clone detection tools, in order to overcome the bias of using just one tool, and combining their result sets in some fashion. These candidate clones, or more specifically their subsets, have then been manually examined by clone experts or other participants, who would judge whether a candidate is a true clone or not. Many authors dealt with the problem of creating most objective benchmarks, how the candidate sets should be created, who should judge them, whether the judgment of these participants can be trusted or not. One of the main pitfalls, as with development of a clone detection tool, is the inherent lack of formal definitions and standards when it comes to clones and their classification. Recently, some new approaches were presented which do not depend on any clone tool, but utilize search heuristics in order to find specific functionalities, but these candidates are also manually examined by judges to classify them as true or false clones. This paper has a goal of examining state-of-the-art code clone benchmarks, as well as studies regarding clone judges reliability (and subsequently reliability of the benchmarks themselves) and their possible usage in a cross-language clone detection context.

1. INTRODUCTION

Evaluation of the results is one of the most important parts of any research, and code clone detection is no exception to that. Code clones are defined as similar source code fragments according to some definition of similarity. A code fragment is a tuple which contains a file name, start and end line. Code clones are usually categorized in four groups [Roy et al. 2009]:

(1) Type 1 clones: identical source code fragments except whitespaces, comments and layout,
(2) Type 2 clones: syntactically identical source code fragments except identifiers, types, literals, whitespaces, comments and layout,
(3) Type 3 clones: code fragments with modifications such as added, removed or changed lines of code, in addition to rules for Type 2,
(4) Type 4 clones: code fragments that perform same computation but are not syntactically similar.

Clone detection tools can use different techniques, such as text-based algorithms, token-based algorithms, comparing of source code metrics, comparing of abstract syntax trees, program dependency graphs etc. Overview of these techniques is outside of scope for this paper, but can be found in [Roy et al. 2009] among others.

Several benchmarks [Bellon et al. 2007] [Krutz and Le 2014] [Svajlenko et al. 2014] [Wagner et al. 2016] have been proposed with the goal of evaluating two crucial indicators of clone detection tools' quality: precision and recall. In the context of code clone detection, precision represents a ratio of real

Author's address: T. Vislavski, Department of Mathematics and Informatics, Faculty of Sciences, University of Novi Sad, 4 Dositej Obradović Square, Novi Sad 21000; email: tijana.vislavski@dmi.uns.ac.rs; Gordana Rakić, Department of Mathematics and Informatics, Faculty of Sciences, University of Novi Sad, 4 Dositej Obradović Square, Novi Sad 21000; email: goca@dmi.uns.ac.rs

In: Z. Budimac (ed.): Proceedings of the SQAMIA 2018: 7th Workshop of Software Quality, Analysis, Monitoring, Improvement, and Applications, Novi Sad, Serbia, 27–30.8.2018. Also published online by CEUR Workshop Proceedings (http://ceur-ws.org, ISSN 1613-0073)

clones among all the reported clone candidates:

$$precision = \frac{reported\ real\ clones}{all\ reported\ clone\ candidates},$$

while recall represents a ratio of reported real clones among all clones in the analyzed source code:

$$recall = \frac{reported\ real\ clones}{all\ clones}.$$

Clone benchmark creation usually has following workflow:

(1) Source code is analyzed by one or more code detection tools and candidate clones are detected,
(2) Candidate clones (or a fraction of them) are evaluated by one or more human judges who decide whether they are true clones or not.

Many methods have been applied to try to overcome the bias of subjectivity of judges and tools used for creation of a benchmark, but the objectivity of such benchmarks still remains an open topic. This paper aims to provide an overview of current state-of-the-art code clone benchmarks and papers that investigate validity and objectivity of these benchmarks and their described creation process, among them works presented in [Charpentier et al. 2017] [Kapser et al. 2007] [Walenstein et al. 2003]. Other processes of creating clone benchmarks have also been proposed such as artificial mutation of arbitrary code fragments [Svajlenko and Roy 2014], and these will also be described in subsequent chapters.

Final goal of this overview is to investigate to which extent, if any, could current benchmarks be used (with some adjustments) for evaluating results of clone detection in a cross-language context. LICCA [Vislavski et al. 2018] is a language independent tool for code clone detection with main focus on detecting clones in such context. Even though finding similar code fragments between languages is a challenging task, it is very important to have means to evaluate even a slightest progress.

Rest of the paper is organized as follows: In section 2 an industry standard benchmark is described, followed by a critique of this benchmark. Section 3 introduces other proposed benchmarks, including some new and different approaches. Section 4 gives an overview of works that deal with reliability of judges included in benchmark creation. Finally, section 5 concludes this paper with authors' findings.

2. BELLON'S CLONE BENCHMARK

Work presented in [Bellon et al. 2007] by Bellon et al. was first benchmark for code clones and has since become an industry standard for evaluating code clone detection tools [Charpentier et al. 2015]. In this paper, authors presented results of the experiment in which they evaluated six clone detection tools (Dup [Baker 2007], CloneDR [Baxter et al. 1998], CCFinder[Kamiya et al. 2002], Duplix [Krinke 2001], CLAN [Merlo 2007], Duploc [Ducasse et al. 1999]) using eight C and Java projects. These tools use a variety of approaches, including metric comparison [Merlo 2007], comparison of abstract syntax trees [Baxter et al. 1998], comparison of program dependency graphs [Krinke 2001] etc. Although tools support detection of clones in different languages, none of them supports detection *between* different languages.

2.1 Benchmark's creation process

During the experiment that produced Bellon's benchmark, authors of clone detection tools that participated in this experiment, applied their tools to followwing C and Java projects and reported clone candidates:

(1) weltab[1]

[1]No link for this project was found

(2) cook[2]
(3) snns[3]
(4) postgresql[4]
(5) netbeans-javadoc[5]
(6) eclipse-ant[6]
(7) eclipse-jdtcore[7]
(8) j2sdk1.4.0-javax-swing[8]

Candidate clones were then evaluated by first author of the paper, Stefan Bellon, as an independent third-party. LOC (Lines of code) for these projects varied from 11K to 235K. Each project was analyzed with tool's default settings (mandatory part), and with tuned settings (voluntary part). Only clones of six lines or longer were reported.

The reference corpus of 'real' clones was created manually by Stefan Bellon. He looked at 2% of all submitted candidates (approx. 6K) and chose real clones among them (sometimes he modified them slightly). He did not know which candidates were found by which tool and 2% of candidates was distributed equally among the six tools. Additionaly, some clones were injected back in each of the analyzed programs in order to get a better information about the recall.

When evaluating whether a candidate matches a reference, authors did not insist on complete overlapping of candidate and reference clone, but rather on sufficiently large overlapping. In order to precisely and consistently measure this overlap, authors introduced some definitions and metrics. For two code fragments CF_1 and CF_2 they define:

(1) $overlap(CF_1, CF_2) = \frac{|lines(CF_1) \cap lines(CF_2)|}{|lines(CF_1) \cup lines(CF_2)|}$,
(2) $contained(CF_1, CF_2) = \frac{|lines(CF_1) \cap lines(CF_2)|}{|lines(CF_1)|}$,

Clone pair CP is defined as a tuple $CP = (CF1, CF2, t)$, i.e. a tuple containing two code fragments and a clone type. For two clone pairs CP_1 and CP_2 authors define:

(1) good-value: $good(CP_1, CP_2) = min(overlap(CP_1.CF_1, CP_2.CF_1), overlap(CP_1.CF_2, CP_2.CF_2))$,
(2) ok-value: $ok(CP_1, CP_2) = min(max(contained(CP_1.CF_1, CP_2.CF_1), contained(CP_2.CF_1, CP_1.CF_1)),$
$max(contained(CP_1.CF_2, CP_2.CF_2), contained(CP_2.CF_2, CP_1.CF_2)))$

Obviously, ok-value is less strict than good-value (ok-value requires for one clone pair to be contained in other for at least $p \times 100\%$ which can lead to one clone pair being much longer than the other, while in good-value this cannot happen since it looks at the minimum of intersections in clone pair fragments). For both metrics, a threshold of 0.7 was used. Since the minimum number of lines in a clone was set to 6, this value allows for two six-line-long fragments to be shifted by one line. The same threshold value was chosen for both metrics for uniformity: both are measures of overlap, the difference is in the perspective (good-value from the perspective of both fragments, and ok-value from the perspective of the smaller fragment).

[2]http://miller.emu.id.au/pmiller/software/cook/ (Accessed: October, 2017)
[3]http://www.ra.cs.uni-tuebingen.de/SNNS/ (Accessed: October, 2017)
[4]https://www.postgresql.org/ (Accessed: October, 2017)
[5]https://netbeans.org/ (Accessed: October, 2017)
[6]http://www.eclipse.org/eclipse/ant/ (Accessed: October, 2017)
[7]https://www.eclipse.org/jdt/core/ (Accessed: October, 2017)
[8]http://www.oracle.com/technetwork/java/index.html (Accessed: October, 2017)

From approximately 6 thousand investigated clone candidates, Bellon accepted around 4 thousand (66%) as true clones. However, this is a percentage across all analyzed programs. For individual programs, percentage of accepted true clones varied from 35% to 90%. While analyzing specific tools, it has been found that tools which have a higher number of reported candidates, have a higher recall and also a higher number of rejected candidates. Additionally, number of CLAN's candidates that hit the reference corpus was the highest, although it reported the smallest number of candidates. Although authors report some values for recall and precision for all tools, they have to be taken with caution since only a portion of all candidates was examined. Nevertheless, since authors compared these numbers with results obtained after examination of first 1% of candidates, they concluded that numbers are pretty much stable.

Authors also found that different tools yield different sizes of clone candidates, which can be explained by the fact they use different techniques. Only 24%-46% of injected clones was found.

Overall conclusions of the experiment were:

(1) Text and token based tools have higher recall, while metric and tree based tools have higher precision.
(2) The PDG (Program Dependency Graph) tool doesn't perform well (except for type-3 clones).
(3) There were a large number of false positives. Type-1 and type-2 clones can be found reliably, while type-3 clones are more problematic.
(4) Although Stefan Bellon was an independent party, the judgment was still biased by his opinion and experience.

2.2 Critique of Bellon's benchmark

Several years later, authors of [Charpentier et al. 2015] empirically examined Bellon's benchmark validity. Their main argument was that the benchmark has the probability of being biased, since it was created by Bellon alone, who was not an expert for the systems being analyzed.

They seeked an opinion of 18 participants on a subset of reference clones determined by Bellon. Conclusion is that some clones are debatable, as well as that the fact whether a candidate is considered a clone or not is subjective: it largely depends on perspective and intent of a user, since the definition of the clones itself is open to interpretation (similar conclusion was come up with by Yang et al. in their paper [Yang et al. 2015]).

In more detail, experimental setup of this empirical assessment consisted of nine groups of two participants, where each group analyzed randomly chosen subset of 120 reference clones defined by Bellon (total of 1080 reference clones). All participants were students of undergraduate (last year) and graduate studies with IT background (both for Java and C). Participants judged whether the references were true clones or not in order to analyze whether there are some reference clones for which participants had different opinion than Bellon.

After the collections of participants' answers, there were in total 3 answers for each examined reference clone (one positive from Bellon and two given by the students). Each clone was assigned a trust level based on these answers:

(1) *good* for clones with 3 positive answers
(2) *fair* for clones with 2 positive and 1 negative answer,
(3) *small* for clones with 1 positive and 2 negative answers.

Using this data, authors calculated, with 95% confidence intervals, proportions of clones in each of the trust levels. According to this calculation, about half of clones have trust level less than *good*, and about 10-15% of clones have only trust level *small*, which are not negligible percentages.

Next, authors examined how do trust levels affect recall and precision values that are calculated by the benchmark. Two scenarios were considered: one that took into account only clones with trust level *good* as true clones, and other that took into account clones with both trust levels *good* and *fair*. Then precision and recall were calculated by the Bellon's formula with new reference set. Conclusion was that both recall and precision decrease significantly (up to 0.49 for the first scenario, and up to 0.12 for the second scenario).

In the end, authors investigated whether clones with trust level *good* have some common characteristics that separate them from other clones. 3 features were taken into account: type, size and language. They statistically tested hypotheses about correlation between these features and clone‘s trust level. For size, only a loose correlation has been calculated in favor of bigger clones. For clone type, a moderate negative correlation has been calculated, which means that only type 1 clones can be considered as having *good* trust level after one opinion, while other types need more opinions. This makes sense when we consider the definition of each type, where type 1 clones are identical code fragments, while higher types are more loosely and less strictly defined. Regarding programming languages' impact on trust level, negligible correlation has been calculated.

The main threats to validity that authors report are mainly the fact that students were used for the assessment (although they all had an IT background, none of them was an expert for the analyzed systems, which is a thing that authors themselves reported as a Bellon's disadvantage, and only half of them knew about the concept of a code clone before the experiment), as well as the fact that only a small portion of the Bellon's benchmark was assessed. Furthermore, authors only considered clones that were reported as true clones by Bellon. A large portion of clone candidates investigated by Bellon were graded as false clones. In order to better investigate the effect of different opinions on calculated precision and recall, these false clones should also be graded by multiple people.

3. OTHER BENCHMARKS

After Bellon et al., in the following years there were more efforts to create an unbiased and objective code clone benchmark.

3.1 Intra-project, method-level clones

Krutz et al. [Krutz and Le 2014] constructed a set of intra-project, method-level clones from 3 open-source projects: PostgreSQL, Apache[9] and Python[10], which are all C-based projects. Authors randomly sampled 3-6 classes from each project, created all possible function pairs from these classes, evaluated them by several clone detection tools and then given them to 7 raters for manual examination (4 clone experts and 3 students). Experts examined clone pairs in collaboration and clones were confirmed only after consensus was achieved, while students examined clone pairs independently. Idea behind this setup was to improve the confidence of data, and also to check the difference from students' and experts' answers. In total, 66 clone pairs were found by the expert group, none of which were type-1 clones, 43 were type-2, 14 were type-3 and 9 were type-4 clones. Authors reported only clones found by the expert group, since they concluded that student answers differ largely with one another, and while there is a certain number of clones that they agree on, overlap with clones reported by experts is not a significant one.

[9]https://httpd.apache.org/ (Accessed: October, 2017)
[10]https://www.python.org/ (Accessed: October, 2017)

3.2 Mutation and Injection Framework

In 2014 Svajlenko and Roy [Svajlenko and Roy 2014] evaluated the recall of 11 clone detection tools using the Bellon's [Bellon et al. 2007] benchmark and their own Mutation and Injection Framework. Their main goal was to evaluate the recall, since it was an inherently more difficult metric for calculation than precision. Their proposed framework uses a corpus of artificially created clones based on a taxonomy suggested in the previous study [Roy et al. 2009]. The framework extracts random code fragments from the system and makes several copies which are then changed by mutation operators following the taxonomy. There were 15 operators in total, which create first 3 clone types. Original and mutated fragments are inserted back in the system and the process is repeated thousands of times. This creates a large corpus of mutant systems. When a clone detection tool is tested with the framework, its recall is measured specifically for the injected clones.

Authors compared their calculated recall values for several tools with the ones reported by Bellon and saw that values rarely agree (with threshold for agreement of 15% difference), with their framework generally giving higher recall values. The validity of their measurement over Bellon's was backed up by previous expectations. Before running the experiment, authors made well thought-out assessments of the recall for the analyzed tools based on documentation, publications and literature discussions. The Mutation Framework agreed with these expectations in 90% cases for Java and 74% cases for C, while Bellon's benchmark agreed with expectations in only around 30% of cases. Authors' conclusion was therefore that Bellon's benchmark, while still a very popular benchmark for evaluating clone detection tools, may not be accurate for modern tools, and that there is a need for an updated corpus.

3.3 BigCloneBench

In [Svajlenko et al. 2014] authors state that the common approach for creating a benchmark, which is by using clone detection tools to find clone candidates and manually evaluating them, gives an unfair advantage to the tools that participated in creating the benchmark. As they have proven in the paper described earlier, The Mutation Framework, which is a synthetic way to create clones and inject them back in the system, is a better approach for modern clone detection tools. However, there is still a need for a benchmark of real clones. The contribution of the paper is one such benchmark, one that authors call BigCloneBench and which was built by mining the IJaDataset 2.0 [11].

IJaDataset 2.0 is a big inter-project repository consisting of 25,000 project and more than 365MLOC. Construction of the benchmark didn't include any clone detection tools, which makes possible for the benchmark not to be biased, but rather used a search heuristics to identify code snippets that might implement a target functionality (such as Bubble Sort, Copy a File, SQL Update and Rollback etc). These candidate snippets were than manually judged as true or false clones, and (if true) labeled by their clone type. The benchmark includes 10 functionalities with 6 million true clone pairs and 260 thousand false clone pairs. Since the clones were mined by functionality, this benchmark is also appropriate for detection of semantic clones, which was the first benchmark at that time for semantic clones. One more contribution of this work was the scale of the benchmark. Previous benchmarks were all significantly smaller in size, while the BigCloneBench is appropriate for big data clone detection and scalabillity evaluation.

3.4 Benchmark from Google Code Jam submissions

Another benchmark for *functionally similar clones*, as opposed to copy&paste ones, was recently suggested in [Wagner et al. 2016]. Authors intentionally use this term, instead of well-known Type 4

[11] https://sites.google.com/site/asegsecold/projects/seclone (Accessed: October, 2017)

clones, because they are interested in similar functionalities, not only identical ones. Since the traditional clone detection tools rely on syntactical similarity, they are hardly able to detect these clones. This was evaluated and confirmed as a part of the study and a proposed benchmark is a useful help for further research of this type of clones. The benchmark was constructed from a set of coding contest submissions (Google Code Jam[12]) in two languages - Java and C. Since the submissions are aiming for the solution of same problems, it is expected that they would contain a large number of functionally similar code snippets. The benchmark consists of 58 functionally similar clone pairs. Once again, only clones between same language (either Java or C) were reported, without consideration of inter-language clones.

4. RATERS RELIABILITY

When constructing a clone benchmark, authors traditionally involve *clone oracles*, people (code clone experts or others) who judge whether clone candidates are true clones or not [Bellon et al. 2007] [Charpentier et al. 2015] [Yang et al. 2015] [Krutz and Le 2014] [Svajlenko et al. 2014]. However, as already mentioned in previous sections, code clone definitions lack precision, they are open to interpretation, it is highly subjective whether a clone candidate is a clone to an individual or not, it depends on a person's intent, on their expertise about the system under examination etc. Many authors investigated this problem and tried to gain some insight about raters' agreement in classifying clones, as well as reliability of these judges. [Charpentier et al. 2017] [Kapser et al. 2007] [Walenstein et al. 2003]

One of the first of these works was done in [Walenstein et al. 2003]. Authors themselves acted as judges and classified a selection of function clone candidates (clone candidates that span across whole function bodies) as true or false clones. Their main hypotheses was that, as an educated and informed researchers in the domain of code clone detection, their decisions would be much or less the same. However, they found out that their answers highly differ. As an example, authors report a case where for a set of 317 clone candidates, 3 judges had the same opinion only in 5 cases. Obviously, consensus what is or is not a clone is not automatic, even among researchers that are clone experts. This is, among others, a consequence of vagueness of the definitions for code clones. After this initial results, authors defined classification criteria more precisely and agreed upon them. Then they repeated the process. Although level of agreement increased, it was still considerably low for some systems (authors report level of agreement of .489 for one system). After some in-depth analysis, they realized that their perspective on some design criteria differs (for example, whether constructors should be refactored or not). In the next iteration they introduced one more judge, unfamiliar with all consensuses and criteria agreed upon until then (hours of debating and discussion), to test his responses. Surprisingly, level of agreement improved, which was an indication that maybe choice of individual judges is more important than the group experience. In the final run, authors decided to classify clones specifically based on information from the source code itself, and disregard all context-based knowledge. This, as expected, increased the level of agreement. In the end, authors conclude that inter-rater reliability is potentially a serious issue, and one judge is definitely not enough. Furthermore, their experiment does not support the idea that practice is any more important than set of well-defined guidelines. Finally, specifics of each system can have a great impact on judge reliability.

Another similar experiment was performed at an international workshop (DRASIS [Koschke et al. 2007]) and the results are presented in [Kapser et al. 2007]. After the thorough discussion about fundamental theory behind code clones and lack of formal definitions, a question has arised: will researchers in this area ever agree between themselves what is a code clone and what is not? Such a question has serious consequences to the validity and reproducibility of previous results across literature. During

[12]https://code.google.com/codejam/ (Accessed: October, 2017)

a working session of the mentioned workshop, a 20 clone candidates reported by clone detection tool CCFinder [Kamiya et al. 2002] for PostgreSQL system were chosen by session leader and 8 clone experts privately judged them. Only 50% of candidates were classified with the agreement level of at least 80% (7 or more persons). This seriously puts in question reproducibility of previous studies. After the judgment, results were discussed. It was concluded that attendants interpreted differently same attributes of given candidates, their significance and meaning. For example: size of candidate, level of change in types, possibility to refactor the code etc. This list lead the attendants to the following questions: Was the formulation of the question a problem? Should the question be changed to something more appropriate, using terms such as duplicated code, redundant code, similar code? Would it affect the results of evaluation? One additional point was that maybe forcing the binary answer, a simple yes or no, is increasing the disagreement. Maybe if a multiple-point scale was used, answers would be more agreeable? The session brought many still unanswered questions. But one of the most important conclusions was made: there is little chance that a reviewer can be sure of the true meaning of the results without clear documentation about criteria that were used.

One quite recent work [Charpentier et al. 2017] dealt with the question of non-expert reliability of judgment whether a clone candidate is a true or false clone. The hypothesis is that one clone candidate could be differently rated in different projects and contexts. The experiment was performed on a set of 600 clones from 2 Java projects, with both experts and non-experts used for judgment (authors define term expert as a person with expertise in a particular project, all of the raters were familiar with the notion of clones). Formally speaking, 3 research questions were asked: are the answers consistent over time, do external raters agree with each other and with experts of a project, what are the characteristics that influence the agreement between experts and external raters. Each randomly selected set of clones was presented to 4 raters, including one expert for the underlying system. They were asked to rate the clone pairs with yes/no/unknown answers and used statistical analysis to answer the research questions. Based on the collected data, authors derived following conclusions based on the presented questions: regarding first question (consistency of raters answers), authors noticed that experts' answers are very consistent, while inconsistencies in non-expert judgments vary from 5% to 20% of the rated duplicates. For the second question (inter-rater reliability), authors conclude that for both investigated projects, there is no agreement between external raters, while the strongest agreement is achieved when comparing majority of non-expert answers with the expert ones, and not when comparing any particular non-expert with the expert. Final question (factors influencing agreement between external raters and experts) lead the authors to the conclusion that some characteristics have an impact on the agreement between the majority and the expert, such as size or project (latter one suggesting that some projects are easier for non-experts to judge).

5. CONCLUSION

Although code clone detection is a well-researched area with years of work dedicated to it and few extraordinary, production-quality tools, it is still not mature enough in terms of formal definitions and standards. One of the main consequences of this is the lack of universal and reliable benchmarks. Many attempts exist, and are still being worked on in this area, but the lack of definitions is causing much confusion and disagreement even between code clone experts.

Benchmarks are partially created manually, involving human judges which have the final word whether a clone candidate is a true or a false clone. However, as several experiments showed, some of which were presented in this paper, there is a high level of disagreement between judges, non-experts and experts alike. This seriously questions validity of current benchmarks. Furthermore, it has been shown that different clones are considered interesting in different contexts and with different objectives. This is also a problem in achieving some universal benchmark.

It is clear that code clone benchmarking still requires a lot of effort, primarily overcoming the problematics of manual judgment and bias. However, current benchmarks can provide some valuable information when comparing an arbitrary tool's results with some state-of-the-art tools, which may lack a formal assessment but have proven themselves in practise.

Regarding a possibility of using any of these benchmarks in cross-language context, we can conclude that only those which contain semantic, or functionally similar clones can be considered. Other benchmarks concentrate on copy&paste clones which are not to be found in fragments written in different languages. Since BigCloneBench contains Java projects exclusively, the benchmark which could provide a starting point for evaluating cross-language clones is the one presented in [Wagner et al. 2016]. Although it contains examples in two syntactically similar languages (Java and C), differences in their respective paradigms mean significant differences in code organization and structure.

REFERENCES

Brenda S Baker. 2007. Finding clones with dup: Analysis of an experiment. *IEEE Transactions on Software Engineering* 33, 9 (2007).

Ira D Baxter, Andrew Yahin, Leonardo Moura, Marcelo Sant'Anna, and Lorraine Bier. 1998. Clone detection using abstract syntax trees. In *Software Maintenance, 1998. Proceedings., International Conference on*. IEEE, 368–377.

Stefan Bellon, Rainer Koschke, Giulio Antoniol, Jens Krinke, and Ettore Merlo. 2007. Comparison and evaluation of clone detection tools. *IEEE Transactions on software engineering* 33, 9 (2007).

Alan Charpentier, Jean-Rémy Falleri, David Lo, and Laurent Réveillère. 2015. An empirical assessment of Bellon's clone benchmark. In *Proceedings of the 19th International Conference on Evaluation and Assessment in Software Engineering*. ACM, 20.

Alan Charpentier, Jean-Rémy Falleri, Floréal Morandat, Elyas Ben Hadj Yahia, and Laurent Réveillère. 2017. Raters reliability in clone benchmarks construction. *Empirical Software Engineering* 22, 1 (2017), 235–258.

Stéphane Ducasse, Matthias Rieger, and Serge Demeyer. 1999. A language independent approach for detecting duplicated code. In *Software Maintenance, 1999.(ICSM'99) Proceedings. IEEE International Conference on*. IEEE, 109–118.

Toshihiro Kamiya, Shinji Kusumoto, and Katsuro Inoue. 2002. CCFinder: a multilinguistic token-based code clone detection system for large scale source code. *IEEE Transactions on Software Engineering* 28, 7 (2002), 654–670.

Cory Kapser, Paul Anderson, Michael Godfrey, Rainer Koschke, Matthias Rieger, Filip Van Rysselberghe, and Peter Weißgerber. 2007. Subjectivity in clone judgment: Can we ever agree?. In *Dagstuhl Seminar Proceedings*. Schloss Dagstuhl-Leibniz-Zentrum für Informatik.

Rainer Koschke, Andrew Walenstein, and Ettore Merlo. 2007. 06301 Abstracts Collection–Duplication, Redundancy, and Similarity in Software. In *Dagstuhl Seminar Proceedings*. Schloss Dagstuhl-Leibniz-Zentrum für Informatik.

Jens Krinke. 2001. Identifying similar code with program dependence graphs. In *Reverse Engineering, 2001. Proceedings. Eighth Working Conference on*. IEEE, 301–309.

Daniel E Krutz and Wei Le. 2014. A code clone oracle. In *Proceedings of the 11th Working Conference on Mining Software Repositories*. ACM, 388–391.

Ettore Merlo. 2007. Detection of plagiarism in university projects using metrics-based spectral similarity. In *Dagstuhl Seminar Proceedings*. Schloss Dagstuhl-Leibniz-Zentrum für Informatik.

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

Jeffrey Svajlenko, Judith F Islam, Iman Keivanloo, Chanchal K Roy, and Mohammad Mamun Mia. 2014. Towards a big data curated benchmark of inter-project code clones. In *Software Maintenance and Evolution (ICSME), 2014 IEEE International Conference on*. IEEE, 476–480.

Jeffrey Svajlenko and Chanchal K Roy. 2014. Evaluating modern clone detection tools. In *Software Maintenance and Evolution (ICSME), 2014 IEEE International Conference on*. IEEE, 321–330.

Tijana Vislavski, Gordana Rakic, Nicolás Cardozo, and Zoran Budimac. 2018. LICCA: A tool for cross-language clone detection. In *2018 IEEE 25th International Conference on Software Analysis, Evolution and Reengineering (SANER)*. IEEE, 512–516.

Stefan Wagner, Asim Abdulkhaleq, Ivan Bogicevic, Jan-Peter Ostberg, and Jasmin Ramadani. 2016. How are functionally similar code clones syntactically different? An empirical study and a benchmark. *PeerJ Computer Science* 2 (2016), e49.

Andrew Walenstein, Nitin Jyoti, Junwei Li, Yun Yang, and Arun Lakhotia. 2003. Problems Creating Task-relevant Clone Detection Reference Data.. In *WCRE*, Vol. 3. 285.

Jiachen Yang, Keisuke Hotta, Yoshiki Higo, Hiroshi Igaki, and Shinji Kusumoto. 2015. Classification model for code clones based on machine learning. *Empirical Software Engineering* 20, 4 (2015), 1095–1125.