BigCloneEval: A Clone Detection Tool Evaluation Framework with BigCloneBench

Jeffrey Svajlenko
Department of Computer Science
University of Saskatchewan
Saskatoon, Canada
jeff.svajlenko@usask.ca

Chanchal K. Roy
Department of Computer Science
University of Saskatchewan
Saskatoon, Canada
chanchal.roy@usask.ca

***Abstract*—Many clone detection tools have been proposed in the literature. However, our knowledge of their performance in real software systems is limited, particularly their recall. We previously introduced our BigCloneBench, a big clone benchmark of over 8 million clones within a large inter-project Java repository containing 25,000 open-source Java systems. In this paper we present BigCloneEval, a framework for evaluating clone detection tools with BigCloneBench. BigCloneEval makes it very easy for clone detection researchers to evaluate and compare clone detection tools. It automates the execution and evaluation of clone detection tools against the reference clones of BigCloneBench, and summarizes recall performance from a variety of perspectives, including per clone type, and per syntactical similarity regions.**

I. Introduction

Clone detection tools locate similar source code within or between software systems. Instances of similar code fragments are called clones. Developers create clones when they reuse code using copy, paste and modify, either within a software system or between software projects, although clones may arise for a variety of other reasons [1]. By managing or refactoring their clones, developers can maintain and improve software quality, reduce development costs and risks, prevent and detect bugs, and more [1]. Clone management and research studies depend on clone detection tools. In 2013, Rattan et al. [2] found at least 70 tools in the literature. Despite this, there is a lack of frameworks for evaluating and comparing the performance of clone detection tools.

Clone detection tools are evaluated by their recall and precision. Recall is the ratio of the clones within a software system or repository that a tool is able to detect, and precision is the ratio of the clones reported by a tool that are true clones, not false positives. Precision can be measured by manually validating a sample of the tool's output. On the other hand, recall has been challenging to measure as it requires a benchmark of known reference clones [1].

We previously introduced BigCloneBench [3], a real-world benchmark of manually validated clones in the inter-project Java repository IJaDataset-2.0 [4] (25,000 Java systems). It was built by mining IJaDataset for clones of common functionalities. BigCloneBench was built independently of the clone detection tools themselves, thereby avoiding the biases in previous benchmarks [5], [6]. The current version of the benchmark contains over 8 million clone pairs across 43 functionalities. Each clone pair is semantically similar by its functionality. It contains both intra and inter-project clones spanning the four primary clone types, including the entire range of syntactical similarity. We have shown that BigCloneBench is effective in measuring and comparing the recall performance of modern clone detection tools [7], [8]. While BigCloneBench's primary use-case is measuring the recall of clone detection tools, it can also be used as a basis for other clone and software studies.

In order to make BigCloneBench more accessible, we introduce BigCloneEval, a framework for evaluating clone detection tools using BigCloneBench. It is based on the tool evaluation procedure we have used in our previous tool comparison studies [7], [8]. BigCloneEval makes it very easy for users to evaluate and compare the recall of clone detection tools with BigCloneBench. The user does not have to write any evaluation code beyond configuring their candidate tools for execution and converting clone detection reports to a standard format. BigCloneEval handles the execution of the candidate clone detector for IJaDataset, including managing possible scalability constraints of the tool using deterministic input partitioning. BigCloneEval tracks the detected clones, and efficiently determines which of the reference clones in BigCloneBench the tool was able to detect. The evaluation experiment is highly configurable. The user can specify constraints on the reference clones considered when measuring recall, can customize the clone matching algorithm, or can provide their own clone matching algorithm by a plug-in architecture. BigCloneEval produces a tool evaluation report which summarizes recall per clone type, for both intra-project and inter-project clones, for different syntactical clone similarity regions, and for clones implementing different functionalities.

II. Definitions

Code Fragment: A continuous segment of source code. Specified by the triple (l, s, e), including the source file l, the start line, s and the end line, e.
Clone: A pair of code fragments that are similar: (f_1, f_2).
Type-1 (T1): Syntactically identical code fragments, except for differences in white space, layout and comments [5].
Type-2 (T2): Syntactically identical code fragments, except for differences in identifier names, literal values, white space,

TABLE I
BIGCLONEBENCH CLONE SUMMARY

Clone Type	T1	T2	VST3	ST3	MT3	WT3/T4
Number of Clone Pairs	47146	4609	4191	9516	38894	5818503
			8375313			

layout and comments [5].

Type-3 (T3): Syntactically similar code fragments that differ at the statement level. Fragments have statements added, modified and/or removed with respect to each other [5].

Type-4 (T4): Syntactically dissimilar code fragments that implement the same functionality [1].

III. BIGCLONEBENCH

BigCloneBench [3] is a clone detection benchmark consisting of manually validated clones in IJaDataset 2.0 [4], a large inter-project Java repository consisting of 2.3 million Java source files (365MLOC) from 25,000 open-source projects. The current version of BigCloneBench contains over 8 million validated clone pairs, including both intra-project and inter-project clones, spanning the four primary clone types, and the full range of clone syntactical similarity.

BigCloneBench was created, without the use of clone detection tools, by mining for functions in IJaDataset implementing specific functionalities. Functions that might implement a target functionality were identified using keyword and source-code pattern heuristics. The identified functions were manually tagged as true or false positives of the target functionality by judges. All true positive functions of a functionality form a large clone class of semantically similar functions. A clone class of size n contains $\frac{n(n-1)}{2}$ clone pairs. Post-processing identified the clone types and syntactical similarity of these clone pairs. Further details can be found in our publication [3].

The current version of BigCloneBench contains clones mined for 43 distinct functionalities. We summarize its contents per clone type in Table I. As there is no consensus on the minimum syntactical similarity of a Type-3 clone, it is difficult to separate the Type-3 and Type-4 clone pairs that implement the same functionality. Instead we separate the clones into four categories based on their syntactical similarity.

We define **Very-Strongly Type-3 (VST3)** clones as those with a similarity in range 90% (inclusive) to 100%, **Strongly Type-3 (ST3)**: 70-90%, **Moderately Type-3 (MT3)**: 50-70%, and **Weakly Type-3 or Type-4 (WT3/4)**: 0-50%. Syntactical similarity is measured for each reference clone as the ratio of the lines or tokens a code fragment shares with another after Type-1 and Type-2 normalizations. Shared lines or tokens are identified by unix-diff [9]. We classify the clones into these categories using the smaller of their line and token-based clone similarity measures. Further details on BigCloneBench are found in its publication [3].

IV. FRAMEWORK

BigCloneEval makes it easy to measure the recall of clone detection tools using our clone benchmark, BigCloneBench. It implements an experimental procedure similar to the one we have used in our previous clone detection tool evaluation experiments [7], [8]. BigCloneEval automates the major steps of the experiment, and allows the recall evaluation to be customized. It produces an extensive recall evaluation report that fully highlights the capabilities of a candidate clone detection tool.

TABLE II
BIGCLONEEVAL COMMANDS

Command	Description
registerTool	Registers a tool with the framework.
listTools	Lists the tool(s) registered with the framework.
deleteTool	Removes a tool, and its detected clones, from the framework.
partitionInput	Partitions a clone detection input given a maximum input size.
detectClones	Automates the execution of a tool for IJaDataset.
importClones	Imports a tool's detected clones into the framework.
clearClones	Removes the imported clones of a tool from the framework.
evaluateTool	Measures the recall of a tool and produces the tool evaluation report.

BigCloneEval has four primary components. (1) The BigCloneBench database, which documents the reference clones of BigCloneBench. (2) IJaDataset, the inter-project Java repository containing the reference clones. (3) A tools database, which tracks the clone detection tools being evaluated by the framework, and their detected clones. (4) A set of command-line tools for interacting with the framework, including the registering of clone detection tools, performing clone detection for IJaDataset, importing the detected clones, and performing the recall evaluation experiments. Table II lists the commands, which we describe further in the following sections.

BigCloneEval is distributed as a git repository, so that users can easily pull updates. BigCloneBench and IJaDataset are downloaded separately, and added to the distribution. BigCloneEval uses fast and efficient embedded databases so that the user does not have to install and setup a database server. The BigCloneBench database [10] and IJaDataset [4] repository are very large, so BigCloneEval uses special versions of these that contain only the data and source files needed to perform the recall measurement, reducing their storage requirements.

V. EVALUATION PROCEDURE

The tool evaluation procedure is shown in Figure 1. First the clone detection tool is registered with the framework, which assigns it a unique tool ID. Next, the tool is executed for IJaDataset, and its detected clones are collected. As a speedup, the tool only needs to be executed for the files in IJaDataset that contain clones in BigCloneBench. Clone detection can be executed manually by the user, or the framework can automate this process, including overcoming possible scalability limits of the clone detection tool using deterministic input partitioning. Then, the detected clones are imported into the tools database for the given tool. Lastly, the tool is evaluated against the clones in BigCloneBench. The evaluation is highly configurable, and the output tool evaluation report summarizes the tool's recall per clone type, per syntactical similarity region and per functionality in BigCloneBench. These individual steps, the output, and the framework commands are detailed in the remaining sections.

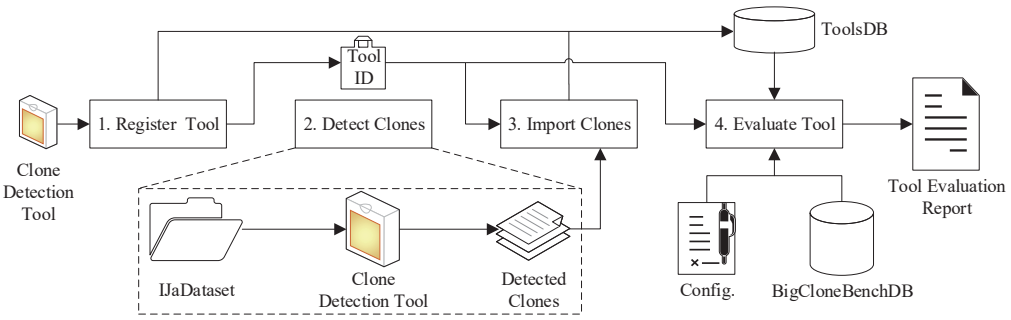

Fig. 1. BigCloneEval Evaluation Procedure

VI. Register Tool

The candidate clone detection tool is registered with the framework using the ***registerTool*** command, which requires the name of the tool and a description of its configuration for the experiment. These are stored in the database for reference, and a unique identifier is provided to the user for specifying this tool with the commands of the proceeding steps. The registered tools, their IDs, names and descriptions, can be listed using the ***listTools*** command. Tools can be removed from the framework using the ***deleteTool*** command.

VII. Detect Clones

Next the user must execute their candidate tool for IJaDataset and collect the detected clones. IJaDataset is very large, and outside the scalability limits of most clone detection tools. However, the clone detection tools do not need to be executed for the entire IJaDataset, only for the files containing reference clones in BigCloneBench. We provide a reduced version of IJaDataset which contains only the relevant source files and is split into a number of smaller subsets for clone detection. There is one subset per functionality in BigCloneBench. Each functionality's subset includes all the files which contain a function tagged as a true or false positive of that functionality in the creation of BigCloneBench. Therefore each subset is a realistic subject system, containing both true and false positive clones. The tool must be executed for each subset of IJaDataset, and the clones collected. This is equivalent to executing the tool for the entire IJaDataset, in terms of measuring recall for the reference clones.

A couple of these subsets may still be too large for some clone detection tools, specifically those that do not scale well in memory. This can be overcome using a deterministic input partitioning approach [11]. This involves partitioning the input and executing the tool for each unique pair of partitions. Partition size is chosen such that a pair of partitions does not exceed the scalability limits of the tool and available hardware. To perform deterministic input partitioning we provide a ***partitionInput*** command. This takes a directory of source files, a maximum input size in source files, and an output directory. Within the output directory it creates a subdirectory of source files for each unique pair of partitions given the maximum input size. Executing the tool for each subdirectory is equivalent to executing it for the original input.

While the user can perform the above manually with their clone detection tool, we also provide the ***detectClones*** command which automates the detection procedure. The user provides a script that configures and runs their tool, and the maximum input size considering their tool and available hardware, if required. The framework will automatically execute the tool for each subset of IJaDataset, using partitioning when needed, and collect the detected clones into a single output.

VIII. Import Clones

Now the user imports the clones detected by their clone detection tool into the tools database. This is done using the ***importClones*** command, which takes the ID of the registered tool and a file containing the clones to import. The clone file must list the clone pairs detected by the clone detection tool in a simple CSV format.

IX. Evaluate Tool

The ***evaluateTool*** command is used to measure the recall of the clone detection tool, and produce its tool evaluation report. This command requires the ID of the registered tool to evaluate, whose detected clones have already been imported, and a file to output the recall measurements to. It iterates through each reference clone in BigCloneBench and uses a clone matching algorithm to determine if the candidate tool was able to detect them. Recall is summarized for strategic subsets of the benchmark (e.g., per clone type) in the tool evaluation report (discussed further in Section X). The user can configure the evaluation procedure with a number of constraints on the clones considered when measuring recall. They can also customize or provide their own clone matching algorithm. We describe these further in the following subsections. By default, a configuration matching our previous clone benchmark experiments is used [7], [8].

A. Reference Clone Selection

The user can specify a number of constraints on the reference clones considered when measuring recall. Users can select clones for consideration by minimum and maximum clone size as measured by language-tokens, pretty-printed source lines, and/or original source lines. These options can be used to measure recall for clones within particular clone size ranges. They are also useful for reducing bias when measuring and comparing the recall of multiple tools. Clone detection tools typically require at least a minimum clone size

configuration, and most tools measure clone size by token or by source line (original or pretty-printed). By selecting a strict minimum and maximum clone size by each measure, the tools can be appropriately configured for BigCloneBench, and their recall results can be compared without bias due to clone size configuration. Users can also select reference clones by the total number of judges that have examined the code fragments of a reference clone, and their collective confidence in their judgment of those code fragments (the difference of true and false positive votes).

B. Clone Matching Algorithm

Recall is measured using a clone matching algorithm, which judges whether a reference clone in BigCloneBench is successfully detected by a candidate tool. BigCloneEval includes our coverage-based clone matcher, which we have used successfully in our previous work [7], [8], and is based on our *covers* metric. A code fragment f_1 covers code fragment f_2 if it intersects a ratio t of the source lines of f_2, as shown in Eq. 1, given that the code fragments are in the same source file. A reference clone R in BigCloneBench is considered detected by the candidate clone detector if there exists a candidate clone C reported by the candidate tool that satisfies the clone matcher. The coverage matcher is shown in Eq. 2, and requires the code fragments of C to cover the code fragments of R given a minimum coverage threshold t. Both orderings of the candidate clone's code fragments are tested. The coverage clone matcher is implemented as a database query over the tool's imported clones. Database indexes are used to make this query efficient, as the number of reference clones in BigCloneBench is very large.

$$covers(f_1, f_2, t) = \frac{min(f_1.e, f_2.e) - max(f_1.s, f_2.s) + 1}{f_2.e - f_2.s + 1} \geq t \quad (1)$$

$$c\text{–}match(C, R, t) = covers(C.f_1, R.f_1, t) \wedge covers(C.f_2, R.f_2, t) \quad (2)$$

The user can choose the coverage threshold of the coverage matcher (the default is 70%), as well as set a number of advanced configurations. The user can also provide their own custom clone matcher by a plug-in architecture. The user specifies the the name of the clone matcher and a configuration string. The clone matcher is discovered and configured at runtime. The existing coverage clone matcher can be used as a template by the user when implementing their own algorithm.

X. Tool Evaluation Report

The tool evaluation report summarizes the tool's recall performance for BigCloneBench given the configuration of the ***evaluateTool*** experiment. Recall is summarized per clone type, including the Type-3/Type-4 categories discussed in Section III. Recall is also measured for different minimum syntactical similarity thresholds, as well as for different regions of syntactical similarity. Recall is summarized for all clones, for just the intra-project clones, and for just the inter-project clones. It is also summarized for all clones, and for each of the individual functionalities in BigCloneBench. The report also summarizes the reference clones of BigCloneBench considered given the configuration of the experiment (e.g., clone size). The report names the versions of BigCloneBench and BigCloneEval used to measure recall, as well as the configurations of the experiment, including the clone matcher, for future reference.

XI. Limitations

BigCloneEval performs our clone detection tool recall evaluation procedure [7], [8]. While it has a number of customization options, including allowing custom clone matching algorithms, it does not extend beyond this procedure. The framework is open-source, so users can adapt the procedure if needed. As well, the full BigCloneBench database is available for users who are developing novel research studies and evaluation procedures [10]. BigCloneEval does not measure clone detection precision. There is no existing methodology for measuring precision automatically, and is typically done by manual clone validation. BigCloneEval measures recall in terms of clone pairs, while some tools also report clones as clone classes. There is not a standard for measuring recall considering clone class reporting. It is an open topic we would like to explore in future work, and integrate into BigCloneEval.

XII. Related Work

Bellon et al. [5] provide a benchmark of four thousand clones and a framework for evaluating clone detectors against this benchmark. Bellon's benchmark was built by manually validating a small fraction of the clones detected by participating tools in their benchmarking experiment [5]. As such, it is limited by the clone detection capabilities of its participating tools, which also introduces some biases [6]. Murakami et al. [12] extended Bellon's benchmarking by identifying the gap lines in Bellon's benchmark. Charpentier et al. [13] re-examined some of the clone validation efforts in Bellon's Benchmark and found disagreement in the results when multiple judges are used. We previously found that Bellon's Benchmark may not be appropriate for evaluating modern clone detection tools [14]. In contrast, BigCloneBench is a much larger benchmark, and was built independently of the clone detection tools in order to avoid bias. We introduced the Mutation and Injection Framework, which automatically measures the recall of clone detection tools in a mutation-analysis procedure. Its synthetic benchmarking compliments the real-world benchmarking strategy used by BigCloneEval.

XIII. Conclusion

In this paper, we introduced BigCloneEval, a framework for measuring the recall of clone detection tools using our BigCloneBench. BigCloneEval makes it very easy to perform clone detection tool benchmarking experiments with the reference clones in BigCloneBench. It gives the user flexibility over the configuration of the evaluation experiment, including the clone matcher used. Recall can be measured for both inter-project and intra-project clones, with recall summarized per clone type, per syntactical similarity range, and per functionality in the benchmark. BigCloneEval, and a demonstration video, is available at http://jeff.svajlenko.com/bigcloneeval.

REFERENCES

[1] C. K. Roy and J. R. Cordy, "A survey on software clone detection research," Queen's University, Tech. Rep. 2007-541, 2007, 115 pp.

[2] D. Rattan, R. Bhatia, and M. Singh, "Software clone detection: A systematic review," *Information and Software Technology*, vol. 55, no. 7, pp. 1165 – 1199, 2013.

[3] 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 *ICSME*, 2014, pp. 476–480.

[4] Ambient Software Evoluton Group, "IJaDataset 2.0," http://secold.org/projects/seclone, January 2013.

[5] S. Bellon, R. Koschke, G. Antoniol, J. Krinke, and E. Merlo, "Comparison and evaluation of clone detection tools," *Softw. Eng., IEEE Trans. on*, vol. 33, no. 9, pp. 577–591, 2007.

[6] B. Baker, "Finding clones with dup: Analysis of an experiment," *Softw. Eng., IEEE Trans. on*, vol. 33, no. 9, pp. 608–621, 2007.

[7] J. Svajlenko and C. Roy, "Evaluating clone detection tools with bigclonebench," in *ICSME*, 2015, pp. 131–140.

[8] H. Sajnani, V. Saini, J. Svajlenko, C. K. Roy, and C. V. Lopes, "Sourcerercc: Scaling code clone detection to big code," in *ICSE*, 2016, pp. 1157–1168.

[9] P. Eggert, M. Haertel, D. Hayes, R. Stallman, and L. Tower, "Diffutils - gnu project - free software foundation," http://www.gnu.org/software/diffutils, 2016.

[10] J. Svajlenko, J. F. Islam, I. Keivanloo, C. K. Roy, and M. M. Mia, "Bigclonebench (github)," https://github.com/clonebench/BigCloneBench, 2016.

[11] S. Livieri, Y. Higo, M. Matushita, and K. Inoue, "Very-large scale code clone analysis and visualization of open source programs using distributed ccfinder: D-ccfinder," in *ICSE*, 2007, pp. 106–115.

[12] H. Murakami, Y. Higo, and S. Kusumoto, "A dataset of clone references with gaps," in *MSR'14*, 2014, pp. 412–415.

[13] A. Charpentier, J.-R. Falleri, D. Lo, and L. Réveillère, "An empirical assessment of bellon's clone benchmark," in *Proceedings of the 19th International Conference on Evaluation and Assessment in Software Engineering*, ser. EASE '15. ACM, 2015, pp. 20:1–20:10.

[14] J. Svajlenko and C. K. Roy, "Evaluating modern clone detection tools," in *ICSME*, 2014, 10 pp.