Pyclone: A Python Code Clone Test Bank Generator

Schaeffer Duncan[1], Andrew Walker[1], Caleb DeHaan[1], Stephanie Alvord[1], Tomas Cerny[1], and Pavel Tisnovsky[2]

[1] Baylor University, Waco TX 76701, USA
[2] Red Hat, FBC II Purkyňova 97b, 612 00 Brno, Czech Republic
tomas_cerny@baylor.edu

Abstract. Code clones are fragments of code that are duplicated in the codebase of an application. They create problems with maintainability, duplicate buggy code, and increase the size of the repository. To combat these issues, there currently exists a multitude of programs to detect duplicated code segments. However, there are not many varieties of languages among the benchmarks for code clone detection tools. Without covering enough languages for modern software development, the development of code-clone detection tools remains stunted. This paper describes a novel tool that will take a seed of Python source code and generate Type 1, 2, and 3 code clones in Python. As one of the most used and rapidly-growing languages in modern software development, our testbed will provide the opportunity for Python code-clone detection tools to be developed and tested.

Keywords: Code-Clone · Benchmark · Quality Assurance · Testbed

1 Introduction

In many programs and code, developers are constantly trying to reduce the number of errors, bugs, and other issues brought upon by poor coding design. One such ingredient to bug-prone software is the inclusion of duplicated code segments or *code clones*. Code clones in programs introduce problems in the forms of bugs, unnecessarily large repositories, and difficulty in software maintenance. These can be introduced into the program through many avenues, including developers copying and pasting code, rather than putting in the effort to refactor code. Because of the pervasive nature of code clones in a repository, care must be taken when modifying a piece of code also to modify any clones. Without a reasonable way to manage code clones, bugs can persist in the application and require significantly more developer overhead to update the repository.

A handful of useful code clone detection tools currently exist to assist developers in locating code clones in their application. However, this field is stunted by the lack of code-clone benchmark applications for the different popular industry languages. Python has recently become one of the predominant languages for applications in the industry. However, a recent mapping study on existing

code clone detection tools [14] found that Python tools ranked 9th in the number of tools, with only 7 out of 67 tools able to detect Python, and no tools solely dedicated to Python code clone detection.

Given a general lack of benchmark applications, it would be beneficial to create a dynamic benchmark or test bank generator that can measure these tools' effectiveness. In this paper, we present a Python-based code clone generator, named Pyclone, that can produce a documented number of code clones to better gauge when seeded with a directory of Python code files how effective code clone detection tools are.

The rest of the paper is organized as follows. Section 2 presents background on code-clones. Section 3 presents background on existing mutation frameworks for code-clone detection tool evaluation. Section 3 presents our tool, Pyclone, and our process for the generation of the test bank. Section 4 discusses threats to validity. Finally, we conclude in Section 5.

2 Background

In this section, we will introduce the background to duplicated code within projects. In general, code clones or duplicate code segments are seen as poor coding practices. This is due to duplicate code being inefficient and because they introduce a level of variability, which is a breeding ground for bugs and unexpected errors. Code clones can introduce unpredictability when inconsistent changes are made to code duplicates. In general, there are four different types of code clones.

2.1 Basic Definitions

This subsection will define some general terms that will be used throughout the rest of the paper. These terms will be adapted from previous works/studies on code clones [2, 8, 9].

Code Fragment - A continuous segment of the source code, specified by *(l, s, e)*, including the source file l, the line the fragment starts on, s, and the line it sends on, e.

Code Clone - Two code fragments that have similarities either in their syntax and/or semantics.

2.2 Types of Clones

In addition to the existence of a generic "code clones," these clones are broken into four sub-categories of clones. In this subsection, we will define the general understandings of the four primary categories of code clones.

Type 1 code clones are said to be completely identical code fragments, disregarding comments and whitespace.

Type 2 code clones are said to be code fragments that are identical except they could have different variable and function names, along with different variable types and literals.

Type 3 code clones are said to be code fragments that are similar yet have some modifications consisting of either added/removed statements, different variable types/names, or differing function names.

Type 4 code clones are said to be code fragments that do not follow the same syntactical structure yet implement the same functionality.

Generally, these clone types are grouped by Types 1-3, which differ based on their text [7, 1] and Type 4, which is different only semantically [5].

Original Code	Type 1	Type 2	Type 3	Type 4

int x = 5;
int y = 0;
 	
// comment
while(y <= x){
  y++;
}
	int x = 5;
	int y = 0;
	
	while(y<=x){
	  y++;
	}
		int a = 5;
		int b = 0;
		
		while(b<=a){
		  b++;
		}
			double a = 5;
			double b = 0;
			
			while(b<=a){
			  b+= 1.0;
			}
				int a = (10/2);
				double b = 0;
				
				while(b <= 5){
				  b += 1.0;
				}

Table 1. Comparing/Contrasting Different Types of Code Clones

3 Related Works

There has been a handful of studies conducted in the pursuit to evaluate the validity of code clone detection tools independently. While developers of tools typically fall back to BigCloneEval [12], several alternatives to methods of validating code clone detection tools have been proposed.

Roy and Cordy [7] proposed a unique approach to evaluating code clone detection tools. They proposed a mutation and injection automatic framework that would evaluate code clone detection tools based on code clones' editing theory. Their preliminary approach focuses on the idea of their framework being injection-based, and their approach solely relies upon textual mutation of the code. They go on to express the need for a benchmark that can be used by developers and researchers to evaluate proposed code clone detection tools. In this work, they relay the importance of verifying clones needs to be able to test results yet require minimal effort thoroughly. This aspect of easily crafting a test bank based on seed code would allow a developer more options in terms of identifying more efficient ways to code.

Svajlenko and Roy [10] also had another study done on the evaluation of code clone detection tools, this time with a focus on a clone detection tool evaluation framework named BigCloneBench [11]. In this extension of their previous work, they use BigCloneBench to validate their own mutation framework discussed above. Their proposed tool validated the effectiveness of inject-based frameworks for code-clone tool verification. Their injection framework solely aims to solve the problem of Java code clone detection tools.

3.1 Need for Pyclone

Python has been steadily rising in popularity for over a decade now and has become one of the preeminent programming languages, especially in industry. One look at the Google search trends in Fig. 1 shows that Python is at its most popular ever. The TIOBE Programming Community index [13] is a good indicator of programming language popularity. The rating bases on the number of skilled engineers world-wide, courses, and third-party vendors. For August 2020, the top was C with 16.9%, Java with 14.4%, and Python with 9.6%. Analysis of 2020 Indeed.com job postings [4] show Python, followed by Java and JavaScript. For Microservice Architecture related jobs that drive the cloud-computing field [15], 48% of companies use Java, and 21% use Python. The top languages for containers are JavaScript, Java, Python, and PHP [3].

For code clone detection, there have been various tools that have been created to test the percentage of code clones within computer programs, and while there exist a multitude of code clone detection tools for C++ and Java, comparably, there are very few available for Python. A recent analysis of open-source tools and benchmarks for code clone detection [14] made clear that there exists a significant gap for the Python language. With there being no benchmark for Python code-clone detection, it falls on these tools to test using another benchmark, in another language, and while these tools claim equal performance across their supported languages, no method exists to verify their tools.

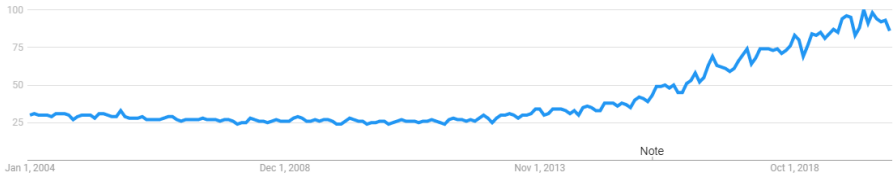

Fig. 1. Google Search Trends for Python Over Time.

To improve the quality of such detection tools as it pertains to the Python language, a standard set of code clones in a test bank needed to be created. The creation of clearly defined code clones would then set a benchmark for the quality of detection across the various tools.

4 Pyclone

In this section, we will detail the way Pyclone generates each type of code clone based upon its representation to the system. Having been developed in Python for testing Python code clone detection tools, Pyclone was written to create code clones based on a seed of Python files passed to it. It works based on the idea of mutation of Abstract Syntax Trees (AST) generated from the seed files.

4.1 Abstract Syntax Trees

The first step in mutating new code-clones is to scan the seed files and construct ASTs for all of the files. An AST is essentially a tree representation of abstract code structure. Each node in an AST is a different statement or condition in the code. If code is different between the two projects, then the ASTs between the two projects will also differ in the same locations, and vice versa.

With these ASTs, our tool can then modify these ASTs in order to change the underlying source code once the tool converts the AST back into Python code, giving us differing code samples.

The Python compiler uses ASTs to compile the Python code into bytecode. However, in order for Pyclone to manipulate the ASTs, it was required to find a more high-level representation of ASTs that can be manipulated within the tool itself.

Pyclone, therefore, uses a Python package called astor[3]. Astor allows for Pyclone to form ASTs based on valid Python files and modify them programmatically. These ASTs are manipulated as any other tree-based object in Python would be manipulated. Astor then handles the recreating code from the ASTs once the mutation is complete.

A key understanding of ASTs, which is used in all three type generations, is the idea of a "copyable" node in the AST. When we construct an AST, we must select a node that can constitute a fully complete and compilable code segment when taken alone. When constructing the ASTs from the seed files, we begin by utilizing the most common "copyable node" for code-clones, which is the *file-root* node. An AST containing an entire Python file is a fully-contained piece of code for code-clone detection purposes. It's worth noting that the code may not be runnable if it references classes or methods from other files or libraries, but it is fully compilable by Python.

The selection process is then further specified with a number of rules about which sub-nodes can be selected. For example, while a pure-method (one not existing as a class method) node is copyable without any additional steps, a class-method node must be copied within a duplicate of the seed class, along with any sub-methods that are called. Besides, the class must copy any necessary class fields. An example of the bare minimum needed to clone a class-method is seen in Fig. 2.

[3] https://pypi.org/project/astor/

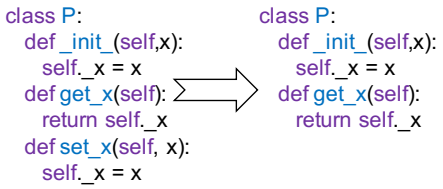

Fig. 2. Cloning a Class Method.

```
def compute_lcm(x, y):
  if x > y:
     greater=x
  else:
     greater= y
  while(True):
     if((greater% x== 0) and (greater% y== 0)):
        lcm = greater
        break
     greater+= 1
     return lcm
num1 = 54
num2 = 24
print("The L.C.M. is", compute_lcm(num1, num2))
```

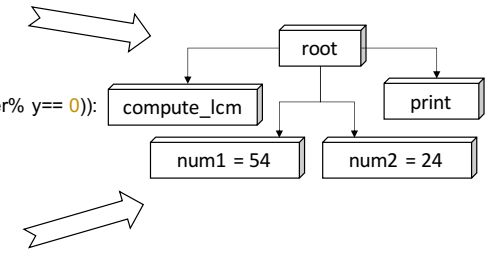

Fig. 3. First Order Nodes of a Python Code Sample.

Further detailed nodes such as individual *for*, *if* or *while* blocks must be examined carefully before being copied. For example, if a *for* block calls a method within its body, then that method must either be copied along with the *for* block.

Last to be considered are individual statements. For our purposes, we only look at statements which we call "first-order" statements. These statements are any that exist as a direct child of either a file-root node or a pure-method node. An example of possible "first-order" statements is shown in Fig. 3.

Once all nodes which are copyable are identified, the process of code-clone creation can begin.

4.2 Generation of Type 1

Creating Type 1 clones is straightforward. The tool essentially selects a random number of copyable nodes to construct new clones out of and directly dumps them into a new file. Occasionally we would modify the AST before constructing the new clone to add in comment nodes. Once the new files are created, we retroactively modify their whitespace as well.

Pyclone creates a directory named 'type1clones' and puts all of the Type 1 clones with identifier file names into this directory.

4.3 Generation of Type 2

Creating Type 2 code clones was a little different from Type 1 because Pyclone has to modify the ASTs before constructing the new clone.

After selecting a new set of copyable nodes to construct the clones out of, we then proceeded to modify them by symbol translation for Type 2 clones.

The process of symbol translation involves mapping function and variable names in the existing code block. Whenever Pyclone would come across a value representing a function name or variable name, it would be inserted into a map that maps previous names to their newly generated name. When traversing future nodes, if a name that had already been mapped is discovered, it is not inserted as a new entry into the map.

After the symbol mapping phase, we begin the symbol translation phase, which involves traversing the entire tree of the node that is being copied and searching for any names that are translated in the symbol table. If the name is found, then it is mapped to its new name.

After the entire tree has been changed, the AST was then converted back to code and returned in the form of a Python code file. Additionally, Type 2 clones' generation can also include steps from the Type 1 protocol, which includes adding comments or modifying whitespace.

Pyclone creates a directory named 'type2clones' and puts all of the Type 2 clones with identifier file names into this directory.

4.4 Generation of Type 3

Creating Type 3 code clones was more similar to the generation of Type 2 clones than Type 1 clones. Like with clone types, the tool then traversed the tree, however this time it was looking for nodes that represented boolean operators, mathematical operations, or comparing operations, as modifications to these types of nodes would generate code nearly similar to the original but with different functionality.

A static list of replacement values was created for each operator type to change these nodes, and then one was chosen at random to replace the original value. Each AST could also have nodes removed. If a node is removed, any children nodes are also removed (i.e., an entire *for* block can be removed). Lastly, each AST is potentially merged with another to add lines to the code-clone. When a new AST is added in, it will have different variable names than the original AST. To mitigate this, we perform symbol realignment on the merged in section. This process is similar to the symbol translation process in Type 2 clone generation. Each symbol in each of the ASTs is mapped from its name to its data type. Then the AST to be merged in is traversed, and each variable name is mapped to a name from the original AST that matches the data type. If no suitable replacement is found, then that statement is removed, or a variable declaration is appended to the top of the code block.

After going through the entire tree, and potentially applying Type 1-2 mutations as well, the AST is then converted back to code and written to a new Python file in a directory for Type 3 clones.

Pyclone creates a directory named 'type3clones' and puts all of the Type 3 clones with identifier file names into this directory.

5 Threats to Validity

There are two main threats to the validity of Pyclone's test bank. The first is that due to the nature of injection-based frameworks, it's possible that by injecting new code lines or removing them, an accidental clone was created with an unintended code segment. The second main threat to validity is that while the code clones are compilable by Python, they may not be runnable, which is an issue with code-clone detection tools that require a running program to detect similarities [6].

The main threat to the validity of Pyclone is the development of the code clones, and whether the clone detection tools will find them. Type 1 and Type 2 clones are fairly straightforward, as they are almost identical copies of the original code with changed variable names at most. However, the detection of Type 3 or Type 4 clone may depend on the tool's implementation for finding clones, as the given tool's definition of Type 3 or Type 4 clone may not be the same as the way our tool has created them, and so the clones may be misidentified or not caught at all.

6 Conclusion

This paper presented an overview of the types of code clones and the reasons for code-clone detection tools. We further explained that there exists a large gap in the code-clone detection research field for the Python coding language, partially due to the lack of a Python code-clone benchmark. We presented an injection-based mutation framework for generating a Python code-clone benchmark from an existing seed of Python files to mitigate this. This paper concludes one of the first steps forward in creating viable means to check Python code clone detection tools.

Acknowledgements

This material is based upon work supported by the National Science Foundation under Grant No. 1854049 and a grant from Red Hat Research https://research.redhat.com.

References

1. Aho, A.V., Lam, M.S., Sethi, R., Ullman, J.D.: Compilers: Principles, Techniques, and Tools (2Nd Edition). Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA (2006)
2. Bellon, S., Koschke, R., Antoniol, G., Krinke, J., Merlo, E.: Comparison and evaluation of clone detection tools. IEEE Transactions on Software Engineering **33**(9), 577–591 (Sep 2007). https://doi.org/10.1109/TSE.2007.70725
3. Datadog: 8 Emerging trends in container orchestration (December 2012), https://www.datadoghq.com/container-orchestration/
4. Dowling, L.: Top 7 programming languages of 2020 (2020), https://www.codingdojo.com/blog/top-7-programming-languages-of-2020
5. Gabel, M., Jiang, L., Su, Z.: Scalable detection of semantic clones. In: 2008 ACM/IEEE 30th International Conference on Software Engineering. pp. 321–330 (May 2008). https://doi.org/10.1145/1368088.1368132
6. Hamerly, G., Perelman, E., Lau, J., Calder, B.: Simpoint 3.0: Faster and more flexible program phase analysis. J. Instr. Level Parallelism **7** (2005)
7. Roy, C.K., Cordy, J.R.: A mutation/injection-based automatic framework for evaluating code clone detection tools. In: 2009 International Conference on Software Testing, Verification, and Validation Workshops. pp. 157–166 (April 2009). https://doi.org/10.1109/ICSTW.2009.18

8. Roy, C.K., Cordy, J.R.: A survey on software clone detection research. School of Computing TR 2007-541, Queen's University **115** (2007)
9. Sheneamer, A., Kalita, J.K.: A survey of software clone detection techniques. International Journal of Computer Applications **137**(10), 1–21 (March 2016), published by Foundation of Computer Science (FCS), NY, USA
10. Svajlenko, J., Roy, C.K.: Evaluating modern clone detection tools. In: 2014 IEEE International Conference on Software Maintenance and Evolution. pp. 321–330 (Sep 2014). https://doi.org/10.1109/ICSME.2014.54
11. Svajlenko, J., Roy, C.K.: Evaluating clone detection tools with bigclonebench. In: 2015 IEEE International Conference on Software Maintenance and Evolution (ICSME). pp. 131–140 (Sep 2015)
12. Svajlenko, J., Roy, C.K.: Bigcloneeval: A clone detection tool evaluation framework with bigclonebench. In: 2016 IEEE International Conference on Software Maintenance and Evolution (ICSME). pp. 596–600 (2016)
13. Tiobe Software B.V.: Tiobe index for august 2020 (2020), https://www.tiobe.com/tiobe-index/
14. Walker, A., Cerny, T., Song, E.: Open-source tools and benchmarks for code-clone detection: Past, present, and future trends. SIGAPP Appl. Comput. Rev. **19**(4), 28–39 (Jan 2020). https://doi.org/10.1145/3381307.3381310, https://doi.org/10.1145/3381307.3381310
15. Zavgorodnya, A., RubyGarage: Moving to microservices: Top 5 languages to choose from (2019), https://rubygarage.org/blog/top-languages-for-microservices