PAPER

Dataset of Functionally Equivalent Java Methods and Its Application to Evaluating Clone Detection Tools

Yoshiki HIGO[†a)], ***Senior Member***

SUMMARY Modern high-level programming languages have a wide variety of grammar and can implement the required functionality in different ways. The authors believe that a large amount of code that implements the same functionality in different ways exists even in open source software where the source code is publicly available, and that by collecting such code, a useful data set can be constructed for various studies in software engineering. In this study, we construct a dataset of pairs of `Java` methods that have the same functionality but different structures from approximately 314 million lines of source code. To construct this dataset, the authors used an automated test generation technique, `EvoSuite`. Test cases generated by automated test generation techniques have the property that the test cases always succeed. In constructing the dataset, using this property, test cases generated from two methods were executed against each other to automatically determine whether the behavior of the two methods is the same to some extent. Pairs of methods for which all test cases succeeded in cross-running test cases are manually investigated to be functionally equivalent. This paper also reports the results of an accuracy evaluation of code clone detection tools using the constructed dataset. The purpose of this evaluation is assessing how accurately code clone detection tools could find the functionally equivalent methods, not assessing the accuracy of detecting ordinary clones. The constructed dataset is available at github (https://github.com/YoshikiHigo/FEMPDataset).

key words: *functionally-equivalent methods, source code analysis, dataset, code clone*

1. Introduction

Current programming languages have a rich syntax, and there are many ways for developers to implement the functionality they need. For example, in the case of `Java`, the `for` statement, the `while` statement, recursive functions, `Stream`, etc. can be used to perform repetitive processing. In the refactoring patterns proposed by Fowler [1], the implementations before and after refactoring have the same external behavior, which means refactoring can be regarded as an implementation change of a functionality. Thus, there are countless ways to implement a certain functionality, and developers implement the necessary functionality according to their own preferences and/or the policies of their software development project.

The authors believe that there is a large amount of source code with the same functionality but different structures in open source software (hereafter referred to as *SFDS code*[*]), which can be a useful dataset for various studies in software engineering. For example, SFDS code can be used to evaluate code clone detection tools. Since it is desirable for source code that implement the same functionality to be detected as code clones, the performance of code clone detection tools can be evaluated by examining the degree to which SFDS code are detected as code clones. Moreover, by using SFDS code, we can investigate which implementations are superior in terms of running performance, such as memory usage and execution speed, and which implementations are superior in terms of software quality, such as ISO/IEC 25010 [2].

However, it is not easy to identify and collect SFDS code. It is not realistic to manually find SFDS code from open source software, and if existing code clone detection tools are used, only SFDS code that can be detected by existing code clone detection tools will be collected. Therefore, SFDS code collected in such a way cannot be used to evaluate code clone detection tools.

In this study, SFDS code to be collected are limited to pairs of methods that return the same output (return value) when the same inputs (arguments) are given (hereafter, *FE methods*[**]). The key idea of this study is to automatically obtain candidate pairs of FE methods using an automated test generation technique, `EvoSuite` by limiting the detection of SFDS code to the detection of FE methods. The obtained candidates of FE method pairs are manually verified to be truly functionally equivalent.

In this study, we selected `IJADataset` [3] as the target for detecting FE method pairs in `Java`. `IJADataset` consists of approximately 2.74 million source files (a total of approximately 314 million lines of source code) and includes approximately 23 million methods. From this dataset, 13,710 candidate FE method pairs were automatically detected, of which 2,194 were visually investigated. As a result, a dataset of 1,342 FE method pairs was constructed. Our dataset also includes 852 method pairs that were determined not to be functionally equivalent by visual inspection. The constructed dataset is available at `GitHub`[***].

We also evaluated code clone detection tools using the constructed dataset. As a result, we found that there are many FE method pairs that cannot be detected by the token-based clone detection technique, and that the detection techniques based on abstract syntax trees and deep learning have a strong tendency to incorrectly detect method pairs that are functionally inequivalent.

Manuscript received December 13, 2023.
Manuscript publicized February 21, 2024.

[†]The author is with the Graduate School of Information Science and Technology, Osaka University, Suita-shi, 565–0871 Japan.
a) E-mail: higo@ist.osaka-u.ac.jp

DOI: 10.1587/transinf.2023EDP7268

[*]Same-Functionality but Different Structures
[**]Functionally Equivalent
[***]https://github.com/YoshikiHigo/FEMPDataset

2. Definition of FE Methods

In this study, two methods are considered to be functionally equivalent if they return the same output (return value) when the same inputs (arguments) are given. Many code clone detection techniques measure the similarity of the source code of the target methods to determine them as code clones [4]–[6], but this study does not use such code similarity to determine whether given two methods are functionally equivalent or not. There are also techniques that determine code clones by using the state of processing (main memory state) of the target methods by running them [7], but this study does not use such a calculation process to determine whether given two methods are functionally equivalent or not.

In research of code clones, detected code clones are often classified according to their similarity as follows.

Type-1 The code snippets are entirely identical except for changes that may exist in the white spaces and comments.

Type-2 The structure of the code snippets is the same while the identifiers' names, types, white spaces, and comments differ.

Type-3 In addition to changes in identifiers, variable names, data types, and comments, some parts of the code can be deleted or updated, or some new parts can be added.

Type-4 Two code snippets have different texts but the same functionality.

FE methods detected in this study are code clones of a part of Type-4 according to the above classification.

3. Key Idea for Automatically Identifying Candidate FE Method Pairs

In this study, we automatically collect candidate pairs of FE methods by using the static features (method signatures) and dynamic behavior (test results) of `Java` methods. Whether or not the obtained candidate FE method pairs are truly functionally equivalent is investigated manually. Therefore, it is important to collect as many candidate FE method pairs as possible automatically.

The static features of the `Java` methods used in this study are the return value type and the parameter types. As the first step in obtaining candidate FE method pairs, methods with the same return value type and parameter types are classfied into the same group. We do not consider throwing exceptions as part of static features since throwing an exception or not can be checked as dynamic behavior.

The dynamic behavior of the `Java` methods used in this study is the result (success/failure) of the execution of a given test case. Test cases are generated from all methods in the same group using an automated test generation technique, `EvoSuite`. Test cases generated by automated test generation techniques have the property that the test cases always succeed. For every pair of methods, the generated test cases are executed against each other to automatically determine if the method pair has the same behavior. The key idea of this research is that if method A succeeds in all tests generated from method B and method B succeeds in all tests generated from method A, then the behavior of methods A and B are equivalent to some extent and their functions may be equivalent.

Based on this key idea, a dataset of FE method pairs is constructed by automatically obtaining pairs of methods from a large amount of open source software that have been successfully executed for cross-testing and visually investigating them. Although methods with identical or similar source code may be functionally equivalent, such methods can be detected by existing code clone detection tools [8]. The objective of this study is to build a dataset of FE method pairs that are not similar in source code.

4. Procedure of Dataset Construction

In this study, the following procedure is used to construct a dataset of FE method pairs.

STEP-1 classifying the methods included in the target projects.

STEP-2 generating test cases from each method.

STEP-3 running test cases against the other method of the pair to obtain candidate FE method pairs.

STEP-4 browsing the source code of each candidate FE method pair to determine whether the pair is truly functionally equivalent.

Figure 1 shows an overview of the above construction procedure. `STEP-1` through `STEP-3` are performed automatically using a tool that we developed, and only `STEP-4` is performed manually. The details of each step are described below.

4.1 STEP-1

`STEP-1` analyzes the source code of the target projects to extract methods, and classifies the extracted methods based on their return value type and parameter types. When extracting methods, the following information is retrieved for each method and registered in the database.

- method name,
- return value type and parameter types,
- (original) source code,
- normalized source code,
- the number of statements and conditional predicates,
- file path,
- start line and end line.

In the normalized source code, all variable names have been renamed to special names. A normalization example is shown in Fig. 2 (b). This normalization allows a group of methods that have the same structure but differ only in variables to be treated as a single method. This normalization eliminates the need to handle each method that has the same structure but differs only in variables individually after

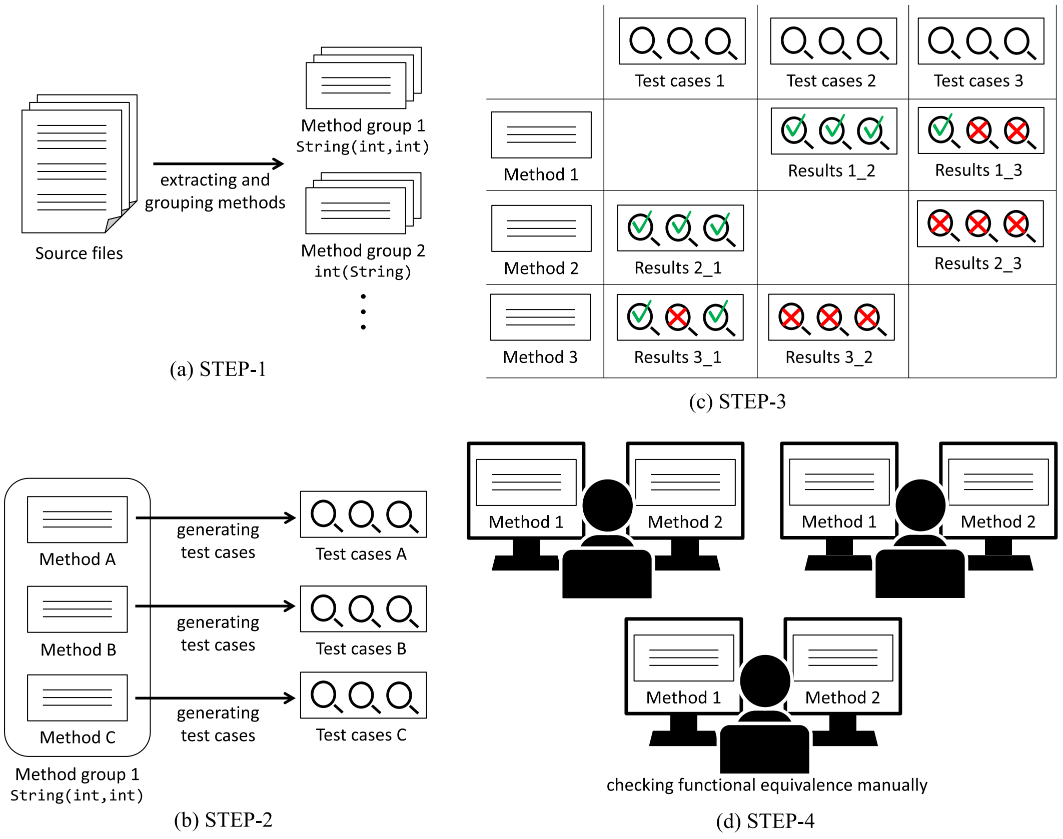


Fig. 1 Steps to obtain pairs of functionally equivalent Java methods

STEP-2, allowing more efficient processing.

Not all methods that exist in the target project are extracted. In this study, methods with the following characteristics are not considered for extraction.

- Methods including reference types defined in neither java.lang nor java.util in their return value, parameters, or method bodies.
- Methods whose return type is void.
- Methods including only a single program statement.

The reason for using the first condition is that in the case of using reference types that are not in the java.lang package, it is necessary to write import statements at the beginning of the source file, or to write the reference types by their fully qualified names. In addition, if the reference type is not included in the standard Java library, it is necessary to prepare its class file (jar file), which requires further compilation preparation. The reason why the reference types included in the java.util package are considered as methods for extraction is because frequently used types such as java.util.List and java.util.Set are included in this package. The authors believe that handling those reference types in the java.util package will dramatically increase the number of methods that are extracted.

The reason for using the second condition is that it is difficult to automatically determine which value within a method is the final results of the method's computation for a method whose return value is void. If the return type is not void, the method's return value can be judged to be the final results of the method's computation.

The third condition is used because the source code of Java contains many setters and getters that have only a single program statement, and they are inappropriate as targets for the detection of SFDS code.

The classification of the extracted methods is based on the return value type and parameter types of the methods. Methods that have exactly the same return type and parameter types are classified into the same group. After all methods have been classified, if there are multiple methods in each group with exactly the same normalized source code, only one of them is retained in the group. The reason for this is that it is no doubt that methods with the identical code have the same behavior, and the detection of such method pairs with the same implementation is not appropriate for the purpose of this study. Note that groups consisting of only a single method are not subject to processing after STEP-2.

4.2 STEP-2

In STEP-2, each method is cut into a file and its test cases are generated. The following processing is performed when the methods are cut into files.

- Insert 'import java.util.*;' at the beginning of the file. This is to enable compilation even if the target

```
package com.intellij.openapi.util.text;
…
public class StringUtil extends StringUtilRt {
…
  @Contract(pure = true)
  public static @NotNull String repeat(@NotNull String s, int count) {
    if (count == 0) return "";
    assert count >= 0 : count;
    StringBuilder sb = new StringBuilder(s.length() * count);
    for (int i = 0; i < count; i++) {
      sb.append(s);
    }
    return sb.toString();
  }
…
```

(a) Original method

```
String $method(String $variable,int $variable){
  if ($variable == 0)   return "";
  assert $variable >= 0 : $variable;
  StringBuilder $variable=new StringBuilder($variable.length() * $variable);
  for (int $variable=0; $variable < $variable; $variable++) {
    $variable.append($variable);
  }
  return $variable.toString();
}
```

(b) Normalized method

```
import java.util.*;
public class Target {
  String __target__(String s,int count){
    if (count == 0)   return "";
    assert count >= 0 : count;
    StringBuilder sb=new StringBuilder(s.length() * count);
    for (int i=0; i < count; i++) {
      sb.append(s);
    }
    return sb.toString();
  }
}
```

(c) Method cut into a single file

Fig. 2 Example of processing for the source code of the target method

method uses a class in the ‘`java.util`’ package.

- Enclose the target with a class declaration. Currently, the class name ‘`Target`’ is used. The name of the target method is also changed to ‘`__target__`’. This is to ensure uniform treatment of all target methods in the scripts created by the authors.
- Remove annotations and ‘`static`’ from the signature of the target method. This is also done to ensure that all target methods are handled uniformly in the script.

Figure 2 (c) shows the source code of the method of Fig. 2 (a) after it has been cut into a file. From this figure, it can be seen that the file contains only the target method, the class and method names are unified, and the annotations and `static` modifiers attached to the method signatures are removed.

Next, test cases are generated for each method that is cut into files. Although we use `EvoSuite` [9] to generate test cases, other test generation tools such as `Randoop` [10] and `Agitar` [11] can also be used. In this experiment, if five or more test cases were generated from a method, the method was used in `STEP-3`. The reason for this restriction is that `EvoSuite` may not be able to generate enough test cases depending on the structure of the target method. We considered that if sufficient test cases could not be generated, it would be inappropriate to use such insufficient test cases to detect candidates for functionally equivalent method pairs.

The files from which each method is extracted are compiled individually just before `EvoSuite` is executed. Since each file includes only a single method, the compilation fails if the method uses a class field or calls other methods that were originally defined in the same class. `EvoSuite` is not executed for methods that fail to compile.

In `STEP-1`, methods with no parameters are not explicitly excluded, but methods where five or more test cases are not generated are excluded in `STEP-2`. Since no more than five test cases are generated from a method with no parameters, all methods with no parameters are excluded in `STEP-2`.

4.3 STEP-3

In `STEP-3`, tests are executed mutually for each method that belongs to the same group and has successfully generated five or more test cases. In Fig. 1(c), test cases are executed against each other for three methods: `Method-A`, `Method-B`, and `Method-C`. `Method-A` succeeds in all test cases generated by `Method-B`, and `Method-B` also succeeds in all test cases generated by `Method-A`. Therefore, `Method-A` and `Method-B` are candidates for functionally equivalent method pairs. `Method-C` did not succeed in one of the test cases generated from `Method-A`, and it did not succeed in all the test cases generated from `Method-B`. Therefore, the pair of `Method-A` and `Method-C`, and the pair of `Method-B` and `Method-C` are not candidates for functionally equivalent method pairs.

4.4 STEP-4

`STEP-4` visually checks the source code of the candidate functionally equivalent method pairs obtained with `STEP-3` to see if they are truly pairs of methods with the same behavior.

5. FEMP Dataset

In this section, we describe the constructed dataset (hereafter referred to as `FEMP` dataset[†]). The `FEMP` dataset was constructed for the source code included in `IJADataset` [3]. `IJADataset` consists of approximately 2.74 million source files, totaling 314 million lines of source code, and contains approximately 23 million methods. In `STEP-1`, 257,012 methods were extracted and they were classified into 14,030 groups. In `STEP-2`, five or more test cases were successfully generated from 27,759 methods. In `STEP-3`, we obtained 13,710 candidate functionally equivalent method pairs.

The visual verification in `STEP-4` was performed by three master’s course students belonging to the Graduate School of Information Science and Technology, Osaka University. The three students had programming experience using Java. First, the three students individually viewed the source code of each candidate pair and judged whether they

[†]Functionally Equivalent Method Pair Dataset

```
double min(double y0, double y1, double y2){
  if ((y0 <= y1) && (y0 <= y2)) {
    return y0;
  } else if (y1 <= y2) {
    return y1;
  } else {
    return y2;
  }
}
```

(a) method min

```
double minimum(double val1, double val2, double val3){
  if (val1 < val2) {
    return val1 < val3 ? val1 : val3;
  } else {
    return val2 < val3 ? val2 : val3;
  }
}
```

(b) method minimum

Fig. 3 Example of functionally equivalent method pair

```
boolean ArrayEquals(boolean[][] a,boolean[][] b){
  if (a.length < 1)   return false;
  if (b.length < 1)   return false;
  if (a.length != b.length)   return false;
  for (int y=0; y < a.length; y++) {
    if (a[y].length != b[y].length)     return false;
    for (int x=0; x < a[y].length; x++) {
      if (a[y][x] != b[y][x])       return false;
    }
  }
  return true;
}
```

(a) method ArrayEquals

```
boolean equals(boolean[][] m1,boolean[][] m2){
  if (m1.length != m2.length)   return false;
  for (int i=0; i < m1.length; i++) {
    if (m1[i].length != m2[i].length)     return false;
    for (int j=0; j < m1[i].length; j++) {
      boolean b1=m1[i][j];
      boolean b2=m2[i][j];
      if (b1 != b2)       return false;
    }
  }
  return true;
}
```

(b) method equals

Fig. 4 Example of functionally inequivalent method pair

were equivalent or not. Next, the three students discussed each candidate pair that they evaluated differently, to determine whether they were functionally equivalent or not.

However, since it is not realistic to visually check all 13,710 method pairs obtained in the STEP-3, the following procedure was used to extract a part of them.

1. Initialize the target method pair *P* and the set of methods *M* in *P* as empty, respectively.
2. Arrange the 13,710 method pairs in ascending order by ID[†], and perform the following processes in order.
   - If both methods comprising the method pair are not included in *M*, the method pair is added to *P* and both methods are added to *M*.
   - If at least one method comprising the method pair is contained in *M*, nothing is done.

After the above procedure, 2,194 method pairs were included in *P*. Those 2,194 method pairs all consisted of different methods. For those 2,194 method pairs, each student took 44 hours and 48 minutes, 33 hours and 19 minutes, and 43 hours and 25 minutes, respectively, to determine whether they were functionally equivalent or not. Subsequently, the discussion of each method pair, for which the evaluation was divided, took 9 hours and 28 minutes in total.

STEP-4 resulted in 1,342 of the 2,194 method pairs being determined to be functionally equivalent, with the remaining 852 not being functionally equivalent. The number of method pairs that were discussed by three students with different evaluations was 296. This dataset is available on github[††].

Figure 3 is an example of a method pair that was determined to be functionally equivalent in STEP-4. Both methods implement the function of returning the smallest value among the three double values given as parameters. The method `min` implements this function only using the `if`-statement, while the method `minimum` implements it using the `if`-statement and the ternary operator. On the other hand, Fig. 4 is an example of a method pair whose functions were determined to be not equivalent in STEP-4. Both methods determine the equivalence of two two-dimensional arrays given as parameters. The functionality differs in that the method `ArrayEquals` returns `false` if an empty array is given, while the method `equals` returns `true` if an empty array is given. EvoSuite generated nine test cases for the method `ArrayEquals` and eight test cases for the method `equals`, but no test cases were generated to find the functional differences between the two methods[†††].

Next, we describe the characteristics of the method signatures in the FEMP dataset. The 1,342 functionally equivalent method pairs consist of 468 signatures, and the 852 functionally inequivalent method pairs consist of 338 signatures. In the dataset, 713(=53.1%) functionally equivalent and 317(=37.2%) functionally inequivalent method pairs include only primitive types such as `int` and `char` in their return value type and parameter types. Table 1 shows the top 10 most frequent signatures of functionally equivalent and functionally inequivalent method pairs in the FEMP dataset. This table also shows that the functionally equivalent method pairs tend to be composed of primitive types only. This fact means that the proportion of method pairs that are determined to be functionally inequivalent in the dataset construction STEP-4 is higher when non-primitive types are included in the signatures. This indicates that it is more difficult to generate sufficient test cases when non-primitive types are included in the signature than when only primitive types are

[†]The 13,710 method pairs obtained in the STEP-3 have unique integer values as their IDs.
[††]https://github.com/YoshikiHigo/FEMPDataset
[†††]FEMP dataset also includes test cases generated by Evosuite.

Table 1 Top 10 signatures of functionally equivalent and inequivalent method pairs

Functionally equivalent method pairs			Functionally inequivalent method pairs		
# pairs	Signature	Only primitive	# pairs	Signature	Only primitive
60	`boolean(byte[], byte[])`	√	69	`boolean(String, String)`	
49	`boolean(char)`	√	40	`String(String, String, String)`	
26	`boolean(String, String)`		23	`int(String, int)`	
25	`boolean(int)`	√	18	`double(double)`	√
22	`String(String, String, String)`		17	`boolean(char)`	√
22	`boolean(int[], int[])`	√	14	`int(Object, Object)`	
22	`int(byte[], int)`	√	14	`int(String, String)`	
22	`int(int, int)`	√	12	`String(String, int)`	
21	`int(int)`	√	11	`boolean(byte[], byte[])`	√
20	`double(double[])`	√	11	`int(int, int)`	√

included.

6. Accuracy Evaluation of Clone Detection Tools

Herein, we describe the results of an accuracy evaluation of clone detection tools as an example of utilization of the FEMP dataset. The purpose of this evaluation is assessing how accurately code clone detection tools could find the functionally equivalent methods, not assessing the accuracy of detecting ordinary clones. In this evaluation, the 1,342 method pairs determined to be functionally equivalent in STEP-4 should be detected as clone pairs, and the 852 method pairs determined not to be functionally equivalent should not be detected as clone pairs. The former set is called *EMP* (*Equivalent Method Pairs*) and the latter set is called *IMP* (*Inequivalent Method Pairs*).

6.1 Clone Detection Tools to be Evaluated

Three clone detection tools were targeted in this evaluation: NIL [12], InferCode [13], and ASTNN [14].

NIL quickly identifies possible clone candidate method pairs using the inverted index and N-gram of the lexical sequence obtained from the target source code. It applies the longest common subsequence algorithm to those candidates to determine whether they are clone pairs or not. NIL is a tool for detecting large-variance clones, which are difficult to detect using conventional clone detection techniques. In the comparison with other clone detection tools LVMapper [15] and CCAligner [16] performed on two open source systems, the number of large-variance clones detected by NIL was 354 and 398 (86% and 88% precisions), whereas LVMapper detected 355 and 389 (64% and 60% precisions) and CCAligner detected 184 and 284 (43% and 49% precisions).

InferCode is a pre-training model using a convolutional neural network [17] based on abstract syntax trees and tree structures. It can be used for unsupervised learning tasks such as source code clustering and supervised learning tasks such as source code classification. Bui et al. have implemented clone detection in InferCode as an unsupervised learning task. In this comparison, we use InferCode as a clone detection technique based on unsupervised learning. The clone detection uses cosine similarity as the similarity between the two methods. Bui et al. set a threshold of 0.8 for the cosine similarity to be considered as a clone pair, and conducted experiments on BigCloneBench [18] and OJClone [19]. For BigCloneBench [18], the precision and recall were 90% and 56%, respectively, and for OJClone [19], the precision and recall were 61% and 70%, respectively.

ASTNN is a model based on abstract syntax trees and regression neural networks. It learns lexical information contained in the target source code and syntactic information per program statement. For clone detection, ASTNN outputs a value between 0 and 1. Two input methods are determined to be a clone pair if this value is greater than or equal to a threshold value. Zhang et al. experimented on BigCloneBench and OJClone with a threshold value of 0.5. For OJClone, precision and recall were 98.9% and 92.7%, respectively. For BigCloneBench, the detection accuracy was evaluated for each clone category. The results showed that for Weakly Type-3/Type-4, which are clones with a syntactic similarity of less than 50%, precision and recall were 99.8% and 88.3%, respectively.

In the papers of NIL, InferCode, and ASTNN ([12]–[14]), those tools are evaluated using BigCloneBench [18] and OJClone [19]. BigCloneBench categorizes all functions into one of 43 different categories and then assumes that all methods that are assigned to a functionality are also clones of each other. The methods are not functionally equivalent and this is not claimed. Also, OJClone is a dataset built from competitive programming code, which is different in nature from the code included in OSS.

6.2 How to Detect Clones

Herein, we explain how we configured the three tools to detect code clones.

We installed NIL according to the instructions in GitHub[†] of NIL. Clone detection was performed by outputting each method in *EMP* and *IMP* as a separate file and giving these two files to NIL as clone detection targets. In other words, we ran NIL 2,194 times (1,342 times for *EMP* and 852 times for *IMP*). Because some methods are small

[†]https://github.com/kusumotolab/NIL

in size, the minimum number of lines and minimum number of tokens for a method to be detected as a clone were both set to 1[†].

We installed `InferCode` according to the instructions provided in `GitHub`[††] of `InferCode`. We used `InferCode` to obtain vector data for each method in *EMP* and *IMP*, and detected clones by changing the threshold of the cosine similarity of the method pairs from 0 to 1 in 0.001 steps.

We obtained a pre-trained model of `ASTNN` according to the description in `ASTNN`'s `GitHub`[†††]. Next, the 1,342 method pairs in *EMP* were divided into 10 blocks. The blocks were classified so that the ratio of fine tuning, validation, and test data was 8:1:1, and clone detection using `ASTNN` was performed so that all blocks were once test data. Since the output of `ASTNN` is a number between 0 and 1, the threshold for being a clone was varied from 0 to 1 in 0.001 increments, and the results were evaluated[††††].

6.3 Evaluation Measures for Clone Detection Results

Three measures, $recall^E(t)$, $recall^I(t)$, and $accuracy(t)$, were used to evaluate the clone detection results of tool t in this evaluation. First, the definitions of $EMP(t)$ and $IMP(t)$ used in these evaluation measures are as follows.

$EMP(t)$ a set of method pairs in *EMP* that have been correctly determined to be functionally equivalent (detected as clones) by detection tool t.

$IMP(t)$ a set of method pairs in *IMP* whose functions are correctly detemined to be not functionally equivalent by detection tool t (not detected as clones).

Using the above definitions, we define $accuracy(t)$, $recall^E(t)$, and $recall^I(t)$ as follows. Note that $|A|$ denotes the number of elements in the set A.

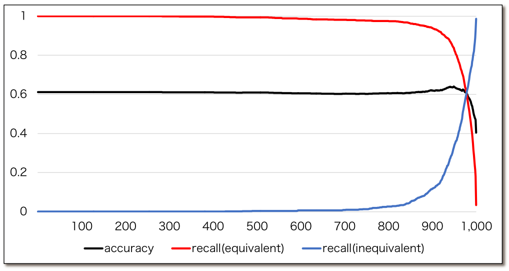
$$recall^E(t) = \frac{|EMP(t)|}{|EMP|}$$

$$recall^I(t) = \frac{|IMP(t)|}{|IMP|}$$

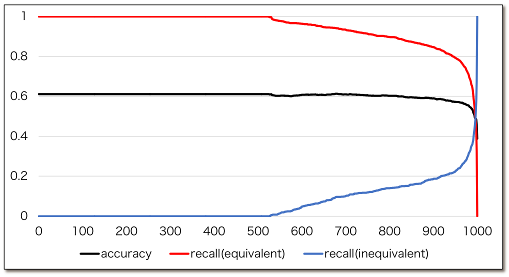
$$accuracy(t) = \frac{|EMP(t)| + |IMP(t)|}{|EMP| + |IMP|}$$

6.4 Detection Results

The detection results of `NIL` are shown in Table 2. The $recall^E(NIL)$ was 34.5% (=463/1,342), $recall^I(NIL)$ was 72.54% (=618/852), and $accuracy(NIL)$ was 49.27% (=1,081/2,194). These results indicate that clone detection by `NIL` has many omissions in finding functionally equivalent methods, and also includes some false positives.



(a) `InferCode`



(b) `ASTNN`

Fig. 5 Detection accuracy vs. threshold. The Y-axis represents a percentage in the range 0 to 100. For the X axis, the left end of the horizontal axis represents a threshold of 0, and the right end a threshold of 1.

Table 2 Detection results of `NIL`

		Detection results	
		EMP	*IMP*
True value	*EMP*	463	879
	IMP	234	618

The results of the `InferCode` evaluation are shown in Fig. 5 (a). When the threshold is 0.557 or less, $recall^E(InferCode)$ is more than 99%, but $recall^I(InferCode)$ is less than 0.35% at the same time. In other words, almost all functionally equivalent method pairs are detected as clones, but almost all functionally inequivalent method pairs are also detected as clones. Increasing the threshold value improves $recall^I(InferCode)$, but worsens $recall^E(InferCode)$ at the same time. The value of $accuracy(InferCode)$ was the highest when the threshold value was 0.949, which was 64.04%. At this time, $recall^E(InferCode)$ was 83.83% and $recall^I(InferCode)$ was 32.86%.

The evaluation results of `ASTNN` are shown in Fig. 5 (b). Overall, the shape of the graph of `ASTNN` is similar to that of `InferCode`. For thresholds below 0.531, $recall^E(ASTNN)$ is greater than 99%, while $recall^I(ASTNN)$ is only 0.59% at the same time. At the threshold values of 0.677~0.681, $arrucary(ASTNN)$ has the highest value of 61.30%, at which $recall^E(ASTNN)$ and $recall^I(ASTNN)$ have values of 94.19% and 9.62%, respectively.

We found that the clone detection tool `NIL`, which is based on lexical sequences, cannot detect many functionally equivalent method pairs as clones and it does not have a high ability to detect functionally equivalent methods. We also found that `InferCode` and `ASTNN`, clone detection tools

[†]However, since `NIL` is internally processed using `N-gram` with N=5, a minimum of five words are required to be detected as a clone.

[††]https://github.com/bdqnghi/infercode

[†††]https://github.com/zhangj111/astnn

[††††]The hyperparameters used for fine tuning were: batch size:32, epoche:5, learning size:2e-3, vector size of word2vec:128, hidden dimension:100, encode dimension:128.

based on abstract syntax trees and deep learning, can detect functionally equivalent method pairs as clones, but they also detect functionally inequivalent methods as clones. From the above results, we conclude that new methods need to be developed to properly find functionally equivalent method pairs.

7. Related Work

This research is inspired by the literature [20]. In the literature, the idea of obtaining a set of functionally equivalent methods by executing the generated test cases against each other is presented[†], and a dataset of functionally equivalent method sets was constructed from the Borge's dataset [21]. The differences between this paper and the literature are as follows.

- The Borge's dataset [21] includes approximately 36 million lines of code, whereas the `IJADataset` used in this paper has approximately 314 million lines of code. There is also a difference in the size of the constructed datasets: the dataset constructed in the literature [20] includes 276 functionally equivalent method sets, while the dataset in this paper contains 1,342 functionally equivalent method pairs.
- While all methods that were able to generate tests were subject to test execution in the literature [20], in this paper, test execution was not performed when the number of test cases generated was less than five. The reason for this is that in the process of constructing the dataset in the literature [20], when the number of tests generated was small, in most cases the methods were judged to be not functionally equivalent by the human eye even if mutual execution of test cases was successful.
- In the dataset construction process, visual checks were performed by one author in the literature [20], whereas in the dataset construction process in this paper, the final decision on whether the functions are equivalent was made after independent evaluation and discussion by three persons.
- This paper evaluates three code clone detection tools as an example of the use of the constructed dataset, but no such use case is given in the literature [20].

Svajlenko et al. constructed the `BigCloneBench` dataset [22]. There are only 43 different functionalities in the current `BigCloneBench` version. At the end of the dataset construction procedure, manual validation was performed. However, the target of the human validation work is the `Java` methods discovered by keywords and code patterns. Therefore, functionally equivalent methods that are not found by keywords and code patterns are not included in the `BigCloneBench` dataset. In addition, the target methods were not executed during the dataset construction procedure, and no functional equivalence checks were performed from a dynamic perspective.

Liu et al. constructed a dataset of functionally equivalent programs using data from past programming competitions. They collected functionally equivalent programs for about 5,000 questions [23]. In this dataset, programs developed by multiple users to a question are treated as functionally equivalent programs. Zhao et al. publish a dataset of programs with the same functionality in `Google Code Jam` [19], and Mou et al. also publish a dataset of programs submitted to programming education support systems [17]. On the other hand, unlike their datasets, our dataset is not a program for competitive programming, but source code with functionally equivalent features included in the OSS.

Rabin et al. have developed a tool, ProgramTransformer, which changes the structure of a given program [24]. The tool possesses a number of rules for changing the structure of a program, and changes the structure based on those rules. For example, it automatically rewrites repetitions described by `for`-statements into `while`-statements, and changes variable names.

8. Threats To Validity

Herein, we describe threads to validity in this research.

8.1 Code Normalization

The normalization in `STEP-1` prevents the proposed procedure in Sect. 4 from outputting Type-1 and Type-2 clones as functionally equivalent method pairs. This normalization also considerably reduces the number of methods targeted after `STEP-2`, which considerably improves the overall processing speed of the proposed procedure.

However, there is a negative aspect to this normalization. Figure 6 shows two artificial methods that end up in exactly the same source code due to the normalization. These methods are not simple Type-2 clones because they perform semantically different operations. Therefore, these methods can be included in the dataset constructed in this study. However, since the source codes of these two methods are exactly the same after normalization, one of the two methods will be removed in `STEP-1` if both of them exist in

```java
boolean out(int min, int max, int i){
  if (i < min && i > max) return true;
  return false;
}
```

(a) method out

```java
boolean in(int min, int max, int i){
  if (i < max && i > min) return true;
  return false;
}
```

(b) method in

Fig. 6 An artificial example of two methods that are semantically processed differently, although the normalization in `STEP-1` results in exactly the same source code for them.

[†]In the literature [20], Higo et al. obtained a set of functionally equivalent methods, whereas in this paper we determine functional equivalence by pairs of methods rather than a set.

the target source code. The proposed procedure in Sect. 4 is a way to construct a large dataset as efficiently as possible, and does not detect all functionally equivalent methods that exist in the target source code. Therefore, it is not a major problem to miss such corner-case methods.

8.2 Human Judgement

In `STEP-4`, human judgements whether the target method pairs are truly functionally equivalent or not. Since the judgments were made by human, it was not possible to completely eliminate subjectivity, and the possibility of judgment errors cannot be denied. In order to minimize these possibilities as much as possible, this research decided to obtain consensus among three persons instead of two.

9. Conclusion

In this study, we identified functionally equivalent method pairs by automatically generating test cases for `Java` methods extracted from open source software and executing the generated test cases against each other. Functionally equivalent method pairs were extracted from the 314 million lines of `Java` source code included in the `IJADataset` dataset. As a result, we obtained 13,710 candidate functionally equivalent method pairs. Of these, 2,194 method pairs were visually verified, and 1,342 functionally equivalent method pairs were obtained. This study also used this dataset to evaluate the accuracy of three code clone detection tools. This dataset can also be used to examine code quality, such as code maintainability and understandability.

The current challenge is that it takes a very long time to find candidate functionally equivalent method pairs, since all combinations of methods with equal return value and parameter types that could generate tests are tested against each other. In this experiment, it took approximately 40 days for `STEP-3` alone. In the future, we plan to introduce heuristics to reduce the number of method combinations that need to be executed mutually in order to find candidate functionally equivalent method pairs more quickly.

The large percentage (852/2,194=38.8%) of method pairs that were determined to be functionally inequivalent by visual inspection also needs to be improved. If there are few method pairs that are determined to be functionally inequivalent by visual inspection (if the mutual execution of test cases sifts out most of the methods that are not functionally equivalent), then the need for visual inspection would be eliminated and a large data set could be created almost automatically. We are aiming for higher quality automatic test generation by improving the test generation algorithm in `EvoSuite`.

Acknowledgments

This research was supported by JSPS KAKENHI Japan (JP20H04166, JP21K18302, JP21K11829, JP21H04877, JP22H03567, JP22K11985).

References

[1] M. Fowler, "Refactoring: Improving the Design of Existing Code," Addison-Wesley Longman Publishing Co., Inc., USA, 1999.
[2] ISO/IEC 25010, "ISO/IEC 25010:2011, systems and software engineering - systems and software quality requirements and evaluation (square) - system and software quality models," 2011.
[3] J. Svajlenko, "Ijadataset 2.0 + bigclonebench samples." https://1drv.ms/u/s!AhXbM6MKt_yLj_N3FAIGw3CJb1JGOg?e=NsP59Z, 2020.
[4] Q.U. Ain, W.H. Butt, M.W. Anwar, F. Azam, and B. Maqbool, "A systematic review on code clone detection," IEEE Access, vol.7, pp.86121–86144, 2019.
[5] 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.
[6] M. Zakeri-Nasrabadi, S. Parsa, M. Ramezani, C. Roy, and M. Ekhtiarzadeh, "A systematic literature review on source code similarity measurement and clone detection: Techniques, applications, and challenges," Journal of Systems and Software, vol.204, p.111796, 2023.
[7] H. Kim, Y. Jung, S. Kim, and K. Yi, "Mecc: Memory comparison-based clone detector," Proc. 33rd International Conference on Software Engineering, ICSE '11, New York, NY, USA, pp.301–310, Association for Computing Machinery, 2011.
[8] K. Inoue and C.K. Roy, "Code Clone Analysis: Research, Tools, and Practices," Springer, Singapore, 2021.
[9] Evosuite, "Evosuite: Automatic test suite generation for java," https://www.evosuite.org/, 2021.
[10] Randoop, "Randoop: Automatic unit test generation for java." https://randoop.github.io/randoop/, 2022.
[11] A. Technologies, "Agitarone." http://www.agitar.com/solutions/products/agitarone.html, 2015.
[12] T. Nakagawa, Y. Higo, and S. Kusumoto, "NIL: Large-Scale Detection of Large-Variance Clones," Proc. 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering, ESEC/FSE 2021, pp.830–841, 2021.
[13] N.D.Q. Bui, Y. Yu, and L. Jiang, "Infercode: Self-supervised learning of code representations by predicting subtrees," 2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE), pp.1186–1197, 2021.
[14] J. Zhang, X. Wang, H. Zhang, H. Sun, K. Wang, and X. Liu, "A novel neural source code representation based on abstract syntax tree," 2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE), pp.783–794, 2019.
[15] M. Wu, P. Wang, K. Yin, H. Cheng, Y. Xu, and C.K. Roy, "LVMapper: A Large-Variance Clone Detector Using Sequencing Alignment Approach," IEEE Access, vol.8, pp.27986–27997, 2020.
[16] P. Wang, J. Svajlenko, Y. Wu, Y. Xu, and C.K. Roy, "CCAligner: A Token Based Large-Gap Clone Detector," 2018 IEEE/ACM 40th International Conference on Software Engineering (ICSE), pp.1066–1077, 2018.
[17] L. Mou, G. Li, L. Zhang, T. Wang, and Z. Jin, "Convolutional neural networks over tree structures for programming language processing," Proc. Thirtieth AAAI Conference on Artificial Intelligence, AAAI'16, vol.30, no.1, pp.1287–1293, AAAI Press, 2016.
[18] J. Svajlenko, "Bigclonebench." https://github.com/clonebench/BigCloneBench, 2022.
[19] G. Zhao and J. Huang, "Deepsim: Deep learning code functional similarity," Proc. 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering, ESEC/FSE 2018, New York, NY, USA, pp.141–151, Association for Computing Machinery, 2018.
[20] Y. Higo, S. Matsumoto, S. Kusumoto, and K. Yasuda, "Constructing dataset of functionally equivalent java methods using automated test

generation techniques," Proc. 19th International Conference on Mining Software Repositories, pp.682–686, Association for Computing Machinery, 2022.

[21] H. Borges, A. Hora, and M.T. Valente, "Understanding the factors that impact the popularity of github repositories," 2016 IEEE International Conference on Software Maintenance and Evolution (ICSME), pp.334–344, IEEE, Oct. 2016.

[22] J. Svajlenko and C.K. Roy, "Bigcloneeval: A clone detection tool evaluation framework with bigclonebench," 2016 IEEE International Conference on Software Maintenance and Evolution (ICSME), IEEE, pp.596–600, Oct 2016.

[23] H. Liu, M. Shen, J. Zhu, N. Niu, G. Li, and L. Zhang, "Deep learning based program generation from requirements text: Are we there yet?," IEEE Trans. Softw. Eng., vol.48, no.4, pp.1268–1289, 2020.

[24] M.R.I. Rabin and M.A. Alipour, "Programtransformer: A tool for generating semantically equivalent transformed programs," Software Impacts, vol.14, p.100429, 2022.

Yoshiki Higo received his master's degree and Ph.D degree in information science and technology from Osaka University in 2004 and 2006, respectively. At present he is a professor at Osaka University. His research interests include mining software repositories, program analysis, and automated program repair. He is a member of IEEE, IPSJ, IEICE, and JSSST.