Master Thesis

Title

On Source Code Neutralization

– Integrating Preprocessing Methods for Static Code Analysis –

Supervisor
Prof. Shinji KUSUMOTO

by
Nozomi Nakajima

February 5, 2020

Departmant of Computer Science
Graduate School of Information Science and Technology
Osaka University

Master Thesis

On Source Code Neutralization
– Integrating Preprocessing Methods for Static Code Analysis –

Nozomi Nakajima

Abstract

Source code is a significant subject of research on software engineering, especially in source code analysis. The purpose of the source code analysis includes quantitative software quality assessment, prediction of source code problems, and reuse of existing software. Especially, the appearance of Mining Software Repositories (MSR) accelerates the momentum of source code analysis. MSR realizes various analyses combining source code and development history information. In any programming language, source code generally allows grammatical flexibility. The flexibility includes: selecting for-loop or while-loop, using ternary operator or if-statement, and whether or not to use break or continue statement. Coding style, which frequently sparks discussion between developers, arises from such flexibility. Although the flexibility gives various implement options for developers, it negatively affects source code analysis. When researchers conduct source code analysis, they generally perform preprocessing methods for eliminating syntactic differences before the analysis. Removing blank lines and code comments, formatting source code, replacing variables with specific tokens are major methods of source code preprocessing. Although source code preprocessing is a common task for many studies, these know-hows and tools have not been shared enough. In this study, we propose a concept of **Source Code Neutralization**. The purpose of neutralization is to avoid negative effects on source code analysis by transforming the given code into the normalized form. One of the important ideas is that neutralized source code behaves completely the same as the original. Also, the neutralized code keeps syntactic correctness, which means the code is always in compilable. The above restrictions enable fluent neutralization chains as a pipeline which has been leveraged in many software systems such as Unix and CI/CD (continuous integration and delivery). Under the concept of neutralization, we integrate preprocessing methods conducted in existing studies. Besides, we propose a tool named Neu4j as a prototype system. With Neu4j, researchers neutralize source code collectively and try various neutralizations freely. In order to indicate the effectiveness of neutralization and usefulness of Neu4j, we reproduced an existing experiment of cross-project defect prediction.

Keywords

Source Code Analysis
Code Preprocessing
Code Variety

Contents

List of Figures

List of Tables

1 Introduction

Source code is a significant subject of research on software engineering, especially in source code analysis [1] [2]. The purpose of the source code analysis includes quantitative software quality assessment [3], prediction of source code problems [4], and reuse of existing software [5]. Especially, the appearance of Mining Software Repositories (MSR) accelerates the momentum of source code analysis. MSR realizes various analyses combining source code and development history information.

Source code generally contains syntax differences arising from the flexibility of programming languages. The flexibility includes selecting for-loop or while-loop, using ternary operator or if-statement, and whether or not to use break or continue statement. Coding style, which frequently sparks discussion between developers, arises from such flexibilities. Although the flexibility gives various implementation options for developers, it negatively affects source code analysis. Source code analysis is not performed by humans so that understandability and readability are an obstacle. In order to eliminate such varieties and obtain more meaningful results, source code has been preprocessed depending on the analysis [6] [7]. Well-known and often used methods of source code preprocessing are removing blank lines and code comments, formatting source code, and replacing the variable name with anonymous one (e.g., `i` → `$1`).

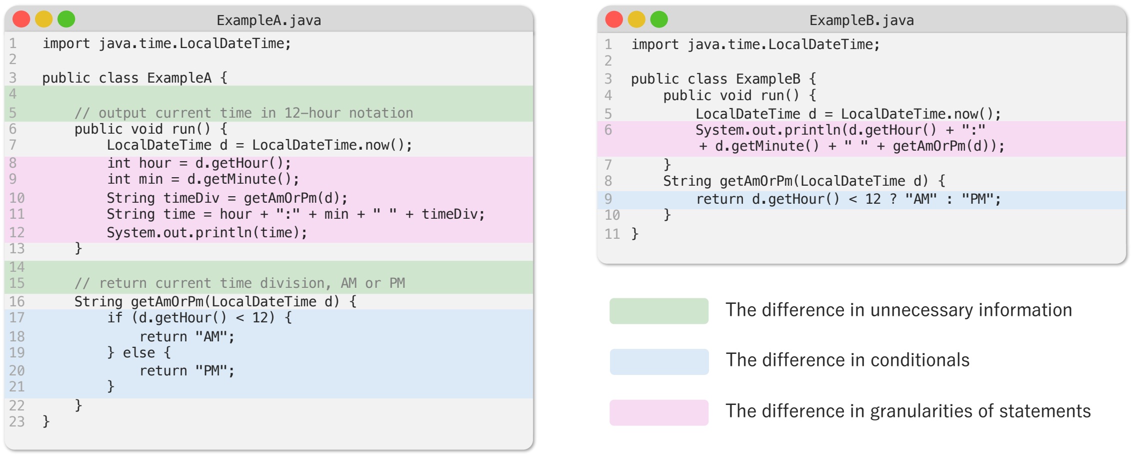
Preprocessing source code is also effectiveness when using various static code analysis tools. Lincke et al. compared several tools for software metrics measurement [8]. They pointed out that each tool takes a different policy, and the measurement results are different depending on tools. Even if measuring simple metrics such as logical LOC, metrics tools indicate different numbers. In such a situation, source code preprocessing is an effective way to avoid the problem. By preprocessing source code with a certain standard, researchers will do not have to care about such problems.

Although source code preprocessing is a common task for many studies, these know-hows and tools have not been shared enough. So far, researchers eliminate code varieties by using techniques such as program dependence graph (PDG) [9] [10] and abstract syntax tree (AST) [11]. However, researchers have conducted such preprocessing methods for a specific analysis. It is not clear that the preprocessed source code is reusable for another kind of analysis. Also, the result of preprocessing is not uniformed. When researchers develop a preprocessing tool, they will not develop the tool under the specific rule. Therefore, it is not easy to reuse preprocessing methods for another type of analysis.

In this research, we propose a concept of *Source Code Neutralization.* We define *Source Code Neutralization* as to transform given source code into normalized form by eliminating specific syntax flexibility. Neutralized code behaves the same as the original and keeps syntax correctness. Researchers can easily perform a series of neutralization methods like a pipeline. Under the concept of neutralization, we integrate source code preprocessing methods and perform them at one time. Besides, we propose a tool named Neu4j, which neutralizes source code. Neu4j take source file or source file directory and neutralize them. By providing Neu4j, we expect to eliminate the current incoherent on preprocessing and realize various combination and ordering of neutralization.

Besides, we reproduce an existing experiment of cross-project defect prediction with neutralized source code. The purpose of this experiment is to confirm that neutralization has some influence on static code analysis. As a result, it is indicated that neutralization certainly affects the result of the analysis.

The remainder of this thesis is organized as follows: in Chapter 2, we show motivating examples for this research. In Chapter 3, we explain our proposal concept of *Source Code Neutralization*. In Chapter 4, we present our proposal tool of *Neu4j*. In Chapter 5, we show the usage examples of our approach. In Chapter 6, we present our experiment and evaluation of our approach. In Chapter 7, we discuss the threats to validity of the evaluation. In Chapter 8, we provide related works for this research. Finally, in Chapter 9, we discuss the conclusions and future work.



```java
1   import java.time.LocalDateTime;
2
3   public class ExampleA {
4
5       // output current time in 12-hour notation
6       public void run() {
7           LocalDateTime d = LocalDateTime.now();
8           int hour = d.getHour();
9           int min = d.getMinute();
10          String timeDiv = getAmOrPm(d);
11          String time = hour + ":" + min + " " + timeDiv;
12          System.out.println(time);
13      }
14
15      // return current time division, AM or PM
16      String getAmOrPm(LocalDateTime d) {
17          if (d.getHour() < 12) {
18              return "AM";
19          } else {
20              return "PM";
21          }
22      }
23  }
```

```java
1   import java.time.LocalDateTime;
2
3   public class ExampleB {
4       public void run() {
5           LocalDateTime d = LocalDateTime.now();
6           System.out.println(d.getHour() + ":"
             + d.getMinute() + " " + getAmOrPm(d));
7       }
8       String getAmOrPm(LocalDateTime d) {
9           return d.getHour() < 12 ? "AM" : "PM";
10      }
11  }
```

🟩 The difference in unnecessary information

🟦 The difference in conditionals

🟪 The difference in granularities of statements

Figure 1: Motivating examples. Both source code have exactly the same functionalities but have some syntactic differences.

2 Motivating Examples

In this section, we introduce motivating examples of our work. ExampleA.java and ExampleB.java, shown in Figure 1, have the same functionalities. However, they do not have the same implementation.

2.1 The difference in unnecessary information

In Figure 1, yellowish-green highlights indicate comments and blank lines. Researchers often regard comments and blank lines on source code as unnecessary information. Though both lines give useful information for developers, they are unnecessary for analysis. Also, the insertion frequency and the number of such lines depend on the developer. For example, researchers often use logical lines of code (LLOC), which is the number of lines excluding comments and blank lines [12].

2.2 The difference in conditionals

Blue highlights shown in Figure 1 indicate the difference in conditionals. In any programming language, conditionals are a part of a significant basis. The use of such conditional notations strongly depends on each developer [10] [13]. For example, lines 17 to 21 in ExampleA.java are conditionals using if-else statement. In ExampleB.java, a ternary operator is used on line 9. Both ternary operator and if-else statement has the same function. Nevertheless, their numbers of statements are quite different. Not only that, switch case statement also have the same function. Such differences in conditionals generate inequality.

2.3 The difference in granularities of statements

In Figure 1, pink highlights show the difference in granularities of statements. Some developers prefer to use temporary variables for giving an explicit name, statements more straightforward, or easing debugging. Others prefer to stuff information into one program line. Such preference of developers occurs the difference in granularities of statements.

For example, ExampleA.java is described more finely than ExampleB.java. Lines 8 to 12 in ExampleA.java are the step to get the current time and output it. Some variables such as `hour` and `min` are used temporary. On the other hand, line 6 in ExampleB.java executes the above steps in one statement. There is no temporary variable for the output string in ExampleB.java, and this leads to reduce the count of line. In the case of source code analysis, It is desirable to eliminate such an imbalance of granularities. In order to adjust the granularity of source code, Higo et al. proposed code flattening [14]. Code flattening dissolves a complex program statement to multiple simple ones. It was indicated that code flattening contributes to obtaining more meaningful results of the analysis.

3 Neutralization

In this section, we define our proposal concept of *Source Code Neutralization* and show some examples. Hereafter, we call Source Code Neutralization as neutralization.

3.1 Definition

Our proposed concept, *Source Code Neutralization* is to transform a given source code to remove its varieties without changing external behavior. The variety of source code includes coding styles and other differences that arose from flexibilities on the programming language. The purpose of neutralization is to avoid the negative effect on source code analysis by removing these varieties. Since neutralization is assumed to be performed for source code analysis, understandability and readability are not considered. As a premise, the target of neutralization is compilable and valid source code. Before and after neutralization, source code behaves completely the same as the original.

Refactoring performs a similar process with our concept in terms of source code transformation. However, the purpose of refactoring is to improve maintainability, understandability, and other qualities for developers. Neutralization is conducted for source code analysis. Their purposes are definitively different.

3.2 Methods

Neutralization is conducted in various researches. Table 1 shows the neutralization methods conducted by existing researches.

Removing blank lines and comments from source code is an example of neutralization [7] [15]. Blank lines and comments are often removed to calculate LLOC. Reformatting source code is also conducted [16] [17]. The varieties that arose from coding-styles are unnecessary information for source code analysis.

Not only style neutralization, but there is also various syntax neutralization [10] [13] [14]. Code flattening [14] neutralizes the granularities of source code. The difference in selecting for-loop or while-loop, using a ternary operator or if-else statement, is also a target of neutralization [10] [13]. Such kind of neutralizations transforms source code structurally.

Table 1: Neutralization Methods

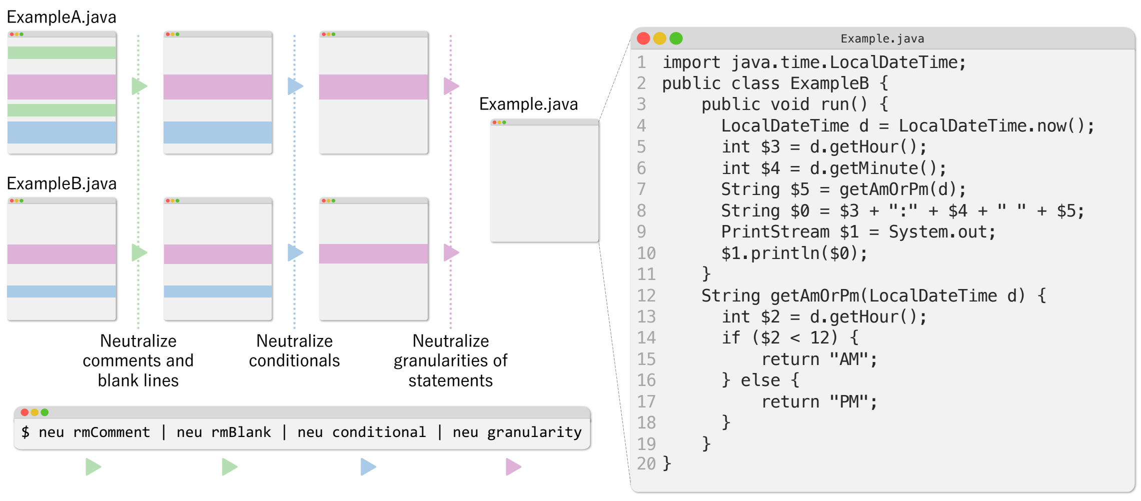
Name	Description	Example	Used on
Removing blank lines	All blank lines in source code will be removed.		metrics measurement [18] [19] [20] fault prediction [21] [22] traceability analysis [23]

Removing comments	All comments in source code will be removed.	`i = i + 1; // comment`
		→ `i = i + 1;`
			metrics measurement [18] [19] [20]
			fault prediction [21] [22]
			traceability analysis [23]
			clone detection [24]

Reformat	Coding style will be reformatted by specific standars		git refinement [25] clone detection [24]

Code flattening	A complex program statement will be dissolved into multiple simple ones.	`c = a.fn() + b.fn()`
		→ `$1 = a.fn(); $2 = b.fn();`
		`c = $1 + $2;`
			clone detection [14]
			metrics measurement [14]

Statement ordering	Statements will be reordered not to change the behavior of source code.	`int a; int b;` → `int b; int a;`	clone detection [26] comprehensibility [27] [28]
Tokenization	All tokens will be separated by newline character.	`import java . util . ...`	git refinement [29] natural language processing [30] clone detection [31]
Normalizing conditionals	All conditional statements such as if-else statement and ternary operator will be unified into one type.	`a = b ? c : d;` → `if(b) a = c; else a = d;`	clone detection [9] [10] comprehensibility [32]
Normalizing loops	All loop statements such as for-loop, while-loop, and enhanced for-loop will be unified into one type.	`for(i = 0; i < a.len; i++)` → `forEach(int num : a)`	clone detection [9] [10]
Implicit predicates	Implicit predicates will be specified clearly.	`if (5 % 3)` → `if (5 % 3 != 0)`	comprehensibility [32]
Abstracting identifiers	Identifiers will be replaced specific tokens to ingore the difference of identifier names.	`int count` → `int $1`	clone detection [33]
Expanding identifiers	Abbreviated identifiers will be expanded to the original word.	`execFunc` → `executeFunction`	natural language processing [34]


ExampleA.java
ExampleB.java
Neutralize comments and blank lines
Neutralize conditionals
Neutralize granularities of statements
Example.java

```
$ neu rmComment | neu rmBlank | neu conditional | neu granularity
```

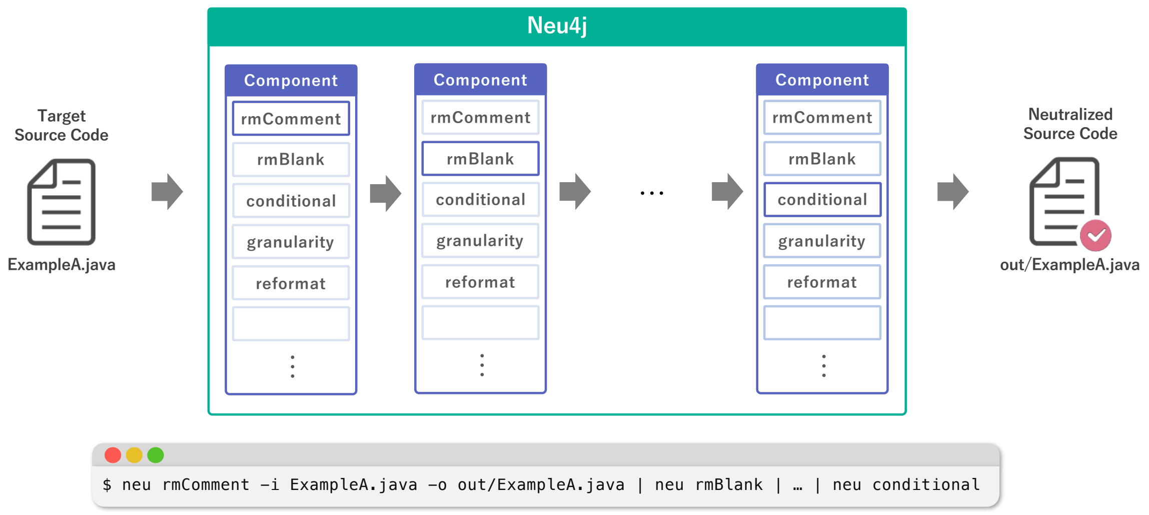
Example.java

```java
import java.time.LocalDateTime;
public class ExampleB {
    public void run() {
      LocalDateTime d = LocalDateTime.now();
      int $3 = d.getHour();
      int $4 = d.getMinute();
      String $5 = getAmOrPm(d);
      String $0 = $3 + ":" + $4 + " " + $5;
      PrintStream $1 = System.out;
      $1.println($0);
    }
    String getAmOrPm(LocalDateTime d) {
      int $2 = d.getHour();
      if ($2 < 12) {
          return "AM";
      } else {
          return "PM";
      }
    }
}
```

Figure 2: The flow of neutralization. ExampleA.java and ExampleB.java correspond to Figure 1. After the neutralization, both .java files have the same implementation shown in Example.java.

3.3 Flow of Neutralization

Figure 2 shows the flow of neutralization. In this figure, two source codes in Figure 1 are neutralized in roughly three steps. At first, comments and blank lines are removed. This step neutralizes the difference in unnecessary information. Second, conditionals are normalized to if-else statements. This step neutralizes the difference in conditionals. At last, the complex program statements are converted to multiple simple ones. This step neutralizes the difference in granularities of statements. After all the steps, ExampleA.java and ExampleB.java will have the same implementation, excluding variable names. It is indicated that neutralization can narrow the gap of source code.



```
$ neu rmComment -i ExampleA.java -o out/ExampleA.java | neu rmBlank | … | neu conditional
```

Figure 3: Neutralization Process with Neu4j

4 Neu4j

In this section, we introduce our proposal tool Neu4j for neutralization. Firstly, we introduce an overview of Neu4j. Then, we describe the current implementation of Neu4j.

4.1 Overview

As mentioned in the previous section, many researchers refer to neutralize source code or variety on source code. Moreover, some of them provide tools for neutralization. Nevertheless, such neutralizations have not been integrated yet. Existing neutralizations are existing independently. In order to integrate existing neutralizations, we propose Neu4j.

Neu4j[1] is a command-line tool which provides many neutralization methods for Java. Figure 3 shows a concrete example of neutralization process with Neu4j. At first, Neu4j takes the target source code as input using `-i` option. Neu4j transform target source code by selected preprocessing methods, which are called as components.

In Neu4j, neutralization is regarded as a combination of components. Actors select necessary components and put them in the order they want to try. This sequential processing is similar to pipelining such as Unix, Docker [35], and CI/CD [36]. Not only them, pipelining is a significant philosophy and employed in various fields of software engineering. The architecture of Neu4j imitates this philosophy of pipelining. As we showed in Section 2, source code is often neutralized continuously, but different tools conduct each process. By using Neu4j, researchers do not have to collect and perform tools independently. Also,

[1]Neu4j means abbreviation of *Neutralization for Java*.

researchers do not have to care about the difference between the interface of tools. After all preprocessing steps, Neu4j output neutralized source code.

4.2 Implementation

As of now, Neu4j provides six components of neutralization: rmComment, rmBlank, reformat, granularity, conditional, and tokenize.

removeCommentLines. This component removes unnecessary information of comments from source code. As an implementation, we use CommentRemover provided by Higo [37].

removeBlankLines. This component neutralizes unnecessary of blank lines from source code.

reformat. This component neutralizes the difference in coding style based on default settings of Eclipse. Currently, all parameters are set as default of Eclipse[2].

granularity. This component neutralizes the granularity of program statement. We introduced JCodeFlattener provided by Higo et al. [14] to Neu4j.

conditional. This component neutralizes the difference of conditional expressions. Current implementation convert ternary operators to if-else statements.

tokenize. This component neutralizes the granularity per one line. We introduced JavaTokenizer provided by Doi [38] to Neu4j.

4.3 Specification

As shown in Figure 3, researchers can use Neu4j by starting with `neu` command. The second argument of `neu` command is the component name to use for neutralization. Although Figure 3 shows the example which neutralizes a source code file, Neu4j can neutralize a specific source code directory. All source code files under the directory are neutralized keeping the directory structure.

[2]`https://github.com/eclipse/eclipse.jdt.core/blob/master/org.eclipse.jdt.core/formatter/org/eclipse/jdt/core/formatter/CodeFormatter.java`

5 Usage Example

In order to show the use case of Neu4j, we neutralize the differences of source code shown in Figure 1. For providing familiar examples, we assume two cases of neutralization for measuring different types of lines of code (LOC).

5.1 Measuring LLOC

Before measuring LLOC, we have to eliminate blank lines and comments from source code. As we mentioned in Section 1, we have to clear the definition of each metric on the tool when we use a metrics tool. The definition of the metrics depends on each tool. By neutralization of source code, we can avoid such efforts. When counting the lines of a file, we can use `wc` command with `--lines` option. Before the use of `wc` command, we have to neutralize source code. We can eliminate blank lines and comments by the following command.

```
$ neu rmComment -i ExampleA.java \
  -o out/ExampleA.java | neu rmBlank
```

The processes eliminating comments and blank lines are combined by pipeline. Though file-path is inherited to the next component, Neu4j does not require to set input/output options every component. After this neutralization process, we can count the LLOC by the following command.

```
$ wc -l out/ExampleA.java
```

In addition, Neu4j can take a source file directory as input. When neutralizing source files under `src` directory, the following command is executable.

```
$ neu rmComment -i src/ -o out/ \
| neu rmBlank
```

By specification of a directory path, researchers do not have to neutralize each source file independently. Besides, it handles differences between component interfaces.

5.2 Measuring fine-grained LOC

In this section, we define fine-grained LOC as the LLOC of a neutralized source file. For example, as shown in Figure 1, the granularity per one program statement depends on each developer. By neutralization of granularities, we can expect to obtain more fine-grained results. Before the measurement of fine-grained LOC, we have to remove unnecessary information and neutralize the granularity of source code. We can neutralize source code for fine-grained LOC with the following command.

```
$ neu rmComment -i ExampleA.java \
  -o out/ExampleA.java \
| neu rmBlank | neu granularity
```

Only adding the command of `neu granularity` to the command shown in Section 5.1, we can neutralize source code as we would like.

Moreover, we can easily add and reorder components. Directory specification is also available.

```
$ neu rmComment -i src/ -o out/ \
| neu rmBlank | neu conditional \
| neu granularity \
```

Especially when using preprocessing tools provided by researchers or developers, researchers need to care about the difference in the interface such as input/output specification and option name. In order to avoid such efforts, Neu4j integrates preprocessing tools and behaves as a wrapper of the tools. Furthermore, Neu4j always outputs a compilable source code. This premise enables fluent neutralization chains as a pipeline.

6 Experiment

6.1 Overview

In this section, we explain our experiment to confirm the influence of neutralization on static code analysis. We reproduced a cross-project defect prediction conducted by Zimmermann et al [39]. However, our experiment did not completely reproduce the experiment for some reason. First, the dataset used in the experiment is not available. We use another bug dataset alternatively. Besides, some metrics proposed by them were unclear, and we could not extract the metrics.

In general, the purpose of defect prediction is to predict the number of bugs included in the source file by statistical approaches including supervised machine learning. Some software metrics were extracted from the source files and historical information of the development. The extracted metrics were used as explanatory variables of machine learning, and the number of bugs used as an objective variable. Ideally, prediction model is trained with the data of the project to predict. However, training data is often not available or insufficient when the project is in the step of first release or developing company is too small [39]. In the case of cross-project defect prediction, the model is constructed by the metrics extracted from other software development projects. Therefore, the result of defect prediction may be influenced by the varieties on the project used for learning. We considered that the result of defect prediction would be changed by neutralization.

We briefly describe the flow of this experiment. At first, we retrieved source files of the projects by checking out their Git repositories. Then, all source files were neutralized by a variety of combinations. After that, we calculated the metrics of source code. The defect prediction was conducted with the metrics, and we compared the results of prediction.

6.2 Experimental Methodologies

6.2.1 Dataset

The dataset used in the experiment by Zimmermann et al. [39] is not available. Instead, we used Bug Database of GitHub Projects published by Tóth et al [40] for bug prediction. Table 2 shows a list of projects included in Bug Database of GitHub Projects. In this dataset, there is information about fifteen projects. However, this dataset was published four years ago, and some of them lack the data for defect prediction. For instance, we could not find the hash of Oryx from its Git repository. One possible reason is that Oryx project had applied re-construction for their Git repository. Moreover, we wanted to use the version with most bugs and the previous one. However, Android Universal I. L. and Mission Control T. did not have the previous data. They did not satisfy the conditions, so we removed them from the target. As a result, we use twelve projects for the next step.

6.2.2 Source Files

In the bug database, all source files are assigned a specific id which can uniquely identify the file. This id can be used to map an actual source file to bug information. In order to map source files to bug information, we stored all committed source files in all projects and assigned a unique name to them. Figure 4 shows the naming rule of source files. All files were named based on the following three keys.

id: This key can be used to map source files to bug information.

order: This key means the committed order of each file. The starting point of the order is the latest commit in the version. The order is reversed from the time axis.

commit hash: This key is the first six characters of commit hash.

With these three keys, all source files are named. All files were stored in the directory for each project in serial.

Table 3 shows the total number of stored source files for each project. The numbers of source files heavily depended on the projects. As a premise, the purpose of this experiment is to confirm the influence of neutralization. Therefore, we need to eliminate data imbalance for learning. In order to eliminate the imbalance, we limited the number of stored source files from 1000 to 5000. Finally, we selected six projects shown in Table 3.

Table 2: Projects in Public Bug Database

Project name	Date of latest commit in database	Hash (version with most bugs)
Android Universal I. L.	2013/1/19	48d5c652
ANTLR v4	2014/2/3	5e05b71e
BroadleafCommerce	2014/3/5	72255ca6
Eclipse p. for Ceylon	2014/10/8	29c9597b
Elasticsearch	2014/2/3	bd8cb4eb
Hazelcast	2014/5/9	139f7eb8
jUnit	2011/8/22	61f06547
MapDB	2013/9/27	f6d1d916
mcMMO	2013/7/11	4a5307f4
Mission Control T.	2012/6/29	0cc9d801
Neo4j	2014/4/9	04576eb6
Netty	2013/2/21	b644d4e9
OrientDB	2013/12/10	bbb45db9
Oryx	2013/11/10	291ba768
Titan	2014/10/9	495402f9

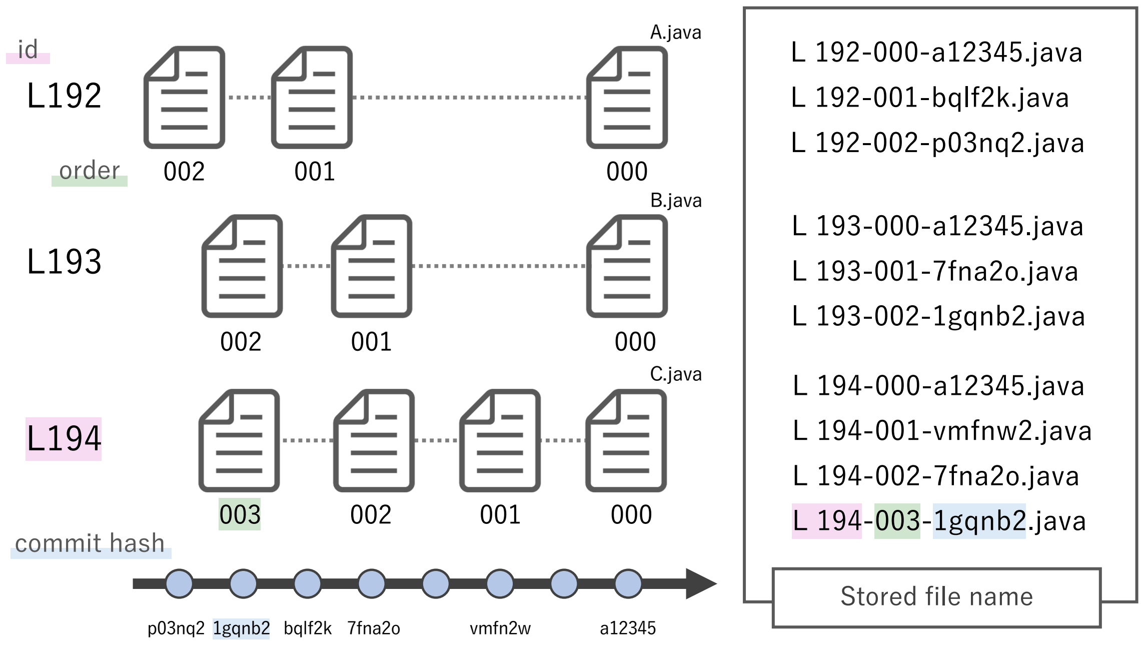


Figure 4: Naming rule of source files

Table 3: Total number of stored source files

Project Name	Total stored files	Selected for prediction
ANTLR v4	771	
BroadleafCommerce	1,895	○
Eclipse p. for Ceylon	4,596	○
Elasticsearch	11,203	
Hazelcast	8,285	
jUnit	472	
MapDB	745	
mcMMO	2,351	○
Neo4j	12,901	
Netty	3,721	○
OrientDB	3,799	○
Titan	3,898	○

6.2.3 Source Code Metrics

This experiment is based on the experiment conducted by Zimmermann et al [39]. We calculated the following four metrics which are used in their experiment.

Added lines/Total LOC: This metric means the ratio of added lines to total LOC. The reason to use relative metrics is that existing studies have shown that absolute metrics are poor in predicting pre- and post-release defects. Using normalized metrics is highly effective to qualify the change in a system. This metric was calculated by the number of added lines between the version with most bugs and the previous one.

Deleted lines/Total LOC: This metric means the ratio of deleted lines to total LOC. This metric was calculated by the number of deleted lines between the version with most bugs and the previous one.

Cyclomatic Complexity/Total LOC: This metric means the cyclomatic complexity per total LOC. This metric was calculated by cyclomatic complexity of source file in the version with most bugs. We use PMD [41] to measure cyclomatic complexity.

(Added lines + Deleted lines)/(commits + 1): This metric means the number of modified lines between each commit. Adding the number of added lines and deleted lines is equal to the number of modified lines. The number of modified lines was divided by the number of commits plus one.

6.2.4 Neutralization

In the experiment by Zimmermann et al. [39], source code was used without any preprocessing. In our experiment, we neutralized source code by Neu4j and used it for prediction.

As shown in the previous section, we used the metrics which heavily depend on the LOC in this experiment. Therefore, we selected four components: **removeBlankLines**, **removeCommentLines**, **granularity**, and **tokenize**. Each component is abbreviated as B, C, G, and T, respectively. We conducted nine combinations of neutralization: B, C, G, T, BC, BG, CG, GT, and BCG. Since the component of tokenize includes rmBlank and rmComment, we did not combine tokenize and rmBlank or rmComment. Besides, these combinations output the same neutralized source code regardless of the order of components.

6.2.5 Prediction

We performed machine learning for prediction. As a learning algorithm, we selected Random Forest [42]. Random Forest was indicated its high performance in Bug Database of GitHub Projects [40].

Though it is general to conduct parameter tuning in machine learning, we used default parameters. Our purpose was not to improve the performance of the model but to confirm the influence by neutralization on analysis. We constructed six learning models. These models learned data of five projects and predicted the remaining one project.

As an evaluation index, we employed Area Under the Curve (AUC) [43] of Receiver Operating Characteristic Curve (ROC) [44]. Also, we analyzed the change of each metric by neutralization. In this experiment, we calculated AUC of ROC with the following steps.

1. Data Sorting

   Before the calculation of AUC, we preprocessed the prediction data. At first, all data were sorted in descending order based on the predicted number of bugs. Next, data were also sorted in descending order based on the actual number of bugs not to change the order of the predicted number of bugs.

2. Plotting and Calculation

   In order to calculate the AUC, we plotted ROC. For normalization, we plotted three cases of ROC: prediction, best, and worst. ROC in the case of prediction is the plot of the integrated value of predict number of bugs. ROC in the best case is plotted based on the integrated value of the actual number of bugs in order of Step 1. ROC in the worst case is the plot of the integrated value of the actual number of bugs in the reversed order of Step 1.

3. Normalization

   Based on three cases of ROC, we calculated AUC. AUC will be maximum in case of best and be minimum in case of worst.

   In this experiment, it was required to compare the results of the six projects. So, we normalized ROC to the range of 0 to 1. When the case X of AUC is described as AUC(X), normalized AUC in the case of prediction is calculated with the following formula.

$$\frac{AUC(prediction) - AUC(min)}{AUC(max) - AUC(min)}$$

6.3 Results

6.3.1 AUC

Figure 5 shows the prediction result with neutralized source code. The vertical axis shows the value of AUC. The horizontal axis shows project names used for prediction. AUC significantly increased in the prediction of ceylon-ide-eclipse and mcMMO. Especially, AUC of mcMMO was improved about 0.2 points in case of neutralization by GT. AUC tends to improve when the value was low without neutralization. Though AUC of orientdb and titan did not change, AUC of BroadleafCommerce and netty decreased. AUC also tends to get worse when the value was high without neutralization.

6.3.2 Added lines/Total LOC

Figure 6 shows the change of Added lines/Total LOC metric. The vertical axis shows the change ratio compared to the metric calculated with source code without neutralization. When the value in the vertical axis is one, it is the same as in the case without neutralization. Overall, the change tendency is similar to each other. However, the change ratio of BroadleafCommerce is greatly different from other projects. The change ratio of orientdb is also different from others. Also, when introducing the component of G, the ratio tends to become high. As shown in Figure 6a, one-time neutralization did not change the metric so much excluding the component of G.

6.3.3 Deleted lines/Total LOC

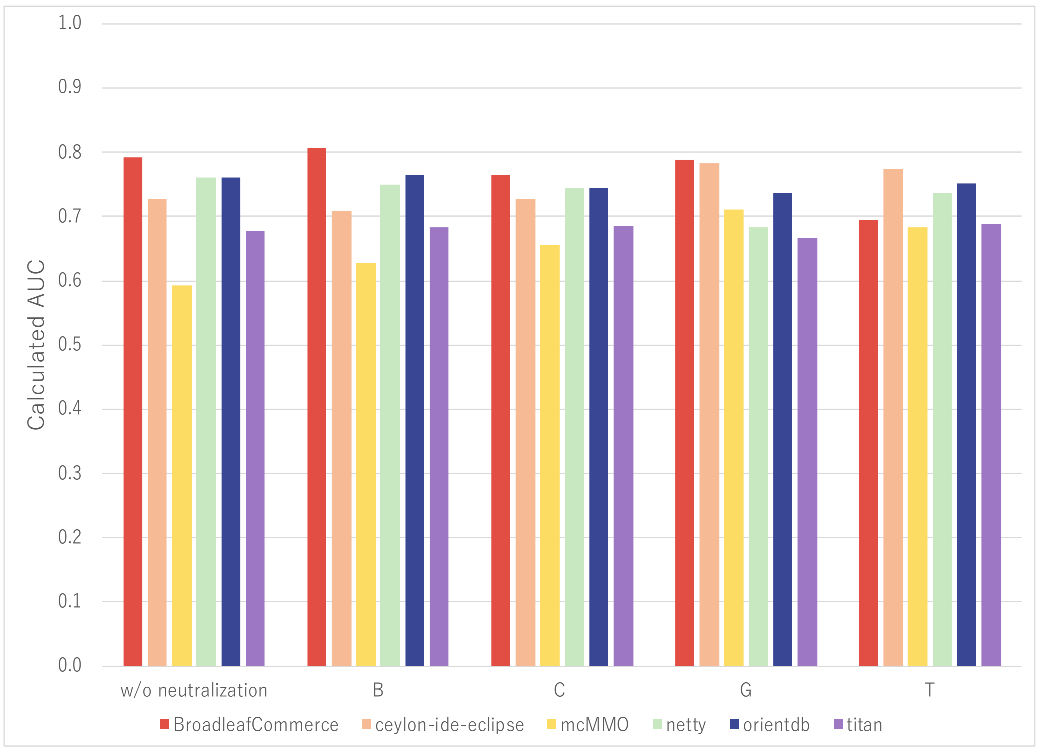
Figure 7 shows the change of Deleted lines/Total LOC metric. Although the change tendency is similar to Figure 6, the change rate is a little lower. In the case of this metric, the change ratio of BroadleafCommerce and orientdb is different from others. This tendency is also similar to the case of Figure 6. As shown in Figure 7a, one-time neutralization did not change the metric so much excluding the component of G.

6.3.4 Cyclomatic Complexity/Total LOC

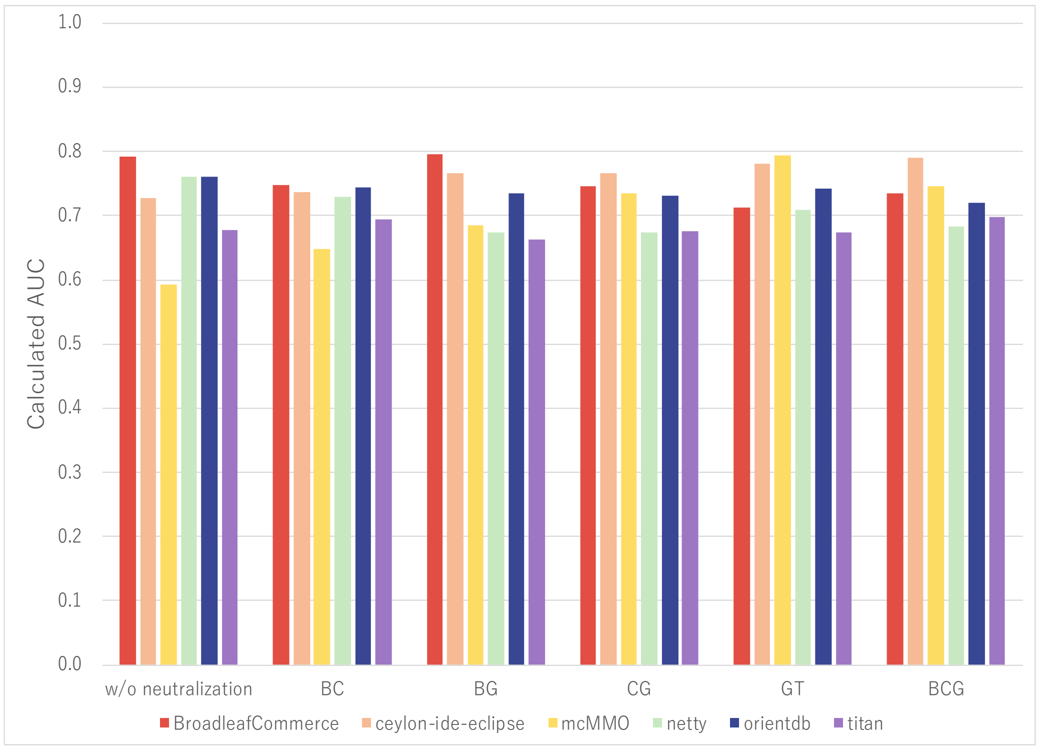
Figure 8 shows the change of Cyclomatic Complexity/Total LOC metric. Compared to previous results, this metric was not changed so much. Besides, this metric greatly decreased with neutralization by T and GT. Neutralization by tokenize increased the number of total LOC, this may strongly affect the metric. Also, the metric extracted from netty tends to be affected more strongly than other projects.

6.3.5 (Added lines + Deleted lines)/(commits + 1)

Figure 9 shows the change of (Added lines + Deleted lines)/(commits + 1) metric. Unlike the other figures, the vertical axis is shown as a logarithmic. This metric became lower with neutralization by B, C, and BC. Before the neutralization, such unnecessary manipulation was also included in the development history. Neutralization by the combination with G and T made the metric higher. This metric greatly changes in the case of BroadleafCommerce. Especially, the metric was over 160 times greater in the case of GT than without neutralization.

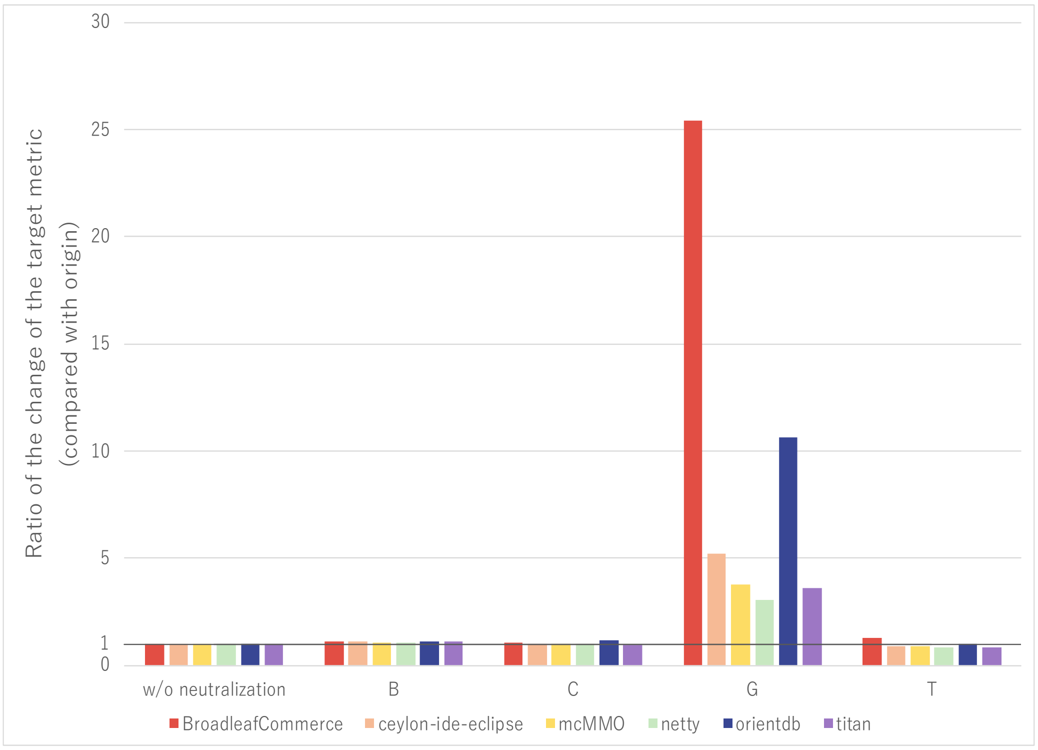


(a) Prediction with one-time neutralization

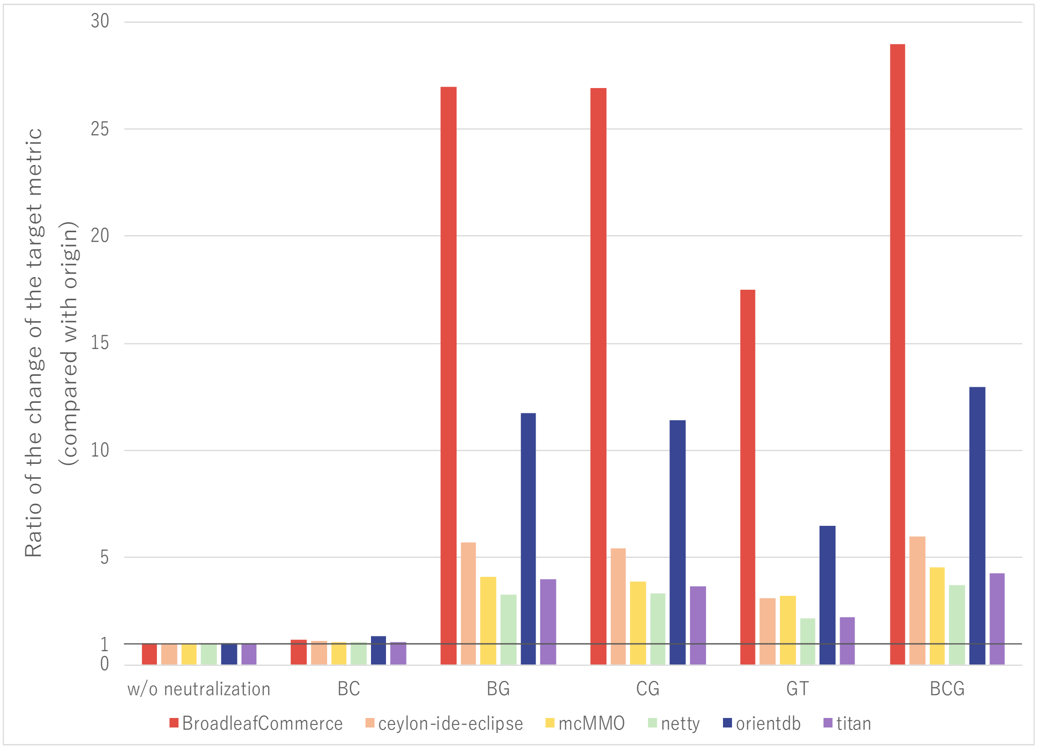


(b) Prediction with combined neutralization

Figure 5: Prediction result with neutralized source code

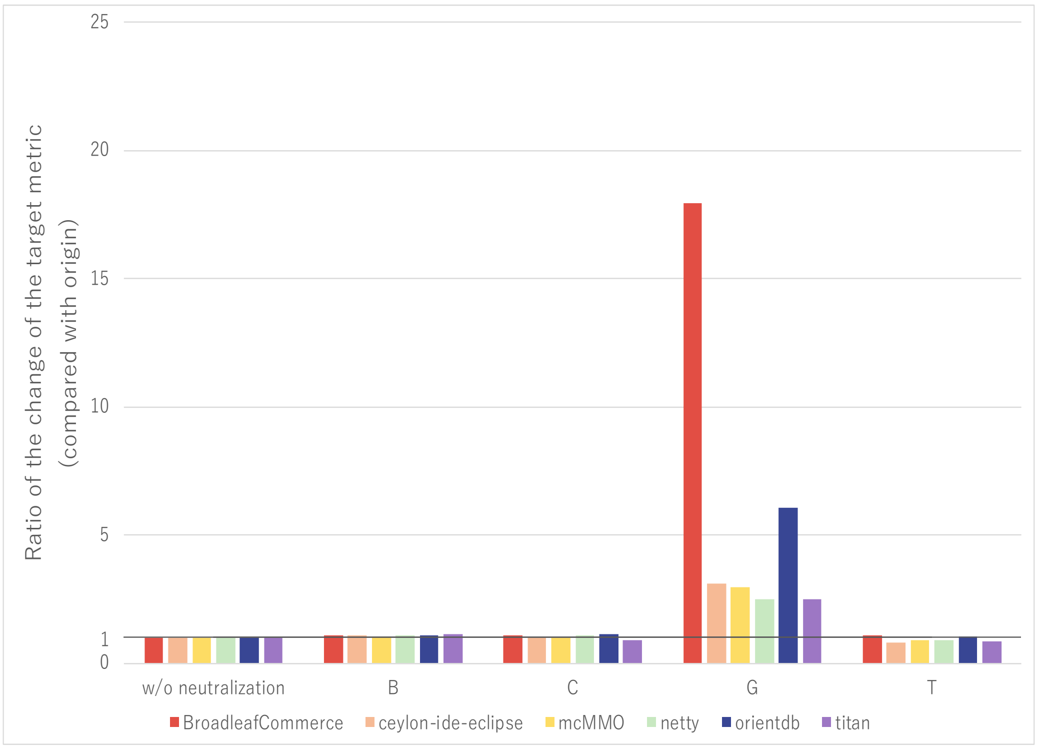


(a) Change ratio by one-time neutralization

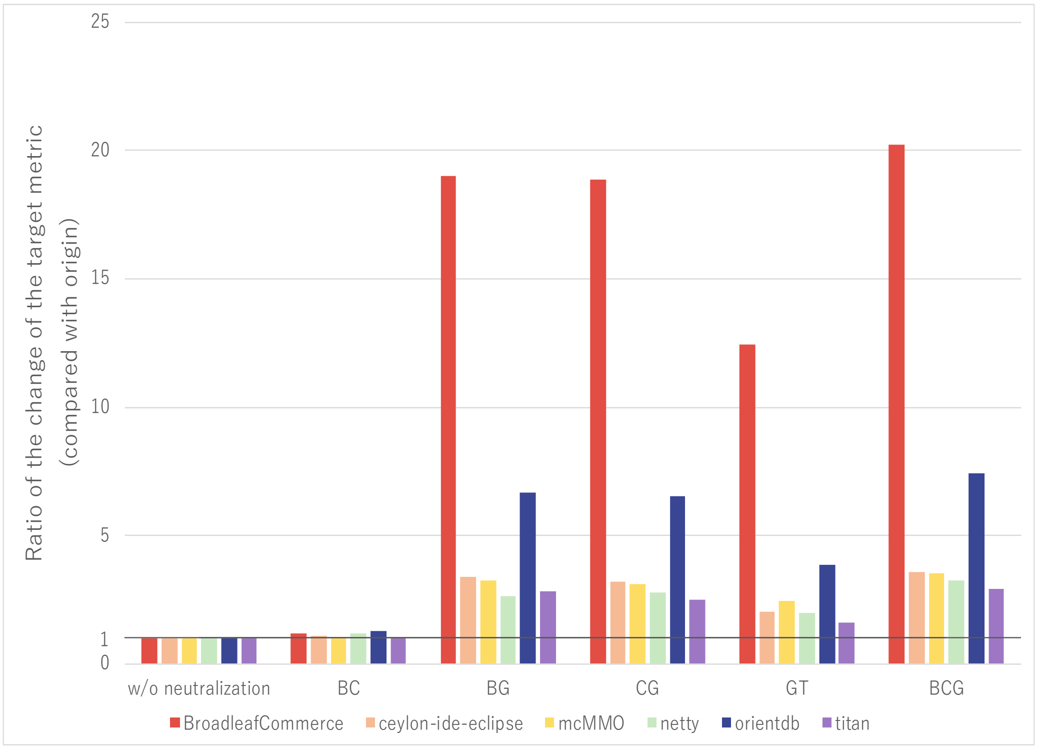


(b) Change ratio by combined neutralization

Figure 6: Change ratio of Added lines/Total LOC metric

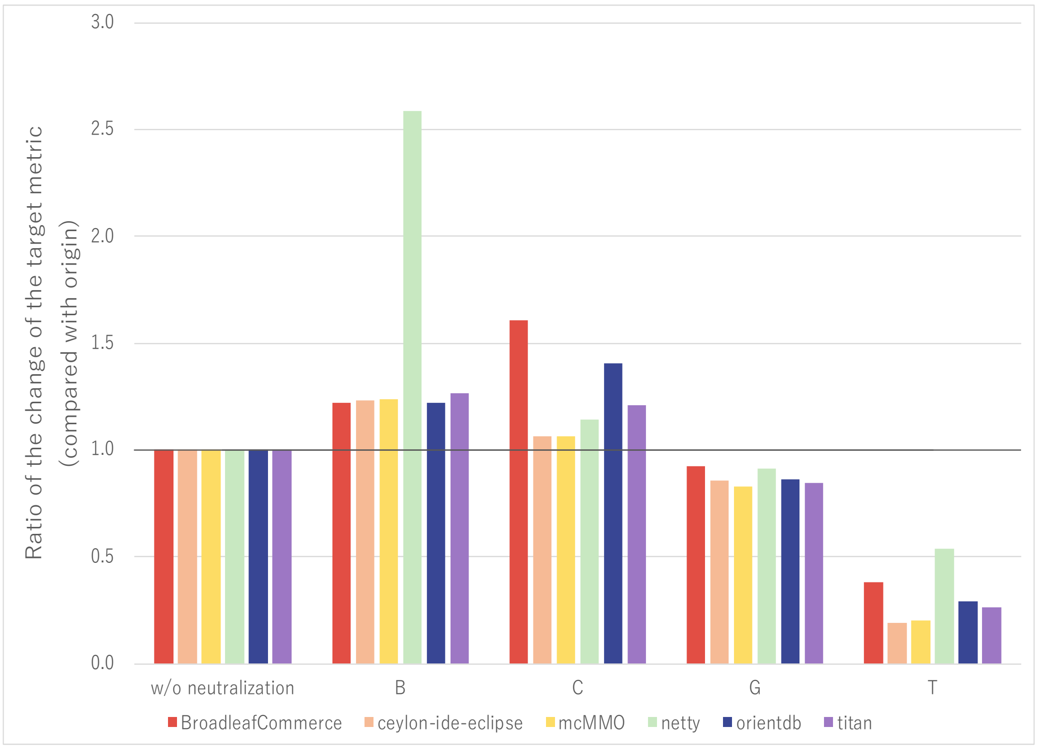


(a) Change ratio by one-time neutralization

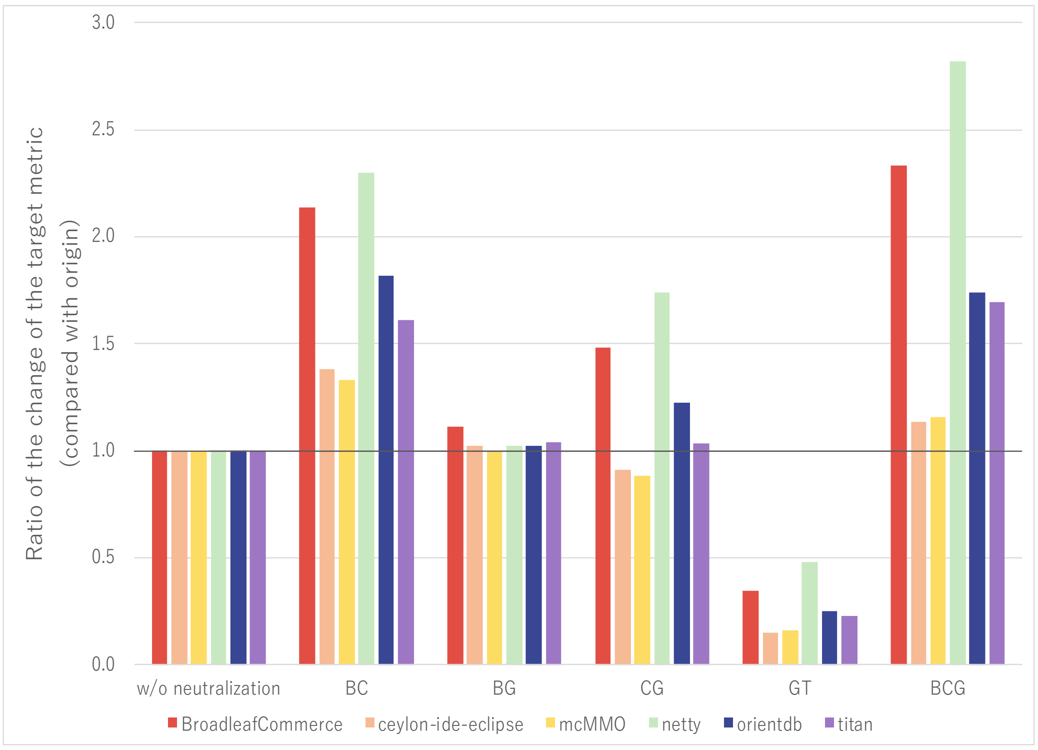


(b) Change ratio by combined neutralization

Figure 7: Change ratio of Deleted lines/Total LOC metric

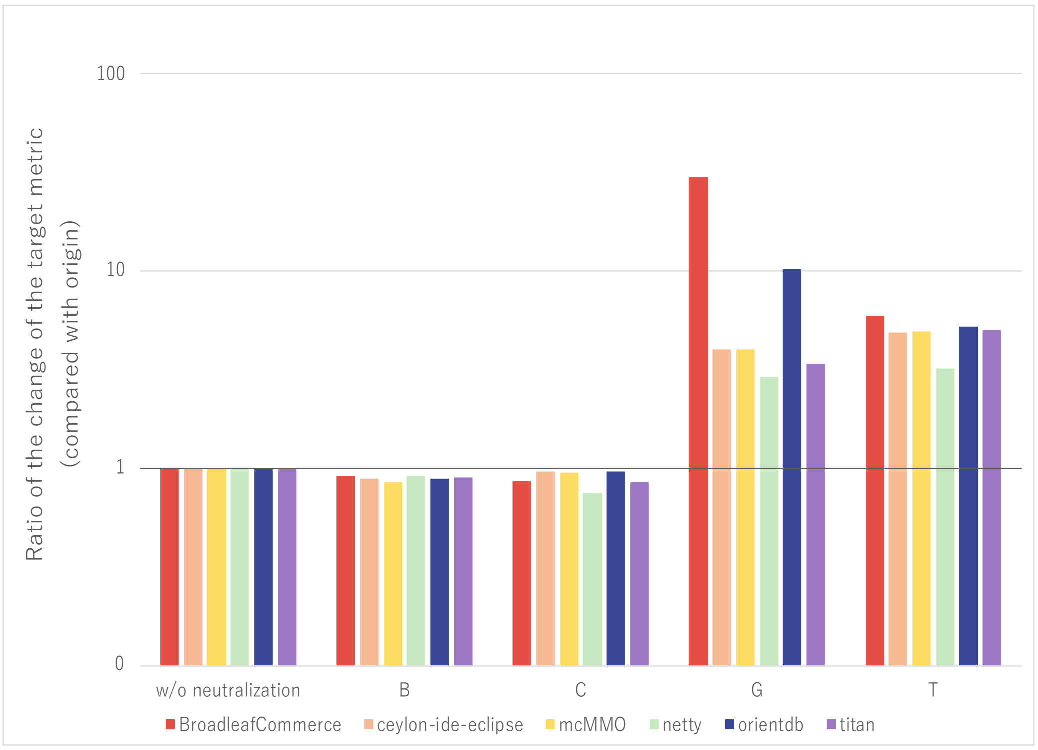


(a) Change ratio by one-time neutralization

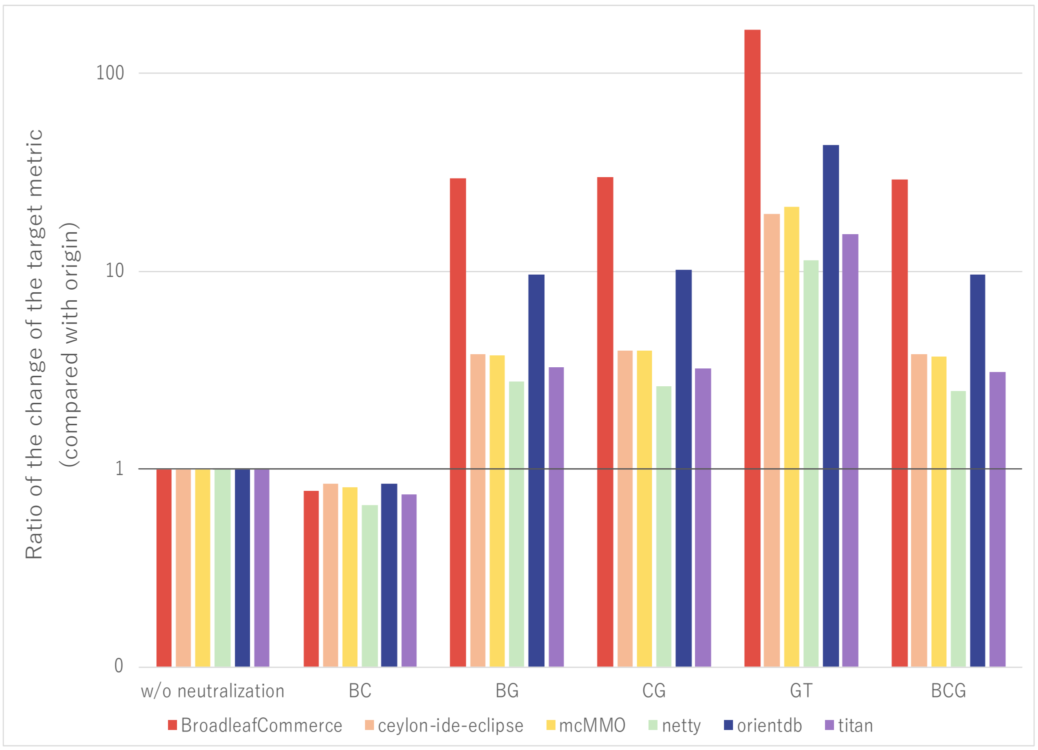


(b) Change ratio by combined neutralization

Figure 8: Change ratio of cyclomatic complexity/Total LOC metric



(a) Change ratio by one-time neutralization



(b) Change ratio by combined neutralization

Figure 9: Change ratio of (Added lines + Deleted lines)/(commits + 1) metric

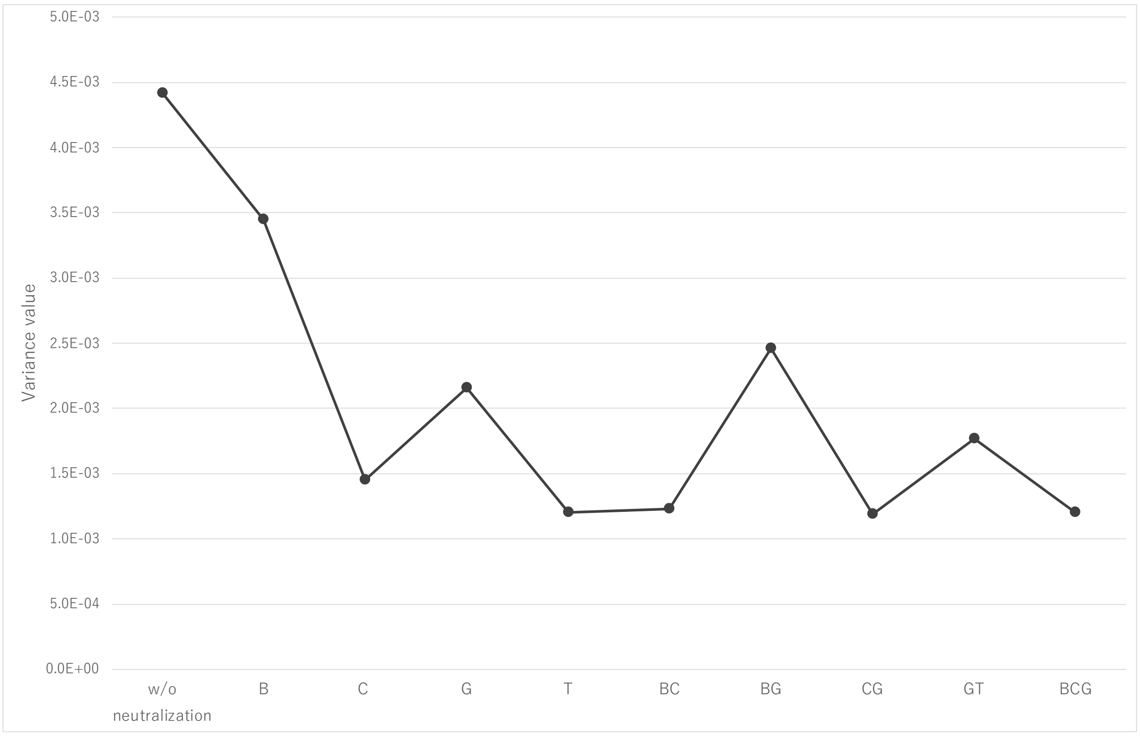


Figure 10: Variance values of AUC

6.4 Discussion

We discuss the result of this experiment from two viewpoints.

At first, we discuss the result from the viewpoint of cross-project defect prediction. Figure 10 shows the variance values of AUC. The vertical axis shows the variance values of AUC in the case of each neutralization. As shown in the figure, the variance values of AUC is decreased with neutralization. Especially, combining some methods of neutralization is effective for reducing the variance values. This tendency means that neutralization neutralizes the result of prediction. Though AUC became worse in some cases shown in Figure 5, it is indicated that learning models got more versatile with neutralization. Even if the target project for prediction is characteristic, the learning model can keep the constant prediction accuracy. For constructing a versatile prediction model, learning with neutralized source code is extremely effective.

Next, we describe the usefulness of Neu4j on static code analysis. In our experiment, we could employ the neutralization methods proposed by other researchers. This is because Neu4j imports existing neutralization methods to itself. This feature can solve the problem that existing neutralization techniques are inapplicable to other analysis in most cases. Besides, we could easily try various combinations of neutralization methods in our experiment. For example, components of granularity and tokenize are provided by other developers, and their interfaces are not the same. Nevertheless, we did not have to care about such differences and could conduct frequent neutralization. Neu4j enables to chain some neutralization methods like pipelining, so researchers do not have to care about such differences. It is indicated that this feature contributes to examine which neutralization is effective in the analysis. From the above reasons, researchers can easily introduce various neutralization with Neu4j.

7 Threats to Validity

In the evaluation, there are some factors which may affect the validity of the results. First, our experiment could not completely reproduce the experiment by Zimmermann et al [39]. There are mainly two reasons for this.

When we measure metrics, we could not clarify the definition of some metrics. Therefore, two metrics used in the original experiment are unused in our experiment. The lack of metrics may influence the validity of the prediction result. Also, the dataset is quite different from the original experiment. If used metrics are not effective on the dataset, they also influence on learning and prediction.

Besides, we did not use all projects in the dataset. This is because unused projects did not meet the conditions of our experiment. In the case of prediction based on all projects, the result may be different from ours.

8 Related Works

Source code analysis has been conducted for long decades [1] [2]. The analytical method and purpose of the analysis are diverse. The measurement of software metrics is one of the significant analysis methods. Software metrics are used in various fields, such as defect prediction [4] and quality measurement [45]. Metrics enable quantitative software evaluation [16]. Code clone is also significant on source code analysis [6]. They contribute to software reuse [5] and refactoring [13]. Recently, source code analysis is conducted with various techniques such as natural language processing [46] and deep learning [47].

So far, some researchers proposed neutralization methods for source code analysis. Higo et al. propose code flattening, which transforms a complex program into simple one [14]. Code flattening neutralizes the granularity of the statement per one line. Qiao pointed out that there are various implementations for the same scenario [10]. For example, loop implementation has several types: for-loop, enhanced for-loop, and while-loop. Using a ternary operator or if-else statement is another example of various implementations. Since such varieties directly affect source code analysis, they are desirable to eliminate. In Neu4j, we employ such effective neutralization methods. By using Neu4j, researchers can try various combinations of neutralizations.

9 Conclusion

In this paper, we introduced the concept of *Source Code Neutralization*. The purpose of neutralization is to avoid negative effects on source code analysis by transforming the given code into the normalized form. Also, we proposed a tool named Neu4j, which neutralizes source code. Neu4j enables neutralization chains like a pipeline which has been leveraged in many software systems such as Unix and CI/CD. In order to confirm the influence by neutralization on static code analysis, we conducted reproduce experiment.

There are mainly two future works. Currently, Neu4j is just a prototype and has only four components of neutralization. Also, the three components neutralize from the viewpoint of source code format. Therefore, current Neu4j do not change the structure of the source code drastically. The components which change the structure of source code affect the result of analysis more directly. Not only that, reordering and combining such components may generate other formats of neutralized source code. In order to expand the possibilities of Neu4j, we are planning to introduce more components to Neu4j. Also, Neu4j has not been evaluated empirically. We need to evaluate and show the effectiveness of Neu4j.

Acknowledgements

During this work, I have been fortunate to have received assistance from many people. This work could not have been possible without their valuable contributions. First, I wish to express my deepest gratitude to my supervisor Professor Shinji Kusumoto at Osaka University, for his continuous support, encouragement and guidance of the work. I also thank him for providing me an opportunity to do this work.

I am also deeply grateful to Yoshiki Higo, Associate Professor at Osaka University for his encouragement, practical advice, valuable suggestions, helpful comments and discussions throughout this work.

I would like to express my sincere gratitude to Shinsuke Matsumoto, Assistant Professor at Osaka University for his adequate guidance, valuable suggestions and discussions throughout this work. Without his supports, this work would never have reached completion.

I thank greatly, Hiroto Tanaka, Masayuki Doi, and Junnosuke Matsumoto, a colleague in Kusumoto Laboratory for their constant encouragements and valuable advices.

Finally, I would like to thank all of my friends in the Department of Computer Science at Osaka University, especially the members in Kusumoto Laboratory, for their helpful advices, suggestions and assistance.

References

[1] David Binkley. Source Code Analysis: A Road Map. In *Future of Software Engineering*, pp. 104–119. IEEE Computer Society, 2007.

[2] Mark Harman. Why Source Code Analysis and Manipulation Will Always Be Important. In *Proceedings of the 10th IEEE Working Conference on Source Code Analysis and Manipulation*, pp. 7–19. IEEE Computer Society, 2010.

[3] Akito Monden, Daikai Nakae, Toshihiro Kamiya, Shin-ichi Sato, and Ken-ichi Matsumoto. Software Quality Analysis by Code Clones in Industrial Legacy Software. In *Proceedings of the 8th International Symposium on Software Metrics*, pp. 87–94. IEEE Computer Society, 2002.

[4] Tibor Gyimothy, Rudolf Ferenc, and Istvan Siket. Empirical Validation of Object-Oriented Metrics on Open Source Software for Fault Prediction. *IEEE Trans. Softw. Eng.*, Vol. 31, No. 10, pp. 897–910, 2005.

[5] Naohiro Kawamitsu, Takashi Ishio, Tetsuya Kanda, Raula Gaikovina Kula, Coen De Roover, and Katsuro Inoue. Identifying Source Code Reuse across Repositories Using LCS-Based Source Code Similarity. In *Proceedings of IEEE 14th International Working Conference on Source Code Analysis and Manipulation*, pp. 305–314, 2014.

[6] Toshihiro Kamiya, Shinji Kusumoto, and Katsuro Inoue. CCFinder: A Multilinguistic Token-Based Code Clone Detection System for Large Scale Source Code. *IEEE Transactions on Software Engineering*, Vol. 28, No. 7, pp. 654–670, 2002.

[7] Brenda Sue. Baker. On Finding Duplication and Near-duplication in Large Software Systems. In *Proceedings of the Second Working Conference on Reverse Engineering*, pp. 86–. IEEE Computer Society, 1995.

[8] Rüdiger Lincke, Jonas Lundberg, and Welf Löwe. Comparing Software Metrics Tools. In *Proceedings of the International Symposium on Software Testing and Analysis*, pp. 131–142, 2008.

[9] Tiantian Wang, Kechao Wang, Xiaohong Su, and Peijun Ma. Detection of semantically similar code. *Frontiers of Computer Science*, Vol. 8, pp. 996–1011, 12 2014.

[10] Guo, Qiao. Mining and Analysis of Control Structure Variant Clones. Master thesis, Concordia University, 2015.

[11] Peter Bulychev and Marius Minea. Duplicate Code Detection Using Anti-Unification. 2008.

[12] Reisha Humaira, Kazunori Sakamoto, Akira Ohashi, Hironori Washizaki, and Yoshiaki Fukazawa. Towards a Unified Source Code Measurement Framework Supporting Multiple Programming Languages. In *Proceedings of the 24th International Conference on Software Engineering & Knowledge Engineering*, pp. 480–485, 2012.

[13] Davood Mazinanian, Nikolaos Tsantalis, Raphael Stein, and Zackary Valenta. JDeodorant: Clone Refactoring. In *Proceedings of the 38th International Conference on Software Engineering Companion*, pp. 613–616. ACM, 2016.

[14] Yoshiki Higo and Shinji Kusumoto. Flattening Code for Metrics Measurement and Analysis. In *Proceedings of IEEE International Conference on Software Maintenance and Evolution*, pp. 494–498, 2017.

[15] James A. Jones and Mary Jean Harrold. Empirical Evaluation of the Tarantula Automatic Fault-localization Technique. In *Proceedings of the 20th IEEE/ACM International Conference on Automated Software Engineering*, pp. 273–282. ACM, 2005.

[16] Stéphane Ducasse, Oscar Nierstrasz, and Matthias Rieger. On the Effectiveness of Clone Detection by String Matching: Research Articles. *Journal of Software Maintenance and Evolution*, Vol. 18, No. 1, pp. 37–58, 2006.

[17] Yusuke Sasaki, Tetsuo Yamamoto, Yasuhiro Hayase, and Katsuro Inoue. Finding file clones in FreeBSD Ports Collection. In *Proceedings 7th IEEE Working Conference on Mining Software Repositories*, pp. 102–105, 2010.

[18] Marcos Dósea, Cláudio Sant'Anna, and Bruno C. da Silva. How do Design Decisions Affect the Distribution of Software Metrics? In *Proceedings of the 26th International Conference on Program Comprehension*, pp. 74–85, 04 2018.

[19] Greg Wilson Andy Oram. *Making Software What Really Works, and Why We Believe It.* O'Reilly Media, 10 2010.

[20] Samuel. Daniel. Conte, Hubert. Earl. Dunsmore, and Vincent. Shen. Benjamin-Cummings Publishing Co., Inc., USA, 1986.

[21] Yue Jiang, Bojan Cuki, Tim Menzies, and Nick Bartlow. Comparing Design and Code Metrics for Software Quality Prediction. In *Proceedings of the 4th International Workshop on Predictor Models in Software Engineering*, 2008.

[22] Chris Lewis, Zhongpeng Lin, Caitlin Sadowski, Xiaoyan Zhu, Rong Ou, and E. Whitehead. Does Bug Prediction Support Human Developers? Findings From a Google Case Study. In *Proceedings of the 2013 International Conference on Software Engineering*, pp. 372–381, 05 2013.

[23] Marc Eaddy, Alfred V. Aho, Giuliano Antoniol, and Yann-Gaël Guéhéneuc. CERBERUS: Tracing Requirements to Source Code Using Information Retrieval, Dynamic Analysis, and Program Analysis. pp. 53–62, 07 2008.

[24] Jens Krinke. A Study of Consistent and Inconsistent Changes to Code Clones. In *Proceedings of the 14th Working Conference on Reverse Engineering*, pp. 170–178, Oct 2007.

[25] Naoto Ogura, Shinsuke Matsumoto, Hideaki Hata, and Shinji Kusumoto. Bring Your Own Coding Style. In *Proceedings of the IEEE 25th International Conference on Software Analysis, Evolution and Reengineering*, Vol. 1, pp. 527–531, March 2018.

[26] Yusuke Sabi, Yoshiki Higo, and Shinji Kusumoto. Rearranging the Order of Program Statements for Code Clone Detection. In *Proceedings of the 11th IEEE International Workshop on Software Clones*, pp. 15–21, 2 2017.

[27] Yorai Geffen and Shahar Maoz. On Method Ordering. In *Proceedings of the IEEE 24th International Conference on Program Comprehension*, pp. 1–10, May 2016.

[28] Yui Sasaki, Yoshiki Higo, and Shinji Kusumoto. Reordering Program Statements for Improving Readability. In *Proceedings of the 17th European Conference on Software Maintenance and Reengineering*, pp. 361–364, March 2013.

[29] Daniel M. German, Bram Adams, and Kate Stewart. cregit: Token-level blame information in git version control repositories. *Empirical Software Engineering*, Vol. 24, No. 4, pp. 2725–2763, Aug 2019.

[30] Vasiliki Efstathiou and Diomidis Spinellis. Semantic Source Code Models Using Identifier Embeddings. In *Proceedings of the 16th International Conference on Mining Software Repositories*, pp. 29–33, 2019.

[31] Hamid Abdul Basit, Simon J. Puglisi, William F. Smyth, Andrew Turpin, and Stan Jarzabek. Efficient Token Based Clone Detection with Flexible Tokenization. In *Proceedings of the 6th Joint Meeting on European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering: Companion Papers*, pp. 513–516. Association for Computing Machinery, 2007.

[32] Dan Gopstein, Jake Iannacone, Yu Yan, Lois DeLong, Yanyan Zhuang, Martin K.-C. Yeh, and Justin Cappos. Understanding Misunderstandings in Source Code. In *Proceedings of the Joint Meeting on Foundations of Software Engineering*, pp. 129–139. Association for Computing Machinery, 2017.

[33] Mark Gabel and Zhendong Su. A Study of the Uniqueness of Source Code. In *Proceedings of the Eighteenth ACM SIGSOFT International Symposium on Foundations of Software Engineering*, pp. 147–156. Association for Computing Machinery, 2010.

[34] Dawn Lawrie and Dave Binkley. Expanding Identifiers to Normalize Source Code Vocabulary. In *Proceedings of the 27th IEEE International Conference on Software Maintenance*, pp. 113–122, Sep. 2011.

[35] Carl Boettiger. An Introduction to Docker for Reproducible Research. *Operating Systems Review*, Vol. 49, No. 1, pp. 71–79, 2015.

[36] Mojtaba Shahin, Muhammad Ali Babar, and Liming Zhu. Continuous Integration, Delivery and Deployment: A Systematic Review on Approaches, Tools, Challenges and Practices. *IEEE Access*, Vol. 5, pp. 3909–3943, 2017.

[37] CommentRemover. `https://github.com/YoshikiHigo/CommentRemover`.

[38] JavaTokenizer. `https://github.com/DoiMasayuki/JavaTokenizer`.

[39] Thomas Zimmermann, Nachiappan Nagappan, Harald Gall, Emanuel Giger, and Brendan Murphy. Cross-Project Defect Prediction: A Large Scale Experiment on Data vs. Domain vs. Process. In *Proceedings of the 7th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on The Foundations of Software Engineering*, pp. 91–100. Association for Computing Machinery, 2009.

[40] Zoltán Tóth, Péter Gyimesi, and Rudolf Ferenc. A Public Bug Database of GitHub Projects and Its Application in Bug Prediction. In *Proceedings of the International Conference on Computational Science and Its Applications,*, pp. 625–638. Springer International Publishing, 2016.

[41] PMD. `https://pmd.github.io/`.

[42] Tin Kam Ho. Random Decision Forests. In *Proceedings of the Third International Conference on Document Analysis and Recognition (Volume 1) - Volume 1*, p. 278. IEEE Computer Society, 1995.

[43] Joel Myerson, Leonard Green, and Missaka Warusawitharana. Area under the curve as a measure of discounting. *Journal of the experimental analysis of behavior*, Vol. 76 2, pp. 235–43, 2001.

[44] Nancy Cook. Use and Misuse of the Receiver Operating Characteristic Curve in Risk Prediction. *Circulation*, Vol. 115, pp. 928–35, 03 2007.

[45] Henrike Barkmann, Rudiger Lincke, and Welf Lowe. Quantitative Evaluation of Software Quality Metrics in Open-Source Projects. In *Proceedings of the International Conference on Advanced Information Networking and Applications Workshops*. IEEE Computer Society, 2009.

[46] Abram Hindle, Earl T. Barr, Zhendong Su, Mark Gabel, and Premkumar Devanbu. On the Naturalness of Software. In *Proceedings of the 34th International Conference on Software Engineering*, pp. 837–847. IEEE Press, 2012.

[47] Michele Tufano, Cody Watson, Gabriele Bavota, Massimiliano Di Penta, Martin White, and Denys Poshyvanyk. Deep Learning Similarities from Different Representations of Source Code. In *Proceedings of the 15th International Conference on Mining Software Repositories*, pp. 542–553. ACM, 2018.