2021 IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER) | 978-1-7281-9630-5/20/$31.00 ©2021 IEEE | DOI: 10.1109/SANER50967.2021.00052

Evaluating Bug Prediction under Realistic Settings

Sho Ogino*, Yoshiki Higo* and Shinji Kusumoto*
*Graduate School of Information Science and Technology, Osaka University, Japan
{s-ogino, higo, kusumoto}@ist.osaka-u.ac.jp

***Abstract*—Bug prediction is expected to reduce the cost of quality assurance. To build a reliable bug prediction model, we should use realistic settings that satisfy all three of the following conditions. (1) We should build a dataset in a way that allows us to evaluate the prediction performance of the model correctly. (2) We should adopt the optimal granularity of bug prediction to minimize the cost of quality assurance. (3) We should use a dependent variable that correctly represents the presence or absence of bugs in the software modules to be predicted. However, no research has been conducted on bug prediction models built under the above realistic settings. Consequently, we established the following two objectives in this research. (1) We experimentally evaluate the prediction performance of bug prediction models built under realistic settings. (2) We propose techniques to improve the prediction performance of bug prediction models built under realistic settings. The first objective has now been achieved. Our experimental results show that the F-Measure of the bug prediction models built under realistic settings is only 0.19. Thus, there are still some issues to be solved to build a high-performance bug prediction model under realistic settings.**

***Index Terms*—quality assurance; bug prediction; machine learning**

I. Introduction

In recent years, the scale of software development has been steadily increasing. Under this situation, techniques to reduce development costs are indispensable. Techniques to reduce the cost of quality assurance such as reviews are especially important because quality assurance occupies a large amount of the development cost [1].

Bug prediction is a technique to reduce the cost of quality assurance. Bug prediction means predicting the presence or absence of bugs in the software modules (e.g. source files). Identifying buggy modules and intensively reviewing them can reduce the cost of quality assurance.

To properly evaluate the performance of a bug prediction model, the dataset must be properly built. Traditionally, a dataset has been built by calculating dependent variables for software modules existing in a project at a certain time point and then calculating independent variables using the development history from the beginning of the project to the time point [2], [3].

However, Pascarella et al. claimed datasets that are built in such a way as unrealistic because they contain data that cannot realistically be obtained [4]. Pascarella et al. proposed the release-by-release technique to build a dataset that avoids the problem, and they investigated the prediction performance with a realistic dataset. They concluded that low performance was measured and that challenges remain in building a high-performance bug prediction model with a realistic dataset.

We consider that Pascarella et al.'s bug prediction models built with the *release-by-release* technique are controversial. The problem with their model is that the dependent variable is not the indicator "whether a bug exists in the software module at that point in time (*isBuggy*)", but the indicator "whether a bug has been fixed at least once in the past period (*hasBeenFixed*)". This is because there can be a situation where a software module is "false for hasBeenFixed but true for isBuggy", which means a model built with a dataset whose dependent variable is *hasBeenFixed* (*hasBeenFixed* model) may not predict bugs correctly.

As mentioned above, the prior studies built models under unrealistic settings on dataset building and dependent variables. To build reliable bug prediction models, we should use realistic settings. However, no research has been conducted on bug prediction models built under realistic settings. In this study, the following two objectives were set up and investigated.

1) We experimentally evaluate the performance of bug prediction models built under realistic settings.
2) We propose a technique to improve the performance of bug prediction models built under realistic settings.

The first objective has been achieved at this moment. Our experimental results showed that the F-Measure of the bug prediction model built under realistic settings is only 0.19, and there are still some issues to be solved to build high-performance bug prediction models under realistic settings. In the near future, we will propose a way to improve performance.

II. Background

Various settings exist for building bug prediction models. Herein, we introduce two settings, (1) granularity of prediction target and (2) way of building a dataset, as items closely related to this study.

A. Granularity of prediction target

There are three granularities of target that have been often used in bug prediction models: source file, method, and commit. In this study, we focus on method-level bug prediction because Pascarella et al. adopted method for predictive granularity and one of the goals of this study is to confirm the truth of Pascarella et al.'s conclusions.

B. Way of building a dataset

To properly evaluate the performance of a bug prediction model, the dataset must be built in a proper way. Traditionally,

978-1-7281-9630-5/21/$31.00 ©2021 IEEE
DOI 10.1109/SANER50967.2021.00052

independent variables have been calculated using the development history from the beginning of the project to a certain time point [2], [3]. Pascarella et al. claimed building a dataset in this way as impractical because the dataset contains data that cannot realistically be retrieved.

Pascarella et al. proposed the *release-by-release* technique as a suitable way to build a dataset that avoids such a problem. *Release-by-release* calculates dependent and independent variables for the software modules present at a given release of the project. The independent variables are calculated using the development history from the previous release to the given release. The procedure for building a dataset based on the *release-by-release* is described in detail in section V-B.

III. Research Questions

In this paper, we investigate the following four Research Questions.

RQ1. Is *hasBeenFixed* proper as a dependent variable for building bug prediction models?

Through RQ1, we try to quantitatively judge whether *hasBeenFixed* is proper as a dependent variable for building bug prediction models. If most of the buggy methods are not captured by *hasBeenFixed*, then models to predict *hasBeenFixed* (*hasBeenFixed* models) cannot correctly predict buggy methods, which means that *hasBeenFixed* is inappropriate as a dependent variable for bug prediction models.

RQ2. Is the prediction performance of *isBuggy* models lower than *hasBeenFixed* models'?

Through RQ2, we compare *hasBeenFixed* models and *isBuggy* models from the perspective of prediction performance. If the performance of *isBuggy* models is lower than *hasBeenFixed* models', there are still some issues in this research area.

RQ3. Under the realistic settings, which machine learning algorithm provides the best prediction performance?

Through RQ3, we identify the optimal machine learning algorithm under the following realistic settings.

Granularity method
dataset building *release-by-release*
Dependent variable *isBuggy*

RQ4. Under the realistic settings, how does prediction performance change just as the development history accumulates?

Through RQ4, we try to obtain insights into whether the machine learning algorithms should be varied with the amount of development history.

IV. configurations

In this Section, we describe the experimental configurations common to the experiments described in the following sections.

A. Target projects

Target projects in this study are listed in Table I. Those projects were selected for the following four reasons.

- Their Git repositories are available.
- They are written in Java.
- They adopt semantic versioning.
- There are at least four releases in their projects.

B. How to identify releases

In this study, we build datasets following the *release-by-release* technique. It is necessary to identify the timing of each release to use the *release-by-release* technique. We follow the technique proposed by Pascarella et al. [4] to identify the releases of the projects employing semantic versioning as a versioning scheme. Semantic versioning expresses versions in the form of X.Y.Z (e.g., 1.2.1), and X represents the major version. We identify only releases of major versions, which are versions where both Y and Z are zero (e.g., 2.0.0).

C. How to calculate hasBeenFixed

For each method, *hasBeenFixed* is calculated as follows.

step1 We collect the project repository and bug reports whose status is *FIXED*.

step2 Bug reports are issued for previously discovered bugs, and every bug report has an ID. In this experiment, A commit whose commit message contains bug report ID is regarded as a bug-fixing commit. We search the repository for the bug-fixing commits.

step3 Suppose that a line of method A is changed in a bug-fixing commit. If the commit falls within the interval from the $(n-1)$-th release (R_{n-1}) to the n-th release (R_n), then method A is *true* for *hasBeenFixed* at R_n.

step4 When the above operations are performed for all bug reports, methods that are not *true* for *hasBeenFixed* are judged to be false for *hasBeenFixed*.

TABLE I
TARGET PROJECTS

Project Name	development period	releases	commits	bug reports	methods	Percentage of buggy methods
lucene-solr(LUC)	6,967 days	9	34,319	6,232	72,356	2.3 %
wicket(WIC)	5,860 days	7	20,972	2,871	22,961	2.5 %
cassandra(CAS)	4,239 days	4	25,671	5,402	28,567	5.6 %
linuxtools(LIN)	4,197 days	8	10,733	2,201	28,644	1.2 %
egit(EGI)	4,021 days	5	6,473	2,529	8,082	7.2 %
jgit(JGI)	4,029 days	5	8,014	700	14,098	1.1 %
eclipse.jdt.core(ECL)	7,062 days	4	24,804	9,716	23,910	7.9 %
poi(POI)	6,824 days	4	10,421	2,620	34,218	3.6 %

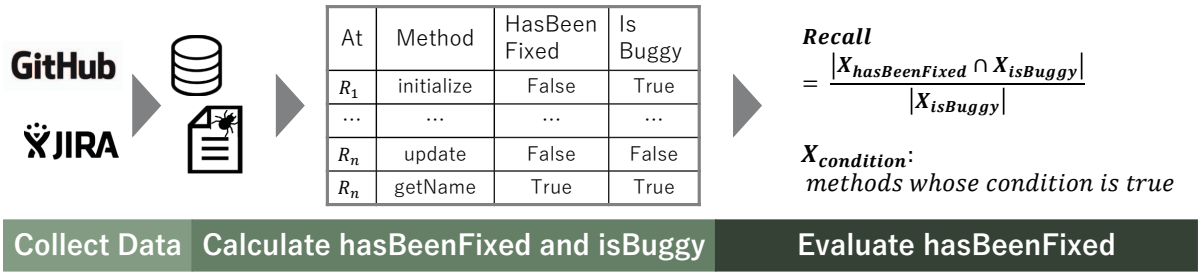

Fig. 1. procedure of the experiment for RQ1

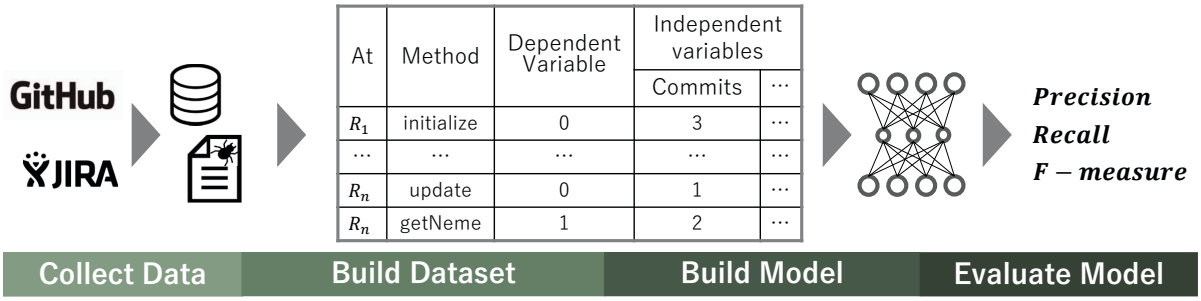

Fig. 2. procedure of the experiment for RQ2

D. How to calculate isBuggy

We used the implementation of literature [5] of an SZZ algorithm [6] to calculate *isBuggy* as follows.

step1 The same as step 1 of the *hasBeenFixed* calculation.

step2 The same as step 2 of the *hasBeenFixed* calculation.

step3 Suppose that a line of method A is changed in a bug-fixing commit. We identify the commit that inserted the changed line with the "git blame", and regard the commit as a bug-inducing commit.

step4 If a bug-fixing commit for method A was made after R_n and the bug-inducing commit for the bug was made between R_{n-1} and R_n, then method A is *true* for *isBuggy* at R_n.

step5 When the above operations are performed for all bug reports, methods that are not *true* for *isBuggy* are judged to be false for *isBuggy*.

V. Experimental Results

A. RQ1

In RQ1, we investigate whether *hasBeenFixed* accurately captures buggy methods. An overview of the experiment is shown in Figure 1.

Procedure. We conducted the following steps to answer RQ1: we (1) collected the repositories and the bug reports of the target projects, (2) calculated *isBuggy* and *hasBeenFixed* for each target method, and (3) evaluated the properness of *hasBeenFixed* quantitatively.

Evaluation metric. We use *recall* defined below to evaluate the properness of *hasBeenFixed* as a dependent variable for building bug prediction models.

$$Recall = \frac{|X_{hasBeenFixed} \cap X_{isBuggy}|}{|X_{isBuggy}|}$$

$X_{hasBeenFixed}$ is a set of methods that are *true* for *hasBeenFixed*, and $X_{isBuggy}$ is a set of methods that are *true* for *isBuggy*. A method called "initialize" that is shown in Figure 1 is true for *isBuggy* but false for *hasBeenFixed* at R_1. So "initialize" existing at R_1 is an element of $X_{isBuggy}$, but not an element of $X_{hasBeenFixed}$. Suppose the evaluated recall is less than 0.5. In that case, we judge *hasBeenFixed* to be improper as a dependent variable for bug prediction models because *hasBeenFixed* models cannot predict more than half of the buggy methods.

Results. The results of the experiment are listed in Table II. Table II lists recall, which is the percentage of the buggy methods captured by *hasBeenFixed*, for each project. The recalls are all significantly below 0.5. We conclude that *hasBeenFixed* is improper as a dependent variable for bug prediction models because *hasBeenFixed* cannot capture more than half of the buggy methods.

B. RQ2

In RQ2, we build *hasBeenFixed* models and *isBuggy* models to compare their performance. An overview of the experiment is shown in Figure 2.

Procedure. We conducted the following steps to investigate RQ2: we (1) collected the repositories and the bug reports of the target projects, (2) built datasets based on the collected data, (3) built bug prediction models based on the datasets, and (4) evaluated the performance of the bug prediction models.

Dependent variable. We adopted *hasBeenFixed* and *isBuggy* as dependent variables and built regression models for

TABLE II
Recall of hasBeenFixed for isBuggy

Project	LUC	WIC	CAS	LIN	EGI	JGI	ECL	POI	ALL
Recall	0.22	0.22	0.36	0.16	0.34	0.16	0.43	0.20	0.28

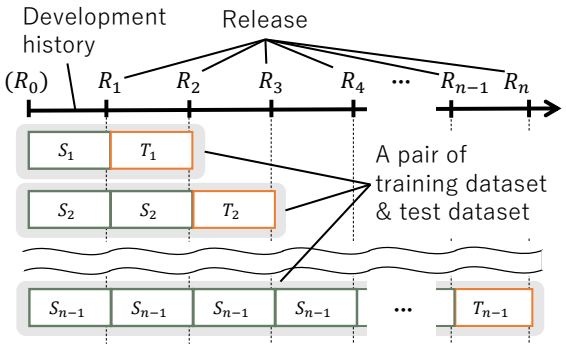

Fig. 3. overview of datasets for each project

each of them. The definitions of the metrics are given in section IV-C and IV-D, respectively.

Independent variables. We adopt the product and process metrics defined by Giger et al. [2] as independent variables. The metrics are listed in Tables III and IV.

Building a dataset. This study follows *release-by-release* technique to calculate pairs of a training dataset and a test dataset. Based on this technique, we can make $n-1$ pairs of training dataset S_k and test dataset T_k from a project that has been released for n times, as shown in Figure 3. The pairs are calculated as follows.

1) We calculate pairs of Dependent and independent variables (instances) using the development history between R_{k-1} and R_k for each release R_k (R_0 is the beginning of the development).
2) We build test dataset T_k of the instances calculated from the development history between R_{k-1} and R_k.
3) We build training dataset S_k of the instances calculated from the development history before R_{k-1}.
4) We over-sample instances in S_k which are true for dependent variable to address class imbalance problem [7].

TABLE III
INDEPENDENT VARIABLES (PRODUCT METRICS)

Metrics name	Description
FanIn	Number of methods that reference a given method
FanOut	Number of methods referenced by a given method
LocalVar	Number of local variables in the body of a method
Parameters	Number of parameters in the declaration of a method
CommentRatio	Ratio of comments to source code (line based)
CountPath	Number of possible paths in the body of a method
Complexity	McCabe Cyclomatic complexity of a method
execStmt	Number of executable source code statements
maxNesting	Maximum nested depth of all control structures

TABLE IV
INDEPENDENT VARIABLES (PROCESS METRICS)

Metrics Name	Description
MethodHistories	Number of times a method was changed
Authors	Number of distinct authors that changed a method
StmtAdded	Sum of all source code statements added
MaxStmtAdded	Maximum StmtAdded
AvgStmtAdded	Average of StmtAdded
StmtDeleted	Sum of all source code statements deleted
MaxStmtDeleted	Maximum of StmtDeleted
AvgStmtDeleted	Average of StmtDeleted
Churn	Sum of stmtAdded - stmtDeleted
MaxChurn	Maximum churn for all method histories
AvgChurn	Average churn per method history
Decl	Number of method declaration changes
Cond	Number of condition changes over all revisions
ElseAdded	Number of added else-parts over all revisions
ElseDeleted	Number of deleted else-parts over all revisions

Machine learning algorithm. The *Random Forest* (RF) [8] was adopted as a machine learning algorithm because RF has been commonly used in the previous studies [2]–[4] and the time cost for building models is small.

Hyperparameter tuning. Hyperparameter tuning is essential to build high-performance models [9]. We performed 10 hours of hyperparameter tuning for each model.

Evaluation metrics. We adopted the following measures to evaluate prediction performance.

$$Precision = \frac{|TP|}{|FP+TP|}$$
$$Recall = \frac{|TP|}{|FN+TP|}$$
$$F-Measure = \frac{2 \times Recall \times precision}{Recall + precision}$$

TP (True Positive) is a set of methods that have been classified as buggy by a model and actually buggy. FN (False Negative) is a set of methods that have been classified as not buggy by a model and actually buggy. FP (False Positive) is a set of methods that have been classified as buggy by a model and actually not buggy.

Results. The results of the experiment are listed in Table V. The table shows the average prediction performance for both the *hasBeenFixed* models and the *isBuggy* models.

The F-Measure of the models adopting *isBuggy* as a realistic dependent variable is about half that of *hasBeenFixed*. We conclude that *isBuggy* models' performance is quite lower than *hasBeenFixed* models' and there are still issues in this research area.

C. RQ3

In RQ3, we investigate which machine learning algorithm provides the best performance under realistic settings. The configurations to build and evaluate models are the same as RQ2's except for dependent variable and machine learning algorithm.

Dependent variable. We set *isBuggy* as the dependent variable. The definition is given in section IV-D.

Machine learning algorithm. We test *Random Forest* (RF) and *Deep Neural Network* (DNN) [10] as machine learning algorithms. DNN is selected because DNN can compute expressive models even though they are not always better than RF models in terms of prediction performance.

Results. The results of the experiments are listed in Table VI. The table shows the average prediction performance for both RF and DNN. We conclude that RF models are better than DNN models from the perspective of F-Measure.

TABLE V
PREDICTION PERFORMANCE FOR DEPENDENT VARIABLES

	Precision	Recall	F-Measure
HasBeenFixed	0.33	0.75	0.40
IsBuggy	0.15	0.61	0.19

TABLE VI
PREDICTION PERFORMANCE FOR MACHINE LEARNING ALGORITHMS

	Precision	Recall	F-Measure
RF	0.15	0.61	0.19
DNN	0.13	0.64	0.18

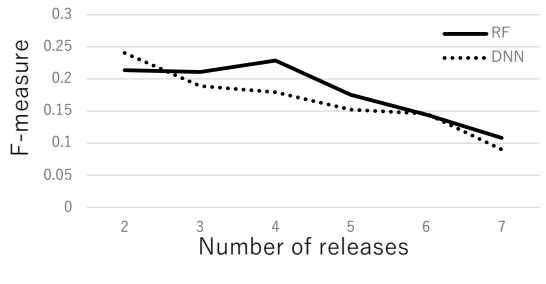

Fig. 4. performance transition as the development history accumulates

D. RQ4

We follow the *release-by-release* technique to build datasets. Under this technique, the size of the training dataset is proportional to the length of development history (the number of releases). In RQ4, we investigate how prediction performance of models built under the realistic settings changes as the development history accumulates. The configurations to build models are the same as RQ3's.

Results.

Figure 4 shows the average performance of models to predict buggy methods at the release. In the results of the experiment, there are two points worthy of special mention.

Firstly, the prediction performance decays as the development history accumulates. This may be because the buggy methods' features change just as the development history accumulates. However, this is a hypothesis, and we will test it in the near future.

Secondly, DNN models are more powerful at an early stage of development, and RF models are more powerful at a later stage of development. This may be because the buggy methods' features change just as the development history accumulates, and DNN models strongly learned the buggy methods' features at the early stage of development.

We conclude that DNN is better for building bug prediction models under realistic settings at the second release, and RF at subsequent releases.

VI. Threats to validity.

Herein, we describe possible threats to our study.

Dependent variable. In this study, we use bug reports whose status is *FIXED* to calculate *hasBeenFixed* and *isBuggy*. On the other hand, the target projects may include latent bugs, which are not exposed yet and any bug reports for them have not been made yet. The presence of latent bugs may harm the accuracy of dependent variables.

Hyperparameter tuning. Hyperparameter tuning is essential to build high-performance models using machine learning algorithms. In this study, we performed 10 hours of hyperparameter tuning for each model. If models are tuned for a longer time, higher performance models may be built.

SZZ algorithm. We used the implementation of literature [5] of an SZZ algorithm [6] to calculate *isBuggy*. However, the accuracy of the SZZ algorithm still has room for improvement [11], [12]. This problem may harm the accuracy of *isBuggy*.

VII. Conclusion

This paper is the first step to build reliable bug prediction models under realistic settings. We first investigated the properness of the metric *hasBeenFixed* as a dependent variable for bug prediction models. Then, we built bug prediction models under realistic settings and evaluated their performance.

As a result, *hasBeenFixed* was able to capture only 28% of buggy methods. Thus, *hasBeenFixed* is improper as a dependent variable for bug prediction models. The F-Measure of the bug prediction models built under realistic settings was only 0.19, and it turned out to be challenging to build practical bug prediction models under realistic settings. From those results, we can say that there are still some issues to be solved to build high-performance bug prediction models under realistic settings.

We are currently trying to construct a new technique to improve bug prediction models' performance under realistic settings. Specifically, we are considering investigating how prediction performance changes if new independent variables are introduced.

Acknowledgment

This work was supported by MEXT/JSPS KAKENHI 20H04166.

References

[1] T. Britton, L. Jeng, G. Carver, P. Cheak, and T. KKatzenellenbogen, "Increasing software development productivity with reversible debugging," accessed 2020-10-16, https://undo.io/media/uploads/files/Undo_ReversibleDebugging_Whitepaper.pdf.

[2] E. Giger, M. D'Ambros, M. Pinzger, and H. Gall, "Method-level bug prediction," in *International Symposium on Empirical Software Engineering and Measurement*, 2012, pp. 171–180.

[3] H. Hata, O. Mizuno, and T. Kikuno, "Bug prediction based on fine-grained module histories," *Proceedings - International Conference on Software Engineering*, pp. 200–210, 2012.

[4] L. Pascarella, F. Palomba, and A. Bacchelli, "Re-evaluating method-level bug prediction," in *2018 IEEE 25th International Conference on Software Analysis, Evolution and Reengineering (SANER)*, 2018, pp. 592–601.

[5] M. Borg, O. Svensson, K. Berg, and D. Hansson, "Szz unleashed: An open implementation of the szz algorithm - featuring example usage in a study of just-in-time bug prediction for the jenkins project," in *Proceedings of the 3rd ACM SIGSOFT International Workshop on Machine Learning Techniques for Software Quality Evaluation*, 2019, pp. 7–12.

[6] J. Sliwerski, T. Zimmermann, and A. Zeller, "When do changes induce fixes?" *SIGSOFT Software Engineering Notes*, vol. 30, no. 4, pp. 1–5, 2005.

[7] N. V. Chawla, *Data Mining and Knowledge Discovery Handbook*. Springer, 2010, pp. 875–886.

[8] A. Liaw and M. Wiener, "Classification and regression by randomforest," in *R news*, vol. 2, no. 3, 2002, pp. 18–22.

[9] J. Bergstra and Y. Bengio, "Random search for hyper-parameter optimization," *Journal of Machine Learning Research*, vol. 13, p. 281–305, 2012.

[10] A. Krizhevsky, I. Sutskever, and G. E. Hinton, "Imagenet classification with deep convolutional neural networks," in *Proceedings of the 25th International Conference on Neural Information Processing Systems*, vol. 1, 2012, pp. 1097–1105.

[11] C. Williams and J. Spacco, "Szz revisited: Verifying when changes induce fixes," in *Proceedings of the 2008 Workshop on Defects in Large Software Systems*, 2008, pp. 32–36.

[12] D. A. da Costa, S. McIntosh, W. Shang, U. Kulesza, R. Coelho, and A. E. Hassan, "A framework for evaluating the results of the szz approach for identifying bug-introducing changes," *IEEE Transactions on Software Engineering*, vol. 43, no. 7, pp. 641–657, 2017.