A Dynamic Model Selection Approach to Mitigate the Change of Balance Problem in Cross-Version Bug Prediction

Hiroshi Demanou[1], Akito Monden[1] and Masateru Tsunoda[2]

[1]*Okayama University, 1-1, Tsushima-Naka, 3-chome, Kita-Ku, Okayama 700-8530, Japan*
[2]*Kindai University, 3-4-1, Kowakae, Higashiosaka City, Osaka 577-8502, Japan*

Abstract
This paper focuses on the "change of balance" problem in cross-version bug prediction where the percentage of buggy modules changes between different versions. Such difference badly affects the prediction performance. To mitigate this problem, this paper employs a dynamic model selection approach equipped with two prediction models (always-buggy model and always-non-buggy model) and Bandit algorithm to select better models in each one-module-by-one prediction. An experiment with data sets of 20 releases of 10 open source software showed that the proposed approach can improve F1-measure compared with the conventional cross-version prediction.

Keywords
software quality assurance, defect-prone module prediction, Bandit algorithm

1. Introduction

Defect-prone software module prediction (or simply, bug prediction) has been studied for the effective software quality assurance [1][2][3]. Typically, prediction is held in cross-version situation where a prediction model is built from data of a past project, and it is applied to the next version of that project. Based on the prediction result, practitioners can allocate limited testing efforts to the defect-prone (buggy) modules to find more bugs with smaller effort.

However, it has been pointed out that such cross-version prediction very often does not work well because of concept drift [4][5]. As one of the factors of concept drift, this paper focuses on the "change of balance" between the number of buggy modules and not buggy modules. Indeed, such balance very often changes between different versions of software. For example, in case of Ant project, the percentage of buggy modules was 10.9% in version 1.5 while it becomes 29.3% in next version 1.6 (as shown in Table 2). Such a difference badly affects the prediction performance of the models in general.

To mitigate this problem, assuming that the modules are predicted one by one manner, this paper employs a dynamic model selection approach equipped with two prediction models: (1) always-buggy model and (2) always-non-buggy model. The always-buggy model outputs "buggy" to any modules input to the model, while the always-non-buggy model outputs "not buggy" to any modules. By employing Bandit algorithm to give scores to two models in each one-by-one prediction, we expect that the relatively better model is wisely selected regardless of the percentage of bugs in the target version.

To date, there are several attempts to employ Bandit algorithm in bug prediction [6][7], none of them tries to solve the "change of balance" problem in cross-version bug prediction.

To evaluate the proposed method, this paper conducts an empirical study using datasets of 20 releases of 10 open source software projects.

2. Cross-version bug prediction and its balance problem

2.1. Cross-version bug prediction

To date, various bug prediction techniques have been proposed and evaluated [1][2][3][7][8]. Bug prediction is carried out before software testing and/or code review. In this paper, we focus on bug module classification, which aims to classify a module as buggy (containing one or more bugs) or not buggy (containing no bug). The objective variable is the probability that a module belongs to the buggy class. Typically, prediction is held in cross-version manner where a prediction model is built from data of a past project, and it is applied to the next version of that project.

2.2. Problem of change of balance between versions

In cross-version bug prediction, balance between the number of buggy modules and not buggy modules is a dominant factor of prediction accuracy. For example, if

QuASoQ 2022: 10th International Workshop on Quantitative Approaches to Software Quality, December 06, 2022, virtual
demanouconstance@yahoo.fr (H. Demanou); monden@okayama-u.ac.jp (A. Monden); tsunoda@info.kindai.ac.jp (M. Tsunoda)

buggy modules are extremely less than not buggy modules, it is very difficult to gain high precision [9]. Moreover, such balance very often changes between different versions, and this cause bad prediction performance in cross-version prediction.

In case the older version has fewer bugs than the newer version, precision tends to become large and recall tends to become small. On the other hand, if the older version has more bugs than the newer version, precision tends to be small and recall tends to be large. For example, in case of Ant project, the older version has fewer bugs (as shown in Table 2), and in such a case, precision is high (.600) but recall becomes small (.163) as later shown in Table 4. Such change of balance between versions very often happens in cross-version prediction, making it difficult to obtain high prediction accuracy.

3. Proposed solution

3.1. Bandit algorithm

Here, we introduce Bandit algorithm which we employ in this paper to mitigate the problem of balance between versions. The K-arm bandit problem is a problem where a user faced with slot machines, must decide which machines to play. Each machine gives different average reward that the user does not know in advance. The goal is to maximize the user's cumulative reward. Consider K slot machines. In game turn n, the user will receive a reward which depends on the machine he chooses. A basic example is the case where machine i brings a reward of 1 with probability p and -1 with probability $1 - p$. In our study, considering that a user wants to conduct unit testing for a set of modules, instead of selecting a slot machine, the user selects a module (*i.e.*, a source file) one-by-one and tries to find the bug prediction model that brings the highest average reward (i.e. prediction performance) to conduct testing. The strategy for the armed bandit problem is an algorithm that chooses the next prediction model based on previous choices and the rewards obtained. This paper introduces the most basic algorithm called epsilon-greedy algorithm. In this algorithm, in each trial, an arm is selected at random for a proportion ϵ, and the best arm (having the largest total reward) is selected for a proportion $1 - \epsilon$. The proper ϵ value may depends on the context. As a simple example, here we consider two arms X and Y exist, and want to maximize the cumulative reward by selecting appropriate arm. An example of arm selection in each trial is illustrated in Table 1. Each trial is proceeded as follows.

1. In the initial trial, an arm is selected randomly. Arm X is selected in this case. Earned reward is -1.

Table 1
An example of applying a bandit algorithm.

Trial	Selected Arm	Earned reward	X's total reward	Y's total reward
1	X	-1	-1	0
2	Y	1	-1	1
3	X	1	0	1

2. Arm Y is selected for a proportion $1 - \epsilon$, as Arm Y's total reward is larger than that of X. Earned reward is 1.
3. The arm is randomly selected for a proportion ϵ. Arm X is selected in this case. Earned reward is 1.

As we illustrated above, arm X received the reward of -1 in the initial trial, and this makes arm X difficult to be selected in later trials. However, the parameter ϵ enables arm X to be selected sometimes to give it to receive positive reward.

3.2. Basic idea to solve the change of balance problem

The problem of change of balance between two versions can be classified into the following two cases (a) the newer version has fewer bugs, or (b) the newer version has greater bugs. The problem here is that it is not possible to determine in advance whether we are in case (a) or (b). However, if the bug prediction and testing is carried out on a one-by-one basis, we are gradually getting to be aware of it. That is, we assume the following process: (1) we conduct bug prediction to all modules, (2) we pick a single module that has highest probability of being "buggy", (3) conduct testing if the module is predicted as "buggy" and (4) now we know the prediction is correct or wrong. Repeating the above process, we expect that the false-positive will increase if we are in case (a). On the contrary, we expect that the true-positive will increase if we are in case (b). Based on the above expectation, our idea is to employ two different types of bug prediction models as follows:

1. Always-buggy model: It always predicts that there is a bug in a module.
2. Always-non-buggy model: It always predicts that there is no bug in a module.

Firstly, we employ a normal bug prediction model to predict all modules to obtain the probability of being "buggy" of each module. Then, a one-by-one prediction process is carried out such that: a module having the largest probability is picked, the prediction model is selected by Bandit algorithm, prediction result is obtained, the reward is given to all prediction models based on the correctness of the prediction

3.3. Proposed algorithm

Based on the basic idea above, we propose an algorithm to select a bug prediction model as follows.
(Step. 1) Probability computation

In this step, bug prediction is conducted to all modules using the ordinary model to obtain the probability of being "buggy" of all modules.
(Step. 2) Target module selection

We pick a single module that has the highest probability of being "buggy", from a list of unselected modules. The reason why we start with the buggiest module is that, it is natural for a practitioner to focus first on the riskiest part of the software and examine it to see if there is any bug or serious problem.
(Step. 3) Model selection based on the epsilon-greedy algorithm

Generate a random number x of $[0, 1]$; and,

3-1) if $x < \epsilon$, a bug prediction model is randomly selected from two models (always-buggy and always-non-buggy).

3-2) If $x \geq \epsilon$, select a bug prediction model with the largest sum of recent reward.
(Step. 4) Prediction

Conduct bug prediction with the selected model.
(Step. 5) Testing

Conduct testing if the selected model predicts the target module as "buggy." No test is carried out if "not buggy" is predicted. This is because bug prediction aims to reduce the cost of testing by testing only the modules likely to have a bug.
(Step. 6) Rewarding

Assuming that predictions were made by both two models, the reward +1 is given to a model if the prediction was correct, and -1 is given if the prediction was incorrect. Note that rewarding is conducted only if testing is conducted in Step 5. In Section 3-1, in the conventional Bandit algorithm, the reward was calculated for only one selected arm (*i.e.* bug prediction model), but in our proposal, the reward for all models is calculated. Because, in the case of a slot machine, we can only bet by putting money in one of them each time, but since bug prediction can be executed by all prediction models in every trial, there is no point in limiting the calculation of reward to a single prediction model. Therefore, we decided to use both models in each trial.
(Step. 7) Compute the sum of recent rewards for all models.

Here, we ignore "old" rewards because we want to select a model with good "recent" performance. Therefore, we set a threshold w on the number of trials, and the calculation of total reward includes only the recent w trials. We refer to this threshold w simply as "window size." The optimum w is experimentally determined.
(Step. 8) If the list of unselected modules is empty then end else go to Step. 2.

Table 2
20 releases of 10 data sets used in the experiment.

Project Name	Release	Modules	Modules with bugs	% of modules with bugs
Ant	1.5	293	32	10.9
	1.6	350	92	26.3
Camel	1.4	856	144	16.8
	1.6	945	188	19.9
Forrest	0.7	29	5	17.2
	0.8	32	2	6.3
Ivy	1.4	241	16	6.6
	2.0	352	40	11.4
Jedit	4.2	367	48	13.1
	4.3	492	11	2.2
Log4j	1.1	109	37	33.9
	1.2	205	189	92.2
Lucene	2.2	247	144	58.3
	2.4	340	203	59.7
Poi	2.5	384	248	64.6
	3.0	441	281	63.7
Prop	4	8702	840	9.7
	5	8506	1298	15.3
Synapse	1.0	157	16	10.2
	1.1	222	60	27.0

4. Evaluation

4.1. Data set

As shown in Table 2, this paper uses 20 releases of 10 open source software (OSS) project data sets to conduct cross-release prediction. Each project includes two releases where older release is used as a fit data set (for building a defect prediction model) and newer release is used as a test data set (for evaluation). The percentage of modules widely varies among projects and/or versions (smallest is 2.2% and largest is 92.9%.) Metrics included in these data sets are shown in Table 3. These data sets are donated by Jureczko et al. [10][11] and the details of the data measurement are described in [11]. We obtained these data sets from SeaCraft repository [12].

Table 3
Metrics used in the data sets.

Name	Definition
WMC	Weighted Methods per Class
DIT	Depth of Inheritance Tree
NOC	Number of Children of a class
CBO	Coupling Between Classes
RFC	Response for a Class
LCOM	Lack of Cohesion in Methods
LCOM3	Lack of Cohesion in Methods
NPM	The number of public methods
DAM	Data Access Metric
MOA	Measure of Aggregation
MFA	Measure of Functional Abstraction
CAM	Cohesion Among Methods of Class
IC	Inheritance Coupling
CBM	Coupling Between Methods
AMC	Average Methods Complexity
Ca	Afferent couplings
Ce	Efferent couplings
MaxCC	Maximum value of cyclomatic complexity of methods in a class
AvgCC	Arithmetic mean of cylcomatic complexity of methods in a class
LOC	Lines of Code

4.2. Bug prediction model

This paper employ random forest because it was shown as one of the best models in bug prediction [13] and it shows performance comparable to the modern auto-ML framework [14]. Although there exist various other predictors, improvement of defect prediction accuracy by employing them is out of scope of this study. To build random forest models, we use the statistical computing and graphics toolkit R and its randomForest library. We use the default parameter values of randomForest library, e.g. the number of trees to grow ntree = 500, and the number of variables randomly sampled as candidates as each split mtry = sqrt(p), where p is the number of predictor variables.

4.3. Accuracy measures

This paper employs three commonly used accuracy measures to evaluate the prediction performance: precision, recall and F1-measure.

4.4. Result and discussion

Table 4 shows the result of defect prediction by the conventional method, that is, cross-version bug prediction with random forest. For the project Ivy, the values of precision and recall are zero, in such a case we consider the F1-measure to be zero. The average of precision (0.488) is higher than that of recall (0.322). The average of F1-measure (0.317) is similar to that of recall.

Table 4
The bug prediction performance of the conventional method.

Project Name	Precision	Recall	F1
Ant	.600	.163	.256
Camel	.481	.266	.342
Forrest	.200	.500	.286
Ivy	0	0	0
Jedit	.132	.455	.204
Log4j	.947	.286	.439
Lucene	.650	.685	.667
Poi	.744	.722	.733
Prop	.493	.026	.050
Synapse	.636	.117	.197
Average	.488	.322	.317

Table 5 shows the result of the proposed method for window size $w = N/A$, 10, 50 and 100, and $\epsilon = 0$, .1, .2, .3 and .4. Here, $w = N/A$ means there is no window (it can be considered that $w = \infty$). The gray cells in the table have the highest values in each window size.

For all w and $\epsilon \geq .2$ cases, the average F1-measure was better than that of the conventional method, which suggests the effectiveness of the proposed method. Compared with the conventional method, the average precision was decreased, but the average recall was greatly improved, resulting in the improved F1-measure. Since the overlook of bugs is crucial in software testing, we believe improvement of recall is preferable from the practical point of view.

Interestingly, $\epsilon = .2$ or .3 showed the best performance for all window sizes. Since $\epsilon = .2$ and .3 cases are always better than $\epsilon = 0$ cases, this suggests the effectiveness of the epsilon-greedy algorithm for dynamic model selection. On the other hand, the window size was found to have negative effect on prediction performance since $w = N/A$ cases showed better performance than $w = 10$, 50 and 100 cases. Therefore, it can be said that the window is not necessary in the current form of the proposal. For more detailed analysis, Table 6 shows the prediction performance of the proposed method ($w = N/A$, $\epsilon = .2$) for each data set. Compared to the result of the conventional method (Table 4), 7 data sets (Ant, Camel, Ivy, Log4j, Lucene, Prop and Synapse) showed improvements in F1-measure, while 3 data sets (Forrest, Jedit and Poi) showed decrease in F1-measure. Looking at Table 2, it seems that the Forrest data set is too small to evaluate. It has only 2 buggy modules in new version. Also, the Jedit data set would be inadequate for evaluation since it contain only 11 buggy modules out of 492 modules. When ignoring these two data sets,

Table 5
The average bug prediction performance of the proposed method.

w	ϵ	Precision	Recall	F1
N/A	0	.380	.346	.310
	.1	.410	.437	.365
	.2	.411	.497	.393
	.3	.397	.497	.386
	.4	.393	.525	.390
10	0	.415	.261	.219
	.1	.400	.285	.302
	.2	.421	.383	.356
	.3	.413	.425	.359
	.4	.392	.430	.351
50	0	.371	.318	.263
	.1	.448	.423	.387
	.2	.427	.459	.374
	.3	.395	.401	.349
	.4	.394	.473	.363
100	0	.369	.331	.266
	.1	.408	.436	.366
	.2	.411	.485	.379
	.3	.376	.430	.350
	.4	.379	.460	.363

Table 6
Details of the result of the proposed method (no window, $\epsilon = .2$).

Project Name	Precision	Recall	F1
Ant	.437	.598	.505
Camel	.342	.356	.349
Forrest	.143	.500	.222
Ivy	.035	.025	.029
Jedit	.026	.091	.041
Log4j	.924	.904	.914
Lucene	.603	.897	.721
Poi	.623	.765	.687
Prop	.186	.129	.153
Synapse	.381	.267	.314
Average	.411	.497	.393

the average F1-measure by the conventional method is .357 and that in the proposed method is .459. Regarding the Poi data set, where the proposed method was not effective, the reason for this result may be that the newer version has fewer bugs than the older version. Based on the investigation of gained rewards in each trial in this data set, we found that always-buggy models received many minus rewards, while always-non-buggy models did not. This is considered to be not fair for always-buggy models because testing is conducted only if the prediction result is "buggy." Therefore, always-buggy models have larger chance to get minus rewards than always-non-buggy models. Resolving such asymmetries is an important issue for the future.

5. Threats to validity

In this section we discuss the threats to validity of our work. We used the single prediction method (random forest). Our important future work is to employ other prediction methods to increase the validity of the result. Another issue is that we conducted only one trial (i.e. no repetition) for each prediction. Since random forest can output different results for the same data set, it is our future work to conduct repetitions in predictions.

In this study we used data sets of 20 releases of 10 open source software donated by Jureczko et al. [10][11]. In future, we will consider using data sets from other data sources to increase the generalization of the results.

In addition, we used three commonly-used performance measures (precision, recall and F1-measure) for evaluation. However, there are several criticism to these measures [9]. Therefore, we will consider adding other performance measures such as probability of false alarm (pf) and Matthews Correlation Coefficient (MCC) [15].

Simulating a sectioning command by setting the first word or words of a paragraph in boldface or italicized text is not allowed.

6. Conclusion

In this paper, we proposed an approach to mitigate the "change of balance" problem in cross-version bug prediction. An experimental evaluation with 10 data sets showed that, by using the proposed approach, although the average precision was decreased, the average recall was greatly improved, resulting in the improved F1-measure. Since the overlook of bugs is crucial in general, we believe that improvement of recall helps practitioners in software quality assurance.

There are several future works as we denoted in the threats to validity session. In addition, this paper compared the proposed method with the most basic cross-version prediction using random forest. Since there are attempts to mitigate the class imbalance problem, such as over/under sampling [1], it is our important future work to compare our approach with these methods.

7. Acknowledgement

This work was supported in part by JSPS KAKENHI Grant number JP20K11749 and JP20H05706.

References

[1] K. E. Bennin, J. Keung, P. Phannachitta, A. Monden, S. Mensah, Mahakil: diversity based oversampling approach to alleviate the class imbalance issue in software defect prediction, IEEE Trans. Software Engineering 44 (2018) 534–550.

[2] Y. Kamei, S. Matsumoto, A. Monden, K. Matsumoto, B. Adams, A. E. Hassan, Revisiting common bug prediction findings using effort aware models, Proc. 26th IEEE Int'l Conference on Software Maintenance (ICSM2010) (2010) 1–10.

[3] A. Monden, T. Hayashi, S. Shinoda, K. Shirai, J. Yoshida, M. Barker, K. Matsumoto, Assessing the cost effectiveness of fault prediction in acceptance testing, IEEE Transactions on Software Engineering 39 (2013) 1345–1357.

[4] J. Ekanayake, J. Tappolet, H. C. Gall, A. Bernstein, Tracking concept drift of software projects using defect prediction quality, Proc. IEEE Working Conference on Mining Software Repositories (2009).

[5] M. A. Kabir, J. W. Keung, K. E. Bennin, M. Zhang, Assessing the significant impact of concept drift in software defect prediction, Proc. IEEE 43rd Annual Computer Software and Applications Conference (COMPSAC'19) (2019).

[6] T. Asano, M. Tsunoda, K. Toda, A. Tahir, K. E.Bennin, K. Nakasai, A. Monden, K. Matsumoto, Using bandit algorithms for project selection in cross-project defect prediction, Proc. International Conference on Software Maintenance and Evolution (ICSME) (2021) 19–33.

[7] T. Hayakawa, M. Tsunoda, K. Toda, K. Nakasai, A. Tahir, K. E. Bennin, A. Monden, K. Matsumoto, A novel approach to address external validity issues in fault prediction using bandit algorithms, IEICE Transactions on Information and Systems E104.D (2021) 327–331.

[8] T. M. Khoshgoftaar, A. Pandya, D. Lanning, Application of neural networks for predicting program fault, Annals of Software Engineering 1 (1995) 141–154.

[9] T. Menzies, A. Dekhtyar, J. Distefano, J. Greenwald, Problems with precision: A response to comments on data mining static code attributes to learn defect predictors, IEEE Transactions on Software Engineering 33 (2007) 637–640.

[10] M. Jureczko, L. Madeyski, Towards identifying software project clusters with regard to defect prediction, Proc. 6th International Conference on Predictive Models in Software Engineering (PROMISE'10) (2010) 9:1–9:10.

[11] M. Jureczko, D. Spinellis, Using object-oriented design metrics to predict software defects, In Models and Methods of System Dependability. Oficyna Wydawnicza Politechniki WrocÅCawskiej (2010) 69–81.

[12] T. Menzies, R. Krishna, D. Pryor, The seacraft repository of empirical software engineering data, https://zenodo.org/communities/seacraft (2017).

[13] S. Lessmann, B. Baesens, C. Mues, S. Pietsch, Benchmarking classification models for software defect prediction: A proposed framework and novel findings, IEEE Trans. on Software Engineering 34 (2008) 485–496.

[14] K. Tanaka, A. Monden, Z. Yücel, Software defect prediction using automated machine learning, Proc. 20th IEEE/ACIS International Conference on Software Engineering, Artificial Intelligence, Networking and Parallel/Distributed Computing (SNPD 2019) (2019) 490–494.

[15] D. Chicco, G. Jurman, The advantages of the matthews correlation coefficient (mcc) over f1 score and accuracy in binary classification evaluation, BMC Genomics 21 (2020).