IEICE TRANSACTIONS

on Information and Systems

DOI:10.1587/transinf.2020EDL8098

Publicized:2020/10/30

A PUBLICATION OF THE INFORMATION AND SYSTEMS SOCIETY

The Institute of Electronics, Information and Communication Engineers

Kikai-Shinko-Kaikan Bldg., 5-8, Shibakoen 3 chome, Minato-ku, TOKYO, 105-0011 JAPAN

LETTER

A Novel Approach to Address External Validity Issues in Fault Prediction Using Bandit Algorithms

Teruki HAYAKAWA†, *Nonmember*, Masateru TSUNODA†, Koji TODA††, *Members*, Keitaro NAKASAI†††, Amjed TAHIR††††, Kwabena Ebo BENNIN†††††, *Nonmembers*, Akito MONDEN††††††, and Kenichi MATSUMOTO†††, *Members*

SUMMARY Various software fault prediction models have been proposed in the past twenty years. Many studies have compared and evaluated existing prediction approaches in order to identify the most effective ones. However, in most cases, such models and techniques provide varying results, and their outcomes do not result in best possible performance across different datasets. This is mainly due to the diverse nature of software development projects, and therefore, there is a risk that the selected models lead to inconsistent results across multiple datasets. In this work, we propose the use of bandit algorithms in cases where the accuracy of the models are inconsistent across multiple datasets. In the experiment discussed in this work, we used four conventional prediction models, tested on three different dataset, and then selected the best possible model dynamically by applying bandit algorithms. We then compared our results with those obtained using majority voting. As a result, Epsilon-greedy with $\varepsilon = 0.3$ showed the best or second-best prediction performance compared with using only one prediction model and majority voting. Our results showed that bandit algorithms can provide promising outcomes when used in fault prediction.

key words: *Defect prediction, Multi-armed bandit, Diversity of datasets, Dynamic model selection, Risk-based testing*

1. Introduction

Fault prediction is an important activity in software planning and quality control. Previous studies have utilized various techniques, such as feature selection [12], to predict faults in software components. Many studies compared and evaluated various models and prediction techniques across multiple datasets [3][9]. However, in most cases, these models show different performance levels across different dataset (e.g., [3][12]), making their practical use limited. This is because existing datasets can be quite diverse, and therefore, there is a risk that the selected model or technique may perform badly on certain datasets. D'Ambros et al. [3] pointed out that external validity in fault prediction is still an open problem. This is because managers mostly use only one fault prediction model across different projects, which might not share the same characteristics. In addition, if two new fault prediction models have been developed, and they have not been compared, it is not clear which model should be used and when.

To help with the selection of accurate and consistent models, we propose the adoption of bandit algorithms to the model selection process. Although bandit algorithms [16] are 'classical' algorithms that have been around for a long time, they have been recently utilized in various new fields to solve different optimization problems, such as web optimization. We explain bandit algorithms in more detail below.

2. Proposed Solution

2.1 Bandit Algorithms

Bandit algorithms are often explained through an analogy with slot machines. Assume that a player has 100 coins to bet on several slot machines, and the player wants to maximize their reward. Usually, the player might select only one slot machine and bet all 100 coins on that machine. In contrast, bandit algorithms may suggest that the player to bet only one coin on each slot machine to seek the best chances.

Multi-armed bandit problem is to seek sequentially best candidates (they are referred to as **arms**) whose expected rewards are unknown, to maximize total rewards. The following is the simplest bandit algorithm:

- The player bets one coin on one arm.
- **Exploration phase:** when the average reward of the currently selected arm is lower than that of others, the player selects another arm with higher average reward. This action is aimed at finding the arm with the highest reward, and therefore, it is referred to as exploration.
- **Exploitation phase**: when the average reward of the current arm is the highest (or equal), the player keeps playing on the same machine. This action is aimed at exploiting the highest reward arm.

An example of bandit algorithm is shown in Table 1. Initially, the average reward of each arm is set to zero.

1. Arm A is selected randomly. When the reward of arm A is -1, its average reward becomes -1.
2. Arm B is selected because its average reward is higher than that of A. When the reward of arm B is 1, its

†The authors are with Kindai University, Japan.
††The author is with Fukuoka Institute of Technology, Japan.
†††The authors are with Nara Institute of Science and Technology, Japan.
††††The author is with Massey University, New Zealand.
†††††The author is with Wageningen University & Research, the Netherlands.
††††††The author is with Okayama University, Japan.

Table 1. Example of bandit algorithm

# of trial	Selected arm	Reward	Avg. reward A	Avg. reward B
1	A	-1	-1	0
2	B	1	-1	1
3	B	1	-1	1

average reward becomes 1.

3. Arm B is selected again because its average reward is higher than that of A.

The first trial is regarded as the exploitation phase, and the second and third trials are the exploration phase.

The epsilon greedy strategy is a method to choose a random option with probability epsilon. It selects *arms* through the following algorithm.

- The best arm is selected with the probability $1 - \varepsilon\ (0 \leq \varepsilon \leq 1)$ for the exploitation phase based on the average reward of each arm.
- One of the arms is randomly selected with probability ε for the exploration phase, ignoring the average reward.

When the value of ε is 0, arms are always selected based on the average reward of each arm. In contrast, when the value of ε is 1, arms are always selected randomly. As bandit algorithms, we used the epsilon greedy strategy, and set ε as 0, 0.1, 0.2, and 0.3 in the experiment.

2.2 Proposed Solution

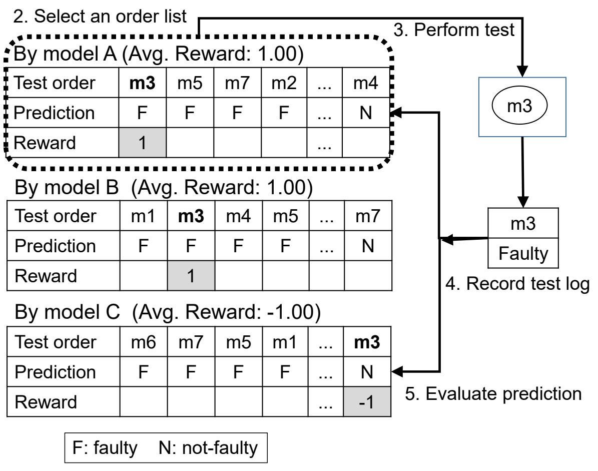
Steps required to utilize bandit algorithms: To apply bandit algorithms to fault prediction models during software testing, the following tasks are required.

1. Building multiple prediction models.
2. Recording testing results.
3. Comparing prediction and testing results.
4. Selecting a model based on the average reward.

Task 1 is performed only once, and task 2 is generally performed during testing - even when we do not apply bandit algorithms. Tasks 3 and 4 are performed automatically using a logging mechanism during testing. While additional tasks may be required, when applying bandit algorithms the risk of using low accuracy prediction models can be suppressed.

Details of proposed solution: In integration testing, except for *big bang testing*, modules are tested in a sequential order. So, the test order (i.e., test priority) of software modules are decided based on the results of fault prediction models. For best results, faults should be detected and removed as early as possible. That is, when modules are predicted as "fault-prone", they should be (re)tested earlier. Our proposed method consists of the following steps:

1. Create test order lists using various prediction models.
2. Select one of the test order lists with the epsilon greedy algorithm.
3. A module is tested based on the order of the list.
4. The test case log is recorded.
5. If the prediction and the test log are found to be the same, the reward of the prediction is set to 1. If they are different, the reward of the prediction is set to -1.
6. Return to step 2.



By model A (Avg. Reward: 1.00)

Test order	**m3**	m5	m7	m2	...	m4
Prediction	F	F	F	F	...	N
Reward	1				...	

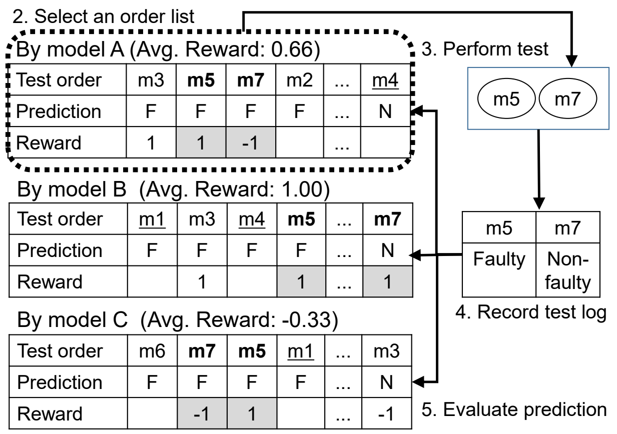
By model B (Avg. Reward: 1.00)

Test order	m1	**m3**	m4	m5	...	m7
Prediction	F	F	F	F	...	N
Reward		1				

By model C (Avg. Reward: -1.00)

Test order	m6	m7	m5	m1	...	**m3**
Prediction	F	F	F	F	...	N
Reward					...	-1

Fig. 1. The first iteration of the proposed method



By model A (Avg. Reward: 0.66)

Test order	m3	**m5**	**m7**	m2	...	m4
Prediction	F	F	F	F	...	N
Reward	1	1	-1		...	

By model B (Avg. Reward: 1.00)

Test order	m1	m3	m4	**m5**	...	**m7**
Prediction	F	F	F	F	...	N
Reward		1		1	...	1

By model C (Avg. Reward: -0.33)

Test order	m6	**m7**	**m5**	m1	...	m3
Prediction	F	F	F	F	...	N
Reward		-1	1		...	-1

Fig. 2. The second and third iterations of the proposed method

In step 1, based on source code metrics of each module such as LOC (lines of codes) and other complexity metrics, each module is predicted as either "fault-prone" or "not-fault prone". Then, modules are sorted by the prediction outcome to create the test order lists. Fig. 1 shows an example of the lists created in step 1. As shown in the figure, there are three lists made by models A, B and C. However, we do not know which list will result in the best accuracy. Our method dynamically creates new orders by selecting one of the lists in steps 2-6. In Figs. 1 and 2, the created new test order includes m3, m5, m7 (shown in bold in the figures), m1, and m4 (shown in underlined). The order of modules m3, m5 and m7 is part of the list created by model A, and m1 and m4 is part of the list by model B.

Fig. 1 shows Steps 2-5 in the first iteration. In the figure, the list created by model A is selected in Step 2 randomly. Module m3 is tested first, since it is predicted as "fault-prone" in the selected list. In Step 3, module m3 is tested, and then faults are found in m3. The test log is recorded in Step 4. In Step 5, we observe that the test log and the prediction of m3

(a) Faults found on/after integration test	F				N	
(b) Faults found on integration test	F		**N**		N	
(c) Prediction	F	N	**F**	**N**	F	N
(e) Reward ((b) = (c)?)	1	-1	**-1**	**1**	-1	1

$\underbrace{\qquad}_{100 - p\%}\ \underbrace{\qquad}_{p\%}$

Fig. 4. The relationship between found faults, prediction, and reward.

included in the list created by A are both the same (i.e., m3 is faulty). Therefore, the reward is set as 1. Similarly, the prediction included in the list created by B and C is evaluated. In Step 6, go back to step 2.

Fig. 2 shows Steps 2-5 during the second and third iteration. In Step 2, based on the average of the reward, the order list created by model A is selected again. As shown in Fig. 2, modules m5 and m7 are tested, since they are predicted as "fault-prone" on the selected list A. In Step 5, compared with the test log, the prediction of m5 included in the list is correct. However, the prediction of m7 is incorrect. Therefore, the rewards in the selected list A are set as 1 and -1, respectively. Similarly, the predictions included in the list created by B and C are evaluated. By comparing the average rewards of the order lists generated by models A, B and C, model B resulted in the highest average.

In the fourth iteration, based on the average value, the order list created by model B is then selected in Step 2. Module m1 (underlined in Fig. 2) is tested first, since it appears at the top of the selected list B. After the test of module m1 (i.e., in the fifth iteration), module m4 (underlined in Fig. 2) is then tested. Although module m3 appears after m1 on the selected list B, m3 has already been tested. So, module m4 is then tested in the fifth iteration.

Steps 2-6 are performed repeatedly during the test execution phase. In contrast, ordinal software testing performs steps 1-4 only once. In step 2, a test order list (i.e., a prediction model) is selected based on some reasoning to enhance the prediction accuracy of models. For example, a list made by logistic regression with the feature selection technique (correlation-based filter-subset technique with the best-first search method) is selected, as the studies of Ghotra et al. [8][9] showed that it outperforms other models on some datasets.

Differences with typical bandit algorithms: There are two major differences between the proposed solution and a typical use of bandit algorithms. Typical bandit algorithms can evaluate only one arm on each trial. In contrast, in software testing, we adopted an approach to evaluate all order lists (i.e., arms) in each iteration (i.e., trial). However, the reward on software testing cannot always be set properly. Not all faults can be detected during integration testing. However, even when faults are not found during integration testing, they could be possibly detected in later testing (i.e., during system testing and after the release of the software). Therefore, as shown in Fig. 4 (text in bold), the probability of errors in rewards is set at *p*% probability (when faults are not detected during integration testing, but found later). In the experiment presented in this paper, we simulated this reward error, setting *p* as 20.

Another possible issue is test flakiness – where test outcomes are nondeterministic. This may lead to either false positives or false negatives. Our approach does not account for such cases.

Differences with ensemble techniques: While both bandit algorithms and ensemble techniques [14] use multiple prediction models, ensemble techniques do not include the exploration phase (in our case, this happens during testing). The prediction results of ensemble techniques are regarded as one of the arms, and therefore, bandit algorithms and ensemble techniques can be used together.

3. Experiment

For the experiment, we used three datasets which were collected as part of the NASA Metrics Data Program. Those datasets are:

- KC4: 125 modules, 61 (48.8%) faulty modules
- MW1: 403 modules, 31 (7.7%) faulty modules
- PC4: 1458 modules. 178 (12.2%) faulty modules

We selected these datasets based on the diversity of the size of each dataset and the ratio of faulty to not-faulty modules. Although those datasets are relatively old, they are still useful and have been used in recent studies, e.g. [9].

To predict fault-prone modules, we built different prediction modules using Decision tree (**DTree**), Neural network (**NN**), Logistic regression (**Logistic**), and Linear discriminant analysis (**LDA**). Those are conventional methods that have been widely used for defect prediction studies, such as [8][9]. Other machine learning approaches such as random forest and support vector machine could also be used. We are considering all those methods in our future work.

Additionally, to compare the effect of our approach with the other existing approach, we used a majority voting algorithm (Voting) [20] based on the above four prediction models. Voting has been used on similar studies on defect prediction (e.g., [11]).

When applying bandit algorithms, prediction results by the prediction models were sorted by module size, in addition to the prediction results (i.e., fault-prone and larger modules are tested first). This is because we assumed that there will be a risk-based testing [6] in place. Note that the indirect effect of modules size should be taken into consideration here before performing any prediction [18].

As explained in section 2.2, not all faults can be found during the integration testing phase. Based on cross-companies' large-scale data [10], about 17% of faults are usually found after integration testing. This means evaluation of prediction results by bandit algorithms are not correct in 17% probability when the modules are faulty. Considering the possibility of faults been overlooked, we randomly reversed signs of the reward settled by the bandit algorithms at 20% probability when the modules are faulty (i.e., the value *p* on Figure 4 was set as 20).

We applied hold-out method to evaluate the prediction accuracy. The dataset was randomly separated into *learning* and *testing* sets, and the ratio of learning to testing sets size is 3:1. We performed the separation 10 times, and to evaluate predictions performance, we calculated the average of evaluation criterion values acquired on the 10 repetition.

We used the area under the curve (AUC) to evaluate the performance of each prediction model. AUC provides a measure of performance across all classification thresholds. To calculate AUC of the bandit algorithms, predictions selected by the algorithms were used. For example, in Fig. 1 and 2, the selected predictions are “m3, m5, m7, m1, and m4 are faulty,” and AUC is calculated based on that.

Prediction values are real numbers in most models. When we applied bandit algorithms, we set the cutoff value as 0.5, and the prediction values were converted into binary values (i.e., faulty or not-faulty). Note that when the predictions are binary values, AUC is the same as the average of a true positive and true negative rates. To align the condition, we also converted the prediction values derived by conventional prediction models into binary values.

4. Results

First, we compared the prediction accuracy of bandit algorithms (Table 2). The table shows AUC values and rank of AUC across the four parameters in each dataset. The average values of AUC and the rank among the three datasets are also shown in the Table 2. The average AUC and average rank of AUC of Epsilon-greedy (ε=0.3) method attained the highest among the three algorithms. Evidently, Epsilon-greedy (ε=0.3) was the algorithm with the highest accuracy across the four parameters.

Next, we compared the prediction accuracy of Epsilon-greedy (ε=0.3) with conventional methods. The prediction accuracy values are shown in Table 3. For the MW1 dataset, the difference in the AUC value between the best model LDA, and Epsilon-greedy was almost the same. For the KC4 dataset, the difference of AUC between the best model (i.e., NN) and Epsilon-greedy was 0.03. For the PC4 dataset, the best model was Epsilon-greedy (ε=0.3). That is, the prediction accuracy of Epsilon-greedy was 0.03 lower than the best model at the worst case. The average AUC of majority voting was the second lowest, and the average rank was the lowest among the prediction methods. Across all three datasets, the average AUC value and rank of Epsilon-greedy was higher than the four prediction methods and majority voting. Therefore, using Epsilon-greedy (ε=0.3), we can obtain better fault prediction performance, without considering which is the best model.

Note that for the PC4 dataset, AUC of bandit algorithms was higher than that of the four prediction models. This is because the prediction results were sorted by module size. When module size was large, bandit algorithms selected the order list by LDA, while selecting the list by NN when the module size was small. The selection affected the prediction accuracy positively.

Table 2. Prediction accuracy of Epsilon-greedy

	AUC			Rank			Avg.	Avg.
	PC4	KC4	MW1	PC4	KC4	MW1	AUC	Rank
ε=0	0.73	0.73	0.56	4	4	4	0.67	4.0
ε=0.1	0.77	0.74	0.57	3	1	3	0.69	2.3
ε=0.2	0.78	0.73	0.58	2	3	2	0.69	2.3
ε=0.3	0.78	0.73	0.60	1	2	1	0.70	1.3

Table 3. Prediction accuracy of conventional models

	AUC			Rank			Avg.	Avg.
	PC4	KC4	MW1	PC4	KC4	MW1	AUC	Rank
DTree	0.70	0.74	0.58	3	2	3	0.67	2.7
LDA	0.69	0.68	0.60	4	6	1	0.66	3.7
Logistic	0.72	0.69	0.57	2	5	4	0.66	3.7
NN	0.61	0.76	0.52	6	1	6	0.63	4.3
Voting	0.65	0.73	0.54	5	4	5	0.64	4.7
ε=0.3	0.78	0.73	0.60	1	3	2	0.70	2.0

5. Related Work

Online learning: Bandit algorithms can be considered as an online learning method [7]. Several online learning methods have been applied to fault prediction in the past (e.g., [17][19]). However, the assumption in the past studies is that fault prediction models should be rebuilt continuously as prediction targets of software systems might change over time. Based on this assumption, it is recommended to build prediction models continuously, using new data which are acquired sequentially during the development (i.e., online). However, in these previous studies, the accuracy of the different prediction models are not taken into considerations in order to optimize the prediction. Our study is different in that we assume the prediction models do not change over time, and based on this assumption, the accuracy of prediction models are taken into consideration.

Wang et al. [19] applied online oversampling and undersampling techniques to software fault prediction in order to address the class imbalance issue in faults datasets. However, the study did not consider the accuracy of prediction models to optimize the prediction.

Tabassum et al. [17] applied online learning to Just-In-Time software defect prediction models in order to compare the outcomes of three different proposed methods. Although the authors used majority voting (simply by counting the majority, and not the weighted majority [13]), their approach aimed at building multiple prediction models repeatedly, based on data points which are sequentially added to the models. That is, they did not consider the accuracy of prediction models to optimize the prediction online. The study [17] assumes that the prediction models are not only used in the testing phase but also in the whole software development process. The study used datasets that were collected from a company from a period of 9-10 months, and other datasets that were collected from open source projects for a period of 6-14 years. Hence, it is natural to assume that the performance of the prediction models will vary over time (given the nature of the different datasets).

Dynamic model selection: Some studies [5][15] focused on the issue that prediction models do not result in best possible performance across different datasets. In these studies, the focus was on the dynamic selection of prediction models from a set of available models. Di Nucci et al. [5] selected prediction models dynamically to predict fault-prone modules where Rathore et al. [15] did the same to predict the number of faults. However, the selection process was based on characteristics of prediction target modules such as code and design metrics, while the proposed methods did not consider the accuracy of prediction models to optimize the prediction. Therefore, their methods might not be suitable to address the issue of the external validity of prediction models.

6. Conclusion and future work

We applied bandit algorithms in order to dynamically select accurate fault prediction models. Bandit algorithms are commonly utilized to help with optimization problems. Previous studies assumed that the accuracy of the prediction models are evaluated before the testing phase (i.e., offline). Hence, external validity in fault prediction was the important problem. In contrast, we empirically show that, with our approach, the accuracy of the models can be evaluated *during* testing phase (i.e., online). To the best of our knowledge, this is the first instance of a bandit algorithm (or similar online learning methods) being applied to the fault prediction problems, and compared the accuracy of the algorithms with various prediction methods.

In this work, we applied Epsilon-greedy bandit algorithm, by changing the parameter ε (i.e., the probability for the exploration). We used four conventional prediction models and majority voting on three datasets. As a result, when we used Epsilon-greedy with $\varepsilon = 0.3$, the prediction accuracy was the best or the second-best on each dataset. The results are promising, as they suggest that bandit algorithms can prevent the selection of low accuracy fault prediction models and achieve better prediction performance than conventional methods and majority voting.

This is an ongoing work. In the future, we plan to evaluate our approach on larger datasets, and use the methods proposed in this work to evaluate the performance of other prediction models across multiple datasets.

Acknowledgments

This research was partially supported by the Japan Society for the Promotion of Science (JSPS) [Grants-in-Aid for Scientific Research (C) (No. 20K11749) and (S) (No. 20H05706)].

References

[1] P. Auer, N. Cesa-Bianchi, Y. Freund and R. Schapire, "Gambling in a rigged casino: The adversarial multi-armed bandit problem," Proc. of Annual Foundations of Computer Science, pp. 322-331, 1995.

[2] S. Bubeck and N. Cesa-Bianchi "Regret Analysis of Stochastic and Nonstochastic Multi-armed Bandit Problems," Foundations and Trends in Machine Learning, vol.5, no.1, pp.1-122, 2012.

[3] M. D'Ambros, M., Lanza, and R. Robbes, "Evaluating defect prediction approaches: a benchmark and an extensive comparison," Empirical Software Engineering, vol.17, no.4-5, pp.531-577, 2012.

[4] V. Dani and T. Hayes, "Robbing the bandit: less regret in online geometric optimization against an adaptive adversary," In Proc. of annual ACM-SIAM symposium on Discrete algorithm (SODA), pp.937–943, 2006.

[5] D. Di Nucci, F. Palomba, R. Oliveto and A. De Lucia, "Dynamic Selection of Classifiers in Bug Prediction: An Adaptive Method," IEEE Transactions on Emerging Topics in Computational Intelligence, vol.1, no.3, pp.202-212, 2017.

[6] M. Felderer and R. Ramler, "Integrating risk-based testing in industrial test processes," Software Quality Journal, vol.22, no.3, pp.543-575, 2014.

[7] Y. Freund and R. Schapire, "A Decision-Theoretic Generalization of On-Line Learning and an Application to Boosting," Journal of Computer and System Sciences, vol.55, no.1, pp.119-139, 1997.

[8] B. Ghotra, S. McIntosh, and A. Hassan, "Revisiting the impact of classification techniques on the performance of defect prediction models," Proc. of International Conference on Software Engineering (ICSE), pp.789-800, 2015.

[9] B. Ghotra, S. McIntosh, and A. Hassan, "A Large-Scale Study of the Impact of Feature Selection Techniques on Defect Classification Models," Proc. of International Conference on Mining Software Repositories (MSR), pp.146-157, 2017.

[10] Information-technology Promotion Agency (IPA), Japan, The 2018-2019 White Paper on Software Development Projects, IPA, 2018 (in Japanese).

[11] T. Khoshgoftaar, P. Rebours, and N. Seliya, "Software quality analysis by combining multiple projects and learners," Software Quality Journal, vol.17, pp.25-49, 2009.

[12] M. Kondo, C. Bezemer, Y. Kamei, A. Hassan, and O. Mizuno, "The impact of feature reduction techniques on defect prediction models," Empirical Software Engineering, vol.24, no.4, pp.1925–1963, 2019.

[13] N. Littlestone, and M. Warmuth, "The Weighted Majority Algorithm," Information and Computation, vol.108, no.2, pp.212-261, 1994.

[14] J. Petrić, D. Bowes, T. Hall, B. Christianson, and N. Baddoo, "Building an Ensemble for Software Defect Prediction Based on Diversity Selection," Proc. of International Symposium on Empirical Software Engineering and Measurement (ESEM), no.46, p.1–10, 2016.

[15] S. Rathore and S. Kumar, "An Approach for the Prediction of Number of Software Faults Based on the Dynamic Selection of Learning Techniques," in IEEE Transactions on Reliability, vol.68, no.1, pp.216-236, 2019.

[16] R. Sutton, and A. Barto, Reinforcement Learning: An Introduction, A Bradford Book, 1998.

[17] S. Tabassum, L. Minku, D. Feng, G. Cabral, and L. Song, "An Investigation of Cross-Project Learning in Online Just-In-Time Software Defect Prediction," Proc of International Conference on Software Engineering (ICSE), 2020.

[18] A. Tahir, K. Bennin, S. MacDonell, and S. Marsland, "Revisiting the size effect in software fault prediction models," Proc. International Symposium on Empirical Software Engineering and Measurement (ESEM), article 23, p.1-10, 2018.

[19] S. Wang, L. Minku, and X. Yao, "Online Class Imbalance Learning and Its Applications in Fault Detection," International Journal of Computational Intelligence and Applications, vol.12, no.4, 2013.

[20] Z. Zhou, Ensemble Methods: Foundations and Algorithms, Chapman and Hall/CRC, 2012.