Regular Paper

Improve Measuring Suspiciousness of Bugs in Spectrum-Based Fault Localization With Deep Learning

Hitoshi Kiryu[†], Nobutoshi Todoroki[‡], Satoshi Suda[‡], Shinpei Ogata*, and Kozo Okano*

[†]Graduate School of Engineering, Shinshu University, Japan
21w2025g@shinshu-u.ac.jp
[‡]Advanced Technology R&D Center of Mitsubishi Electric
{Suda.Satoshi@ay, Todoroki.Nobutoshi@dn}.mitsubishielectric.co.jp
*Faculty of Engineering, Shinshu University, Japan
{ogata, okano}@cs.shinshu-u.ac.jp

Abstract - Localizing Faults is integral for debugging in developing software. Spectrum-Based Fault Localization (SBFL) is a technique to localize faults. SBFL calculates the suspiciousness scores for each line in a source code using execution coverages of tests that represent which lines are executed in which tests. Some studies apply a deep learning techniques to SBFL. In these studies, the suspiciousness scores are calculated by giving a virtual coverage to a trained model. This paper proposes a method to calculate the suspiciousness scores of lines in source code from the execution coverages, and evaluate an effectiveness of a bootstrapping sampling method. The proposed method introduces the virtual coverage that activates consecutive lines whose execution is the same in any execution coverage. The method provides a ranking based on the score. As a result of the evaluation, it is confirmed that the proposed virtual coverage is better than a virtual coverage in previous researches and the sampling method effectively works in training model.

Keywords: SBFL, Fault Localization, Deep Learning

1 INTRODUCTION

When a problem caused by code is found in software development or maintenance, it is necessary to localize and fix bugs. Generally, such a debugging needs a lot of time and human works. Many techniques to localize faults and fix bugs have been studied to support developers in debugging.

One example of a bug-fixing technique is GenProg [1], which outputs code that passes all the test suites with a genetic algorithm. In techniques for localizing bugs, Various studies [2]–[5] have been conducted. These studies propose methods to identify statements that cause bugs by using information such as bug reports, trace information, and visualization.

Another technique for localizing the fault is Spectrum-Based Fault Localization (SBFL). SBFL localizes faults in the source code based on the execution coverage and test results of each test. The technique calculates the suspiciousness of each statement and provides a ranking based on the suspiciousness. The basic idea of SBFL is that a line executed in a failed test is more likely to contain a bug, while a line executed in a successful test is less likely to contain a bug.

As for difficulty of Fault Localization with machine learning, One survey [6] mentions that the lack of large labeled data sets and imbalance of training data makes the fault localization using deep learning more difficult. Especially, the problem of the imbalance of data often appears in fault localization using test cases. This is because a quantity of failed test cases are generally less than that of a quantity of passed test cases. In learning with imbalanced data, minority class can be ignored in prediction. For example, If A class occupies 99 percent of data, a neural network can get more than 99 percent of precision by just classifying inputs to the class.

In this paper, we propose a method based on SBFL techniques and deep learning techniques to support developers in debugging fault in source code. The method we propose also address handling imbalanced data. The proposed method shall localize a single fault in source code.

1.1 Related Work

Deep learning is a technology that has been showing results in a wide variety of fields, including image recognition and natural language processing. The field of fault localization is no exception either. Ikeda et al. [7] proposed the method that localizes a fault with a neural network. They trained the network to predict test results from execution traces and used ablated traces to localize a fault method.

Deep learning technique for NLP is utilized for fault localization. Guo et al. [8] proposed GraphCodeBERT that is a pre-trained model for programming language that considers the inherent structure of code. The model is based on BERT [9], and pre-trained in some tasks including Masked Language Modeling. Source code, text data including comments and data flow graph that represents variable relation are used as explanatory variables.

In the field of SBFL, the study [10] has been conducted to compute the suspiciousness using deep learning. In this study, three architectures, RNN (Recurrent Neural Network), CNN (Convolutional Neural Network), and MLP (MultiLayer Perceptron), are utilized to compute the suspiciousness of bugs, and CNN shows the best results. In another study [11], which calculates the suspiciousness using RBF networks (Radial Basis Function Network), concluded that RBF networks are more effective than existing methods such as Ochiai in fault localization. SBFL techniques with deep learning use execution coverage data that often consist of few failed tests and a lot of passed tests as explanatory variables to predict test results. Such an imbalanced data occurs deterioration of a performance of the deep learning model. This is because the model have to predict failed test, namely, the minority of data. Zhang et al. [12] proposes a SBFL method using deep learning technique to deal with data imbalance. The SBFL method applies upsampling technique that increase quantity of minority data in neural network training. The experiment shows the method performed best with an upsampled data that consist of failed and passed tests in the same ratio. These studies that address SBFL using deep learning train a neural network model to predict test results, and utilize a virtual coverage that is one certain line is executed to calculate the suspiciousness scores from the trained model.

1.2 The Approach

This paper proposes a method to compute the suspiciousness of lines in the source code from the program spectrum with deep learning. Using a trained model to predict test results from the execution coverages, suspiciousness scores are measured for each line. In training model process, a bootstrapping sampling method is utilized to deal with imbalanced data that composed with a lot of passed tests and a few failed tests. We propose a virtual coverage that differs to a virtual coverage in previous researches about SBFL with deep learning techniques. In the proposed virtual coverage, one certain block of lines that are commonly executed across coverage is executed rather than one certain line. The proposed virtual coverage is input to the trained model in order to measure the impact of each line on bug prediction in the trained model. We treat the impact as suspiciousness scores. A ranking of lines that are likely to cause bugs based on the suspiciousness scores are provided.

In evaluation experiment, the results indicates the proposed virtual coverage is better than the virtual coverage in previous researches, and bootstrapping sampling method contributes improving performance of fault localization. From the experiment, We conclude that the proposed virtual coverage and the sampling method effectively work in fault localization.

In the following sections, Section 2 describes the related techniques for this research. Section 3, explain about the proposed method. Section 4 shows the results in the evaluation experiments and Section 5 discuss the results in the experiments. Finally, we conclude in Section 6.

2 PRELIMINARIES

This section explains techniques, which compose the proposed method.

2.1 Spectrum-Based Fault Localization (SBFL)

SBFL is a technique to localize faults in source code by calculating suspiciousness scores that represent probability of causing bugs for every statement. Generally, a ranking based

```
     def fizzbuzz(i):
S1     if i % 15 == 0:
S2       return "FizzBuzz"
S3     elif i % 3 == 0:
S4       return "Fizz"
S5     elif i % 4 == 0:
      # correct condition is “i % 5 == 0”
S6       return "Buzz"
S7     return i
```

	i	s_1	s_2	s_3	s_4	s_5	s_6	s_7	test result
test1	15	1	1	0	0	0	0	0	0
test2	3	1	0	1	1	0	0	0	0
test3	4	1	0	1	0	1	1	0	1
test4	5	1	0	1	0	1	0	1	1
Ochiai		0.707	0.0	0.816	0.0	1.0	0.707	0.707	

Figure 1: Example of SBFL

Table 1: Four Values for Calculation of Suspiciousness

e_f	Number of failed tests that execute the program element.
e_p	Number of passed tests that execute the program element.
n_f	Number of failed tests that do not execute the program element.
n_p	Number of passed tests that do not execute the program element.

on the scores are built to support developers to find the faults out.

For example, Ochiai [13] and Dstar [14] are metrics to calculate the suspiciousness scores. Here, 'N' is the exponent variable of e_f. The score tends to be higher if a line is executed more frequently when the test fails or less frequently when the test succeeds.

$$Ochiai = \frac{e_f}{\sqrt{(e_f + n_f)(e_f + e_p)}} \qquad \text{(Ochiai)}$$

$$Dstar(* = N) = \frac{{e_f}^N}{e_p + n_f} \qquad \text{(Dstar)}$$

The metrics for calculating the suspiciousness of a bug are based on four values described in Table 1.

Figure 1 shows an example of SBFL. The source code in the figure is a function that returns the result of FizzBuzz for a given number i as input. The fifth line of the function contains a bug. This is because the conditional statement is incorrect. The suspiciousness scores of each line is calculated by Ochiai based on the results and the coverage of the tests. The suspiciousness is the highest on lines fifth, which are executed only when the test fails. This indicates that the suspiciousness of the buggy lines is calculated properly. Thus,

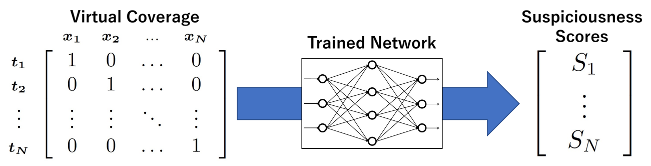

Figure 2: Overview of Score Calculation in The Previous Researches

from the test results and the coverage, it is possible to identify the location of bugs. Note that in the example shown in the figure, the score of S7 will get closer to that of S5 when the test cases which input is a multiple of 5 and not a multiple of 3 increase. Furthermore, the score of S3 will decrease when test cases which execute S3 and result pass increase. SBFL focuses on the frequency of test failure in execution of line.

2.2 SBFL using Deep Learning Techniques

Some studies [10]–[12] focus on SBFL that utilize deep learning techniques. In SBFL using deep learning, a virtual coverage is input to a trained neural network model to calculate suspiciousness scores. The model is trained to predict test results from execution coverages. Figure 2 shows overview of a virtual coverage. The virtual coverage in the figure is a execution coverage based on a scenario that only one certain line is executed in a virtual test. In order to measure the contribution that the execution of statement affect to the result prediction, the virtual coverage is input to the trained model. It is considered that the output of the virtual coverage indicates the impact of the statement on the test results. Therefore, the output of virtual coverage is treated as suspiciousness scores.

3 PROPOSED METHOD

This section describes the methodology of SBFL using a Neural Network in detail.

In the proposed method, first, a neural network model learns about a relation between the execution coverage of lines in the source code for each test and the test result. The model takes the coverage as input, and output respectively probability of a test result, pass and fail. In order to measure the impact on decision-making about test results in the trained model, a virtual coverage is input to trained model. The impact on decision-making is considered as the suspiciousness of bugs. The trained model cannot be applied to other source code. This is because the model learns only the relation between a execution of lines in a certain source code and test results.

3.1 Training Network

Figure 3 shows the overview of the training step. The neural network model learns implicit relations between the execution coverages and test results to predict test results.

Generally, the execution coverages collected by tests are composed with a lot of passed test and a few failed tests, namely imbalanced data. A model trained with such imbalanced data tend to ignore the minority data. In SBFL, the model have to emphasize minority data that is a few failed tests. Therefore, the model have to deal with imbalanced data in learning.

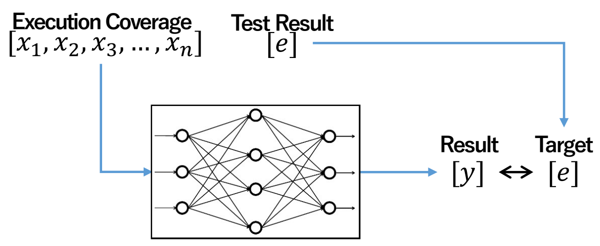

Figure 3: Overview of Training Model

In order to handle the imbalanced data, we utilize the bootstrapping sampling method [15] that Yan et al. proposed. In the bootstrapping sampling method, the ratio of minority to majority is set to 1:1 for each mini-batch. Data of the majority group is divided into N pieces and create tentative mini-batches. For each tentative mini-batches, the same quantity of the mini-batch is randomly extracted from the minority group and joined with the tentative mini-batch as a mini-batch.

3.2 Virtual Coverage

We introduce a block of code for a virtual coverage. Source code is segmented into blocks. The block is defined as: For any execution coverage C and consecutive lines $\forall S_i, S_{i+1} \in C$, S_i and S_{i+1} are in the same block if $S_i = S_{i+1}$. Figure 4 shows the example of block segmentation for statements S_n in the example coverages of the tests t_n. The check marks in the Figure means the statement is executed in the test.

The matrix of Virtual Coverage shows the example virtual coverage of the blocks shown in Fig. 4. In the matrix, 1 means that the line is executed and 0 means not executed. Therefore, lines contained within the block are only executed. As shown in Fig. 2, the virtual coverage in previous research is executed by line. The proposed virtual coverage is executed by block.

Virtual Coverage

$$\begin{array}{c|cccccccccc|}
 & S_1 & S_2 & S_3 & S_4 & S_5 & S_6 & S_7 & S_8 & S_9 & S_{10} \\
block_1 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
block_2 & 0 & 1 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
block_3 & 0 & 0 & 0 & 1 & 1 & 0 & 0 & 0 & 0 & 0 \\
block_4 & 0 & 0 & 0 & 0 & 0 & 1 & 1 & 0 & 0 & 0 \\
block_5 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 \\
block_6 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 1
\end{array}$$

We apply the block segmentation for lines in source code. In the virtual coverage, one certain block, i.e. lines in the same block, are executed. The neural network model learns to focus on patterns that appear in the data and predict the results. Therefore, by inputting the patterns that appear in data as the virtual coverage, the contribution of each pattern to the test results can be effectively extracted. This is the reason why we segmented the source code into the blocks.

Suspiciousness scores are given to each blocks. Hence, lines in the same block have same suspiciousness scores each other. Since lines in the same block have the same execution pattern, SBFL have a limit that difficulty of distinguishing

Table 2: Bugs for The Experiment

Project	Description	Bug Versions	LOC(Lines Of Code)	Tests
Chart	JFreeChart	8	5798	711
Math	Apache Commons Math	24	20803	6265
Lang	Apache Commons Lang	7	13901	677

	t_1	t_2	t_3	t_4	Block Divisions
s_1	✓		✓	✓	1
s_2	✓	✓	✓	✓	2
s_3	✓	✓	✓	✓	
s_4		✓	✓		3
s_5		✓	✓		
s_6	✓			✓	4
s_7	✓			✓	
s_8			✓		5
s_9	✓	✓	✓		6
s_{10}	✓	✓	✓		

Figure 4: Example of Block Segmentation

such lines from each other. The proposed virtual coverage gives blocks suspiciousness scores followingly the limit.

As for granularity of block segmentation, for example, an exception handling can interrupt the execution in middle of a code block. The block segmentation can be more smaller segmentation than code block. Hence, the proposed method can localize faults with the same or smaller granularity than that of code block.

3.3 Extract Suspiciousness of a Bug

The virtual coverage is input to the trained model to caclulate the suspiciousness scores. We consider the output means impact of patterns of lines on results prediction. In other word, the value represents probability of causing bugs. Thus, We treat the values as suspiciousness scores.

The suspiciousness scores are ranked in descending order and output as a result of fault localization. The ranking presents lines and the corresponding suspiciousness scores. As example of Fig. 4, If the second block get the highest suspiciousness score, The ranking has two lines ranked first in the suspiciousness score. Lines that are not executed in failed tests are excluded from the ranking, because they are not likely to contain bugs.

4 EVALUATION EXPERIMENT

In order to evaluate the proposed method, we applied the proposed method to actual bugs in some OSS projects. We collect bugs from Defects4J [16] that is a database and extensible framework providing real bugs to enable reproducible studies in software testing research. We applied the proposed method for bugs in three projects, that is Chart, Math, and Lang. Defects4J has real bug source code and fix patches for each bug. We collected bugs whose fix patch satisfies the following conditions.

- The fix patch has not only add changes : If the patch has only add changes, the original buggy source code have no lines to be fixed.
- Changes in the fix patch spread across multiple chunks : The proposed method aims localizing a single fault, Thus we don't collect such bugs.

The proposed method is applied to the coverage collected from the above three projects. The bugs we collected are shown in Table 2. The table shows bug versions of each project and entire tests and entire LOC(Lines Of Code) of the versions.

Following two types of virtual coverage are compare to evaluate which method gives higher suspiciousness scores to bugs:

- Block (proposed Virtual coverage) : lines in the same block is executed.
- One-Hot (Virtual coverage in the previous researches) : only a certain line is executed.

We collect four rankings based on the scores calculated following four methods:

- One-Hot
- Block
- One-Hot with Bootstrapping Sampling
- Block with Bootstrapping Sampling

The average processing time in the experiment for 79 buggy source code shown in Table 2 was 24.33 seconds. The experiment is conducted with following environment:

- OS : Windows 11 Pro
- CPU: Intel i5-9400F
- Memory: 16GB
- GPU: GTX1660

The cumulative sum charts in the Fig. 5, 6, 7, 8 show the comparison of the experiment results. The charts describe how many y percent of faults are ranked in top x percent. Therefore, when a chart is above another chart, the method of the chart can rank faults more higher.

4.1 Compare of The One-Hot and The Block

Figure 5 shows the comparison of the One-Hot and the Block. The chart of Block is always above one of the One-Hot. In Fig. 6, The chart of Block with bootstrapping sampling is always above one of the One-Hot with bootstrapping sampling.

From the two comparison, The curves of Block always exceed the curves of One-Hot regardless of whether bootstrapping sampling is used.

4.2 Compare of The Effectiveness of Bootstrapping Sampling

Figure 7 shows the comparison of One-Hot and One-Hot with bootstrapping sampling. The chart of One-Hot with bootstrapping sampling is always above or same as another. In Fig. 8, The chart of Block with bootstrapping sampling is mostly above or same as another except 35% of the statements.

From the two comparison, The curves of bootstrapping sampling mostly exceed the curves of method without bootstrapping sampling.

4.3 Statistical Hypothesis Test for TopN%

In order to compare the result of each method, we conducted statistical hypothesis test to TopN% values of each test. Table 3 describes p-values of the Wilcoxon signed-rank test for each combination of the methods. Wilcoxon signed-rank test is a non-parametric statistical hypothesis test for paired data to compare the locations of two populations. In the test, The null hypothesis is that the locations of two populations from paired data are not different significantly, and the alternative hypothesis is that the locations are different significantly. In the right-tailed test for method A vs method B, the alternative hypothesis is that the location of population of method A is greater than the one of B, and in the left-tailed test, the alternative hypothesis is that the location of population of method A is less than the one of B.

If a p-value of a test is greater than a given significance level, the hypothesis of the test is accepted. The hypothesis of right-tailed test means the method A tend to give the statements greater values as rank than the method B, in other word, the method B gives statements higher rank than the method A. This means the method B is more effective to localizing faults.

In order to counteract the multiple comparisons problem, we utilize a significance level corrected by the Bonferroni correction in each testing. the Bonferroni correction tests each individual hypothesis at significance level $\frac{\alpha}{m}$, where α is desired overall significance level and m is the number of hypotheses. This allows the probability of type I error of m testings to be less than or equal to α. In this paper, we test 4 hypotheses at $\alpha = 0.05$, hence, significance level of each test is set to 0.0125.

5 DISCUSSION

As results of Section 4.1, the method using the proposed virtual coverage always above another curve in both results. The results say the proposed virtual coverage gave more lot of bugs higher ranks. This means the proposed virtual coverage can extract the impact on result prediction in trained model. Furthermore, the top two items “One-Hot vs Block,” “One-Hot with bootstrapping sampling vs Block with bootstrapping sampling” in Table 3 in Section 4.3 also suggest the rank of buggy line given by method using the block tend to be higher than another one. These results comfirmed that the virtual coverage based on the block segmentation works more effectively in localizing fault compared with the virtual coverage based on one-hot. The lines in the same block is the pattern that the executions of each line are same. Therefore, the virtual coverage based on the pattern can activate the pattern that trained model learned, and enhance the suspiciousness score of a pattern containing buggy lines.

About results of bootstrapping sampling in Section 4.2, the methods using bootstrapping sampling is mostly same or above to the others that don’t use the sampling method. In another result in Section 4.3, the bottom two items “One-Hot vs One-Hot with bootstrapping sampling,” “Block vs Block with bootstrapping sampling” in Table 3 also suggest the sampling method contribute training of model. The bootstrapping sampling is originally proposed to deal with imbalanced data for classification of movies. These results say the sampling method can prevent the model to excessively emphasize the minority in imbalanced data. Therefore, the results indicate the bootstrapping sampling method is effective in Spectrum-Based Fault Localization.

In order to verify difference of the localization among three projects in the evaluation experiment, Kruskal-Wallis test is conducted. Kruskal-Wallis test is a non-parametric statistical hypothesis test to verify the significant difference between the medians of more than three data. As a result of the test, p-value is 0.788. The value exceeds 0.05, generally used as the significance level, by a wide margin, and the alternative hypothesis is rejected. In other words, no significant difference among the projects. In addition, no significant structual features in source code (e.g., wrong variable reference, wrong if statement) in higher ranked faults are found. Since the execution coverages don’t have context of the source code, such features seem to hardly give a feature to execution coverages.

As for the conditions for the proposeed method, there are several points to be considered. First, on the premise that The

Table 3: P-Value of Wilcoxon Signed-Rank Test in Each Combination

Combination	One-Tailed(Left)	Two-Tailed	One-Tailed(Right)
One-Hot vs Block	1	1.711E-5	8.556E-6
One-Hot with bootstrapping sampling vs Block with bootstrapping sampling	9.91E-1	1.881E-2	9.403E-3
One-Hot vs One-Hot with bootstrapping sampling	9.923E-1	1.607E-2	8.037E-3
Block vs Block with bootstrapping sampling	9.537E-1	9.536E-2	4.768E-2

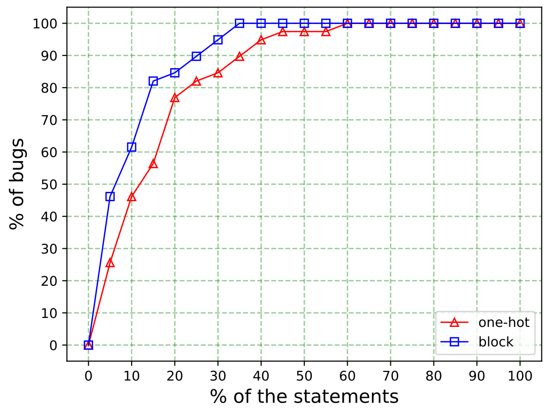

Figure 5: One-Hot Versus Block

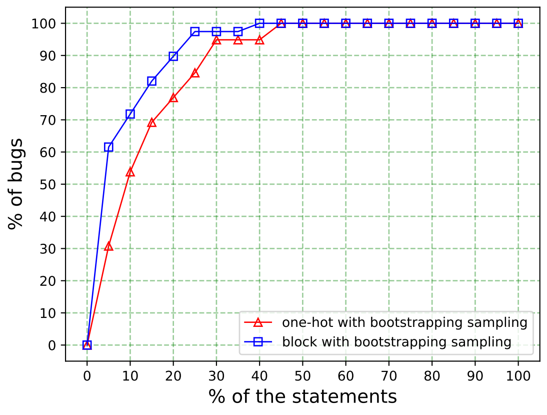

Figure 6: One-Hot with Bootstrapping Sampling Versus Block with Bootstrapping Sampling

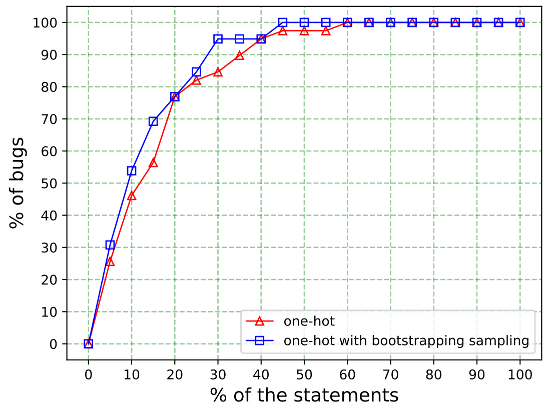

Figure 7: One-Hot Versus One-Hot with Bootstrapping Sampling

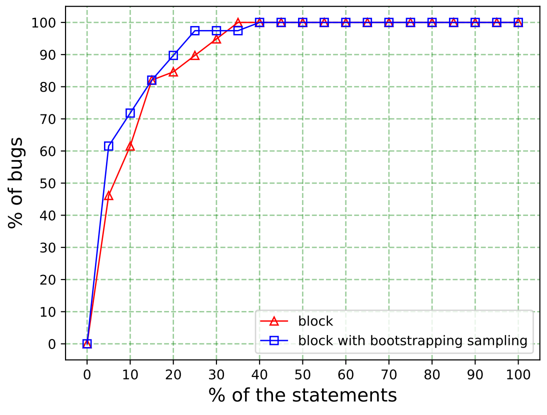

Figure 8: Block Versus Block with Bootstrapping Sampling

execution coverage is line-by-line in the proposed method, projects written in Java is used in the evaluation experiment. It is considered that proposed method can also work in a project written in a programming language which can measure execution coverages by line. Second, a test suite has to be composed with sufficient test cases. The proposed method extracts suspiciousness scores from a trained model. The model cannot learn well from insufficient test cases. Hence, a test suite which has adequate test cases is essential for the model training. Finally, white-box testing may be appropriate test for the proposed in terms of a granularity of localizing faults. The block segmentation in the proposed method divides lines whose execution are same into same blocks. A test suit that branch coverage is 100% makes the block segmentation minimum granularity, and the segmentation is similar to the code block. Therefore, testing methods such as white-box testing, which can cover branches in the source code, allow us to localize faults at smaller granularity.

In the OSS projects on the evaluation experiment, branch coverages are 62.33%, 82.72%, and 88.57% respectively correspond to Chart, Math, and Lang. Branch coverages of Math and Lang exceed 80%, while 62.33% of Chart is not desirable as branch coverage. For test suites whose branch coverage is not desirable, test generation techniques based on static code analysis can emphasise the branch coverage.

The proposed method gives each block suspiciousness scores. Thus, suspiciousness scores of lines in the same block are not different to each line. This means the proposed method cannot distinguish lines that is same pattern in the coverage. This is similar to other SBFL technique, which calculates the probability of bugs based on the coverage. Suspiciousness scores of lines that is identically executed tend to hardly differ each other. This results in a problem that impacts on bugs for such lines is determined equally. Therefore, it is necessary to distinguish such lines that implicit information that doesn't appear in execution coverage such as AST of source code and extracted semantics from documents, etc.

6 CONCLUSION

This paper proposes a Spectrum-Based Fault Localization method with deep learning technique, and the proposed method utilizes a virtual coverage to calculate suspiciousness scores and bootstrapping sampling. The proposed method trains a neural network model that predicts test results based on execution coverages, and measures the impact of each line on bugs from the trained model as suspiciousness scores, and provides a ranking based on the suspiciousness scores. We introduce a block segmentation for the virtual coverage. The block contains consecutive lines whose execution is the same in any execution coverage. In the training model, the bootstrapping sampling generates minibatches that balanced minority data and majority data.

In the evaluation experiments, we obtained results that shows the methods using virtual coverage, the block segmentation, and bootstrapping sampling ranked buggy lines higher rank. The authors concluded that the block segmentation and bootstrapping sampling effectively work in fault localization.

In future work, we are going to utilize implicit information that doesn't appear in execution coverage. The proposed method cannot distinguish lines that is same pattern in the coverage. This is because a execution coverage is a form which dispose of context of source code. Hence, the method have to make such lines different by implicit semantics of source code.

ACKNOWLEDGEMENT

Part of this work is supported by fund from Mitsubishi Electric Corp.

The research is also being partially conducted as Grant-in-Aid for Scientific Research A (18H04094) and C (21K11826).

REFERENCES

[1] C. Le Goues, M. Dewey-Vogt, S. Forrest, and W. Weimer, "A Systematic Study of Automated Program Repair: Fixing 55 out of 105 Bugs for $8 Each," ICSE, pp.3-13 (2012).

[2] J. Nam, S. Wang, Y. Xi, and L. Tan, "A bug finder refined by a large set of open-source projects," Information and Software Technology, Vol.112, pp.164–175 (2019).

[3] S. Kim, T. Zimmermann, K. Pan, and E. James Jr. Whitehead, "Automatic Identification of Bug-Introducing Changes," 21st IEEE/ACM International Conference on Automated Software Engineering (ASE'06), pp.81-90 (2006).

[4] S. Tsakiltsidis, A. Miranskyy, and E. Mazzawi, "Towards Automated Performance Bug Identification in Python," 2016 IEEE International Symposium on Software Reliability Engineering Workshops (ISSREW), pp.132-139 (2016).

[5] K. Matsushita, M. Matsumoto, K. Ohno, T. Sasaki, T. Kondo, and H. Nakashima, "A Debugging Method Based on Comparison of Execution Trace," Symposium on Advanced Computing Systems and Infrastructures (SACSIS), Vol.2011, pp.152-159 (2011) (in Japanese).

[6] A. Elena, B. Alexander, D. Artem, K. Konstantin, K. Anton, M. Ilya, and M. Vladimir, "A Survey on Software Defect Prediction Using Deep Learning," Mathematics, Vol.9, No.11, 1180 (2021).

[7] T. Ikeda, K. Okano, S. Ogata, and S. Nakajima, "Localization of Fault Methods and Ablation of Execution Traces Using A Machine Learning Model to Classify Test Results," IEICE Technical Report, Vol.121, No.416, pp.13-18 (2022) (in Japanese).

[8] D. Guo, S. Ren, S. Lu, Z. Feng, D. Tang, S. Liu, L. Zhou, N. Duan, A. Svyatkovskiy, S. Fu, M. Tufano, S. Kun Deng, C. Clement, D. Drain, N. Sundaresan, J. Yin, D. Jiang, and M. Zhou, " GraphCodeBERT: Pre-training Code Representations with Data Flow," arXiv:cs.SE/2009.08366. (2021).

[9] J. Devlin, M. Chang, K. Lee, and K. Toutanova, " BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding," Proceedings of the 2019 Conference of the North American Chapter of the Association

for Computational Linguistics: Human Language Technologies, Vol.1, pp.4141-4186 (2019).
[10] Z. Zhang, Y. Lei, X. Mao, M. Yan, L. Xu, and X. Zhang, "A study of effectiveness of deep learning in locating real faults," Information and Software Technology, Vol.131, No.1, pp.1–16 (2021).
[11] W. Eric Wong, V. Debroy, R. Golden, X. Xu, and B. Thuraisingham, "Effective Software Fault Localization Using an RBF Neural Network," in IEEE Transactions on Reliability, Vol.61, No.1, pp.149-169 (2012).
[12] Z. Zhang, Y. Lei, X. Mao, M. Yan, L. Xu, and J. Wen, "Improving deep-learning-based fault localization with resampling," Journal of Software: Evolution and Process, Vol.33, No.3 (2021).
[13] R. Abreu, P. Zoeteweij, and A. J. C. van Gemund, "On the accuracy of spectrum-based fault localization," Testing: Academic and Industrial Conference Practice and Research Techniques, pp.89-98 (2007).
[14] W. Eric Wong, V. Debroy, Y. Li, and R. Gao, "The DStar method for effective software fault localization," IEEE Transactions on Reliability, Vol.63, No.1, pp.290-308 (2014).
[15] Y. Yan, M. Chen, M. Shyu, and S. Chen, "Deep Learning for Imbalanced Multimedia Data Classification," 2015 IEEE International Symposium on Multimedia (ISM), pp.483-488 (2015).
[16] R. Just, D. Jalali, and M. D. Ernst, "Defects4J: A database of existing faults to enable controlled testing studies for java programs," in Proceedings of the 2014 International Symposium on Software Testing and Analysis, pp.437–440 (2014).

(Received: November 6, 2022)
(Accepted: April 5, 2023)

Hitoshi Kiryu is a graduate student of Shinshu University. His areas of interest include formal verification.

Nobutoshi Todoroki received his M.E. degrees in Information and Computer Sciences from Osaka University in 2001. He joined Mitsubishi Electric Corp. Currently he is a senior manager of Solution Engineering Dept. at Advanced Technology R&D Center. He is also a member of IPSJ.

Satoshi Suda received his M.S. degree in mathematics from Osaka University, Osaka, Japan, in 2016. He joined Mitsubishi Electric Corp.Currently he is a researcher of Solution Engineering Dept. at Advanced Technology R&D Center.

Shinpei Ogata is an Associate Professor at Shinshu University, Japan. He received his BE, ME, and PhD from Shibaura Institute of Technology in 2007, 2009, and 2012 respectively. From 2012 to 2020, he was an Assistant Professor, and since 2020, he has been an Associate Professor, in Shinshu University. He is a member of IEEE, ACM, IEICE, IPSJ, and JSSST. His current research interests include model-driven engineering for information system development.

Kozo Okano received his BE, ME, and PhD degrees in Information and Computer Sciences from Osaka University in 1990, 1992, and 1995, respectively. He was an Assistant Professor and an Associate Professor of Osaka University. In 2002 and 2003, he was a visiting researcher at the Department of Computer Science of the University of Kent in Canterbury, and a visiting lecturer at the School of Computer Science of the University of Birmingham, respectively. Since 2020, he has been a Professor at the Department of Electrical and Computer Engineering, Shinshu University. Since 2023, he has been the Director of Center for Data Science and Artificial Intelligence. His current research interests include formal methods for software and information system design, and applying deep learning to Software Engineering. He is a member of IEEE, IEICE, and IPSJ.