IEICE TRANS. INF. & SYST., VOL.E107–D, NO.2 FEBRUARY 2024

LETTER

A Data Augmentation Method for Fault Localization with Fault Propagation Context and VAE

Zhuo ZHANG[†], **Donghui LI**[†a)], **Lei XIA**[††b)], **Ya LI**[†††], ***Nonmembers***, ***and*** **Xiankai MENG**[††††], ***Member***

SUMMARY With the growing complexity and scale of software, detecting and repairing errant behaviors at an early stage are critical to reduce the cost of software development. In the practice of fault localization, a typical process usually includes three steps: execution of input domain test cases, construction of model domain test vectors and suspiciousness evaluation. The effectiveness of model domain test vectors is significant for locating the faulty code. However, test vectors with failing labels usually account for a small portion, which inevitably degrades the effectiveness of fault localization. In this paper, we propose a data augmentation method PVaug by using fault propagation context and variational autoencoder (VAE). Our empirical results on 14 programs illustrate that PVaug has promoted the effectiveness of fault localization.

key words: *debugging, fault localization, VAE, fault propagation*

1. Introduction

Debugging has been recognized as one of the most important processes for software development. It usually needs much manual involvement and costs. Researchers have proposed various fault localization techniques to help increase developers' productivity and reduce the development cost. Among these techniques, spectrum-based fault localization (SBFL) [1] and deep-learning-based fault localization (DLFL) [2] are two of the most popular ones and provide tremendous improvement in accuracy [3]. These two types of fault localization methods are based on an information model which is a matrix constructed by dynamic coverage information from test executions and reflects whether a statement is covered by a test case. In the information model, each line could be recognized as a model-domain test vector which corresponds to an input-domain test case. An element in the vector is 1 denotes the corresponding statement is executed by the test case while 0 means otherwise. Fault localization techniques utilize these vectors from the information model to calculate the suspicious values of statements and rank them in descending order as a result.

Manuscript received August 9, 2023.
Manuscript revised October 4, 2023.
Manuscript publicized October 25, 2023.

[†]The authors are with the School of Electrical and Information Engineering, Tianjin University, Tianjin 30072, China.
[††]The author is with the No.83 Army Joint and Truma Disease Treatment Centre of PLA, Xinxiang 453000, China.
[†††]The author is with the Ningbo Artificial Intelligence Institute, Shanghai Jiaotong University, Ningbo, 315000, China.
[††††]The author is with the College of Computer and Information Engineering, Polytechnic University, Shanghai 200127, China.
a) E-mail: lidonghui@tju.edu.cn (Corresponding author)
b) E-mail: drxialei154371@163.com (Corresponding author)
DOI: 10.1587/transinf.2023EDL8052

The computation process of fault localization illustrates that model domain test vectors are significant for seeking out the faulty code. These vectors usually include vectors with passing labels and vectors with failing labels. Fault localization techniques heavily rely on enough number of these two types of vectors to conduct the calculation process with either statistical correlation coefficients [1] or deep learning models [2]. However, in the process of software development, the number of passing vectors is far more than the number of failing ones. This imbalance phenomenon adversely affects the effectiveness of fault localization. As a matter of fact, many studies have deeply investigated the correlation between model domain test vectors and fault localization result. They illustrated that the asymmetry of these vectors, especially the fact that passing vectors outnumber failing ones inevitably leads to negative impact. There are existing studies [4], [5] proposed try to reproduce a given failure using either symbolic execution methods or search-based methods. However, they only focus on generating test cases from input-domain, which is difficult to generate enough number of failing test cases and achieve a balanced data set. The reason lies in the fact that generating input-domain failing test cases needs real inputs that lead to program failures, which account for quite a small part of the input domain. Searching for these sporadic or even random inputs from the distribution of input domain is not an easy task. Meanwhile, these vectors could not reflect the propagation relationships between suspicious statements, which could facilitate understanding of programs and help analyze the root causes of a failure. Therefore, it is necessary to generate more vectors with failing labels according to fault propagation contexts.

In view of this, we investigate more on how to generate useful vectors according to fault propagation contexts and propose a data augmentation method PVaug to benefit fault localization. Specifically, we attempt to build fault propagation contexts that reflect the information of related statements that have correlation with the failure output and then utilize variational autoencoder (VAE) to generate new failing vectors based on the contexts. The aim is to obtain a class-balanced model domain test vectors, so as to improve the fault localization accuracy. PVaug does not need to strenuously execute and seek out a real failing path, thus the generated failing test vectors do not correspond to real failing paths. In order to verify the effectiveness of PVaug to existing fault localization techniques, we design and perform a large-scale experimental study. We choose 14 large real-world programs and 11 state-of-the-art fault localization

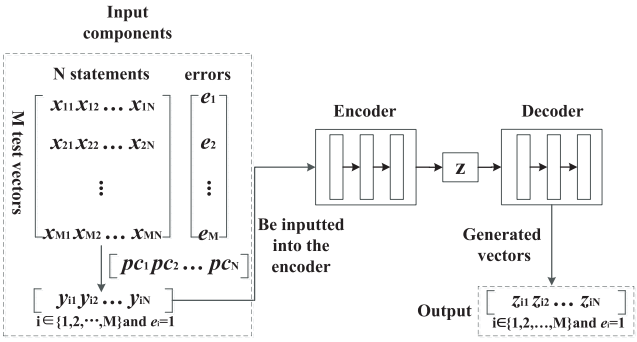

Fig. 1 The architecture of PVaug.

techniques to conduct a comparison. The experimental results verify the effectiveness of our proposed method PVaug.

2. Approach

2.1 Overview

The basic idea of PVaug is to build a fault propagation context showing correlations between execution of statements and test results, and then adopt VAE to build an augmented matrix as the input samples, and use calculation algorithms to quantify the suspicious values of statements being faulty. The augmented matrix is a set of class-balanced model-domain test vectors. Over the past several years, VAE has emerged as one of the most popular approaches to unsupervised learning of complicated distributions [6]. VAE has strong distribution learning capability and has getting promising results on many kinds of complicated tasks. In this study, we propose a new method PVaug, which constructs a fault propagation context and exploits the generation ability of VAE to synthesize new vectors with failing labels, so as to obtain a set of class-balanced model-domain test vectors to promote the efficiency of fault localization. In the generated failing vectors, the value of each element in the fault propagation context is between 0 and 1, the value of each element that is not in the context is 0. During the calculation process of suspicious values, fault localization techniques utilize all the model-domain test vectors including the newly generated ones as input. At this time, the new matrix is a set of class-balanced test vectors, which would benefit fault localization.

Figure 1 shows the architecture of PVaug. There are four main components in PVaug: input components, an encoder, a decoder and the output. Suppose there is a program P that has N statements. The test suite is T that owns M input-domain test cases. After executing program P under T, we could obtain the statement coverage information and check the output against the test oracle. With this information, PVaug could define the model-domain test vectors, which form a $M\times(N+1)$ matrix. The left corner in Fig. 1 illustrates the input components, which include a $M\times(N+1)$ matrix, a vector $pc_1, pc_2, \ldots, pc_N$ and the input data set $y_{i1}, y_{i2}, \ldots, y_{iN}$. The $M\times(N+1)$ matrix represents coverage information of N statements and test results of M input domain test cases. Dynamic slicing is a computation process for reducing the amount of information that needs to be absorbed [7]. Dynamic slice is the computation result of dynamic slicing, it could reflect data and/or control dependencies of program statements and is a subset of statements whose execution affects the output. PVaug chooses a failing test case to conduct the slicing process and get the fault propagation context. We use a vector $pc_1, pc_2, \ldots, pc_N$ to represent this context, in which $pc_i = 1$ means statement S_i is in the fault propagation context and 0 otherwise. We then intersect this vector with the original failing test vectors (*i.e.* $x_{i1}, x_{i2}, \ldots, x_{iN}$, $i \in \{1,2,\ldots, M\}$ and e_i=1) to synthesize the input data set(*i.e.* $y_{i1}, y_{i2}, \ldots, y_{iN}$, $i \in \{1,2,\ldots, M\}$ and e_i=1). On the right side of the input components are three squares that are encoder, low-dimensional hidden variables z and decoder. Encoder and decoder are composed of several fully connected layers. The arrow from $y_{i1}, y_{i2}, \ldots, y_{iN}$ to encoder means the input data set is inputted into the encoder. The function of encoder is to map a high-dimensional input to low-dimensional hidden variables z while decoder is to map from low-dimensional hidden variables back to high-dimensional inputs. Besides, PVaug has a unique noise mechanism. The encoder will output two representations, one is the original representation while the other is the representation that aims to control the noise interference level. Finally, the original representation and the noise representation will be added as the input of the decoder [6]. The final output of the decoder are the generated samples (*i.e.*, $z_{i1}, z_{i2}, \ldots, z_{iN}$) which are new model-domain test vectors with failing labels. After figuring out how to generate failing vectors, we should identify the generation number to achieve better localization efficiency. According to the previous study [8], a balanced set of model domain test vectors is beneficial for improving the effectiveness of fault localization. This indicates that class-balanced model-domain test vectors perform better than unbalanced ones for fault localization, which also means that we need to constantly generate failing vectors until we obtain a class-balanced model-domain data set.

2.2 An Illustrative Example

As shown in Fig. 2, this section uses an example to illustrate the application of PVaug. Suppose there is a program P with a faulty statement S_3. The test suite consists of 6 input-domain test cases shown in the second column on the left, the corresponding model-domain test cases of which are illustrated in the 3rd to 19th columns on the left. The 16 cells below each statement denote whether the corresponding statement is executed by an input-domain test case. We utilize 1 to represent a statement is executed while 0 means a statement is not executed. The rightmost cell records whether a test case is failed or not. We use 1 to denote a failing test vector and 0 to represent a passing one. From Fig. 2, we could observe that the number of failing vectors is 2 while the number of passing ones is 4, which leads to a class-imbalanced phenomenon. Since the fault propaga-

Program P																		Bug information
S1:Read(a,b,c) S2:d1=0,d2=0,d3=0; S3:if(b < 0){ S4:d1 = b; S5:d2 = c; S6:d3 = a;} S7:else {d1 = b+1;					S8: d2 = c+1; S9:if(a < 0){ S10:a = a+c;} S11: else a = a+b; S12: d3 = a+1;} S13:if(c>0){ S14:output(d1);}				S15:else {output(d2); S16:output(d3);} Slicing criterion: {t1, S14, d1} Slice result: {S1,S3,S7,S14}								S3 is faulty. Correct form: If(b<6){
test	a,b,c	S1	S2	**S3**	S4	S5	S6	S7	S8	S9	S10	S11	S12	S13	S14	S15	S16	result
t1	-1,5,3	1	1	1	0	0	0	1	1	1	1	0	1	1	1	0	0	1
t2	-2,-7,5	1	1	1	1	1	1	0	0	0	0	0	0	1	1	0	0	0
t3	5,-6,-8	1	1	1	1	1	1	0	0	0	0	0	0	1	0	1	1	0
t4	-5,8,-8	1	1	1	0	0	0	1	1	1	1	0	1	1	0	1	1	0
t5	4,7,11	1	1	1	0	0	0	1	1	1	0	1	1	1	1	0	0	0
t6	4,2,-1	1	1	1	0	0	0	1	1	1	0	1	1	1	0	1	1	1
t7		0.83	0	0.77	0	0	0	0.85	0	0	0	0	0	0	0.86	0	0	1
t8		0.79	0	0.81	0	0	0	0.87	0	0	0	0	0	0	0.89	0	0	1
MLP	rank	15	9	**12**	2	1	6	16	10	11	14	4	8	7	13	5	3	
MLP	rank	2	5	**4**	9	1	11	3	7	10	13	6	16	12	15	8	14	**PVaug**

Fig. 2 Example illustrating our approach.

tion context is the slice result $\{S_1, S_3, S_7, S_{14}\}$, the vector of context is $\{1,0,1,0,0,0,1,0,0,0,0,0,1,0,0\}$. PVaug intersects this context vector with original failing vectors as input and iteratively generates model-domain failing vectors until there are the same number of passing ones and failing ones, t_7 and t_8 are two generated failing vectors. At this time, an augmented model-domain test vectors are constructed.

With the augmented model-domain test vectors, we choose MLP-FL [2] as our baseline to conduct the suspiciousness calculation process. MLP-FL is a deep-learning-based fault localization method [2]. The MLP model has an input layer with 16 nodes, three hidden layers with the number of each one's nodes being 32 and one output layer. PVaug uses the class-balanced dataset as input to train the network iteratively. After training, we construct a virtual test suite as the test dataset to get the suspicious values of 16 statements and rank them in descending order. The final results are shown in the bottom two rows in Fig. 2. We could find that the faulty statement S_3 is ranked *4th* by using augmented model-domain test vectors while ranked *12th* by using the original model-domain test vectors. Therefore, PVaug obtains a better result on fault localization technique MLP-FL.

3. An Experimental Study

3.1 Experimental Setup

In our experimental study, we utilize 14 subject programs to conduct a comparison between PVaug and baselines, which are *chart*, *math*, *lang*, *closure*, *mockito*, *time*, *python*, *gzip*, *libtiff*, *space*, *nanoxml_v1*, *nanoxml_v2*, *nanoxml_v3* and *nanoxml_v5*. The first six subject programs are collected from Defects4J [†], the next three ones are from ManyBugs [††], and the last 5 ones are got from SIR [†††]. These programs are widely used in the field of fault localization [2]. Previous studies have shown that Ochiai, ER5, GP02, GP03, Dstar, GP19 and ER1' are all state-of-the-art spectrum-based fault localization (SBFL) techniques [9], and MLP-FL, CNN-FL, BiLSTM-FL and DeepFL are representative and effective deep-learning-based fault localization (DLFL) approaches [2], [10]. Thus, we choose and implement these 11 baselines based on the source code from the previous studies. Table 1 has provided the detailed information of the 14 subject programs including programs, the number of faulty versions, the number of thousand lines, and the number of input-domain test cases. We use a computer with 128G physical memory, a CPU of Intel I7-9700, and two 12G GPUs of NVIDIA TITAN X Pascal. The replication package is released online [††††].

Table 1 Subject programs in our experimental study.

Program	Versions	KLOC	Test
JFreeChart (chart)	26	96	2205
Apache Commons Math (math)	106	85	3602
Apache commons-lang (lang)	65	22	2245
Closure Compiler (closure)	133	90	7927
Framework for unit tests (mockito)	38	6	1075
Joda-Time (time)	27	53	4130
General-purpose language (python)	8	407	355
Data compression (gzip)	5	491	12
Image processing (libtiff)	12	77	78
ADL interpreter space (space)	38	6.1	13585
XML parser v1 (nanoxml_v1)	7	5.4	206
XML parser v2 (nanoxml_v2)	7	5.7	206
XML parser v3 (nanoxml_v3)	10	8.4	206
XML parser v4 (nanoxml_v4)	7	8.8	206

The metrics we used to verify the effectiveness of our approach are *Top-N*, *Mean Average Rank (MAR)*, *Mean First Rank (MFR)* and *Relative Improvement (RImp)* [10], [11]. For a fault localization method, *Top-N* means the percentage of faults located within the first *N* positions of a ranked list of all statements in descending order of suspiciousness, *Mean Average Rank (MAR)* is the mean of the average rank of all versions' faults, *Mean First Rank (MFR)* denotes the first faulty statement's rank of all faults when encountering multiple faults. *Relative Improvement (RImp)* is to compare the total number of statements that need to be checked to locate all faults with a baseline using PVaug versus the number that need to be checked without using PVaug.

[†]Defects4J, http://defects4j.org

[††]ManyBugs, http://repairbenchmarks.cs.umass.edu/ManyBugs/

[†††]SIR, http://sir.unl.edu/portal/index.php

[††††]Code is available at https://github.com/toolstemp/VAE_Aug.

Table 2 *Top-N*, *MAR* and *MFR* comparison of PVaug over the 12 fault localization approaches.

Comparison	top-1	top-5	top-10	MFR	MAR
MLP-FL	0%	2.1%	3.1%	213	352
MLP-FL (PVaug)	**0%**	**6.2%**	**12.7%**	**173**	**309**
CNN-FL	1.6%	3.1%	8.2%	179	263
CNN-FL (PVaug)	**1.6%**	**9.6%**	**14.9%**	**114**	**235**
BiLSTM-FL	0%	1.6%	3.1%	235	412
BiLSTM-FL (PVaug)	**0%**	**4.1%**	**9.7%**	**191**	**346**
DeepFL	1.6%	4.1%	9.3%	188	271
DeepFL (PVaug)	**1.9%**	**10.1%**	**16.9%**	**121**	**223**
ER5	0%	6.2%	12.4%	247	421
ER5 (PVaug)	**0%**	**8.5%**	**13.5%**	**214**	**371**
GP02	1.6%	15.5%	20.6%	263	545
GP02 (PVaug)	**1.6%**	**16.8%**	**23.1%**	**216**	**448**
GP03	3.1%	11.3%	12.4%	217	363
GP03 (PVaug)	**3.1%**	**16.2%**	**17.1%**	**194**	**309**
Dstar	3.1%	20.1%	26.3%	241	357
Dstar (PVaug)	**3.1%**	**22.5%**	**31.5%**	**225**	**316**
ER1 '	3.1%	18.6%	26.3%	242	371
ER1 ' (PVaug)	**3.6%**	**20.1%**	**29.4%**	**216**	**329**
GP19	3.1%	9.3%	15.5%	253	391
GP19 (PVaug)	**3.1%**	**11.9%**	**24.2%**	**221**	**334**
Ochiai	1.6%	20.1%	24.2%	215	363
Ochiai (PVaug)	**2.1%**	**23.6%**	**29.4%**	**164**	**276**

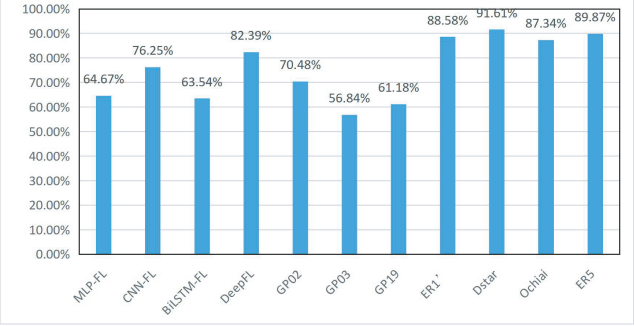

Fig. 3 *RImp* of PVaug on approaches.

3.2 Data Analysis

Table 2 illustrates the comparison results between 11 fault localization techniques using PVaug and without using PVaug. From Table 2, we could observe that after using PVaug, baselines get higher *Top-N* values and lower *MFR* and *MAR* values. This results indicate that PVaug could promote the effectiveness of fault localization.

In order to verify the detailed improvement on each subject program as well as on each baseline, we further utilize *RImp* score. From Fig. 3, we could observe the comparison results of *RImp* score. Concretely, Fig. 3 provides the *RImp* score on 11 fault localization techniques. In Fig. 3, the *RImp* score ranges from 56.84% to 91.61%, which indicates that PVaug is effective on all the 11 fault localization techniques. Taking CNN-FL as an example, the *RImp* score is 76.25%, which means that after seeking out all the faulty statements, the number of statements needs to be examined by CNN-FL after using PVaug is 76.25% of the number of statements needs to be checked by CNN-FL without using PVaug. It also indicates that PVaug helps CNN-FL to save 23.75% of statements that need to be examined to locate all faults of 14 subject programs. The maximum saving is 43.16% on GP03 while the minimum saving is 8.39% on Dstar.

Table 3 Comparison of PVaug over Aeneas, Bcl-fl and CGAN4FL.

Vs Aeneas	Result	Vs Bcl-fl	Result	Vs CGAN4FL	Result
BETTER	7	BETTER	6	BETTER	3
SIMILAR	4	SIMILAR	5	SIMILAR	5
WORSE	0	WORSE	0	WORSE	3

In order to investigate the significance of the statistically difference between PVaug and existing data augmentation techniques such as Aeneas, Bcl-fl and CGAN4FL [12]–[14], this experimental study adopts Wilcoxon-Signed Rank Test [2]. From Table 3, we could observe that PVaug gets 16 *BETTER* results (16/33 = 48.48%). Compared with Aeneas and Bcl-fl, PVaug utilizes a fault propagation context, which could show how a failure is caused and provide better effectiveness. Compared with CGAN4FL, PVaug utilizes a VAE model, which is easier to train. PVaug reduces average training time by 26% compared to CGAN4FL. In total, we could conclude that PVaug improves the efficiency of fault localization.

4. Conclusion

In this paper, we propose a new perspective to augment model-domain test vectors for fault localization with the utilization of fault propagation context and VAE. The large-scale experimental study has shown that PVaug effectively promotes the efficiency of state-of-the-art fault localization techniques.

Acknowledgments

This work is partially supported by China Postdoctoral Science Foundation (Certificate Number: 2023M732594), Characteristic Innovation Project of Ordinary University in Guangdong Province (Certificate Number: 2023KTSCX193).

References

[1] X. Xie, T.Y. Chen, F.-C. Kuo, and B. Xu, "A theoretical analysis of the risk evaluation formulas for spectrum-based fault localization," ACM Transactions on Software Engineering and Methodology (TOSEM), vol.22, no.4, p.31, 2013.

[2] 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, p.106486, 2021.

[3] W.E. Wong, R. Gao, Y. Li, A. Rui, and F. Wotawa, "A survey on software fault localization," IEEE Trans. Softw. Eng. (TSE), vol.42, no.8, pp.707–740, 2016.

[4] M. Böhme, C. Geethal, and V.-T. Pham, "Human-in-the-loop automatic program repair," 2020 IEEE 13th International Conference on Software Testing, Validation and Verification (ICST), pp.274–285, 2020.

[5] G. An and S. Yoo, "Human-in-the-loop fault localisation using efficient test prioritisation of generated tests," CoRR, vol.abs/2104.06641, 2021.

[6] C. Doersch, "Tutorial on variational autoencoders," arXiv preprint arXiv:1606.05908, 2016.

[7] B. Xu, J. Qian, X. Zhang, Z. Wu, and L. Chen, "A brief survey of program slicing," ACM SIGSOFT Software Engineering Notes, vol.30, no.2, pp.1–36, 2005.

[8] L. Zhang, L. Yan, Z. Zhang, J. Zhang, W.K. Chan, and Z. Zheng, "A theoretical analysis on cloning the failed test cases to improve spectrum-based fault localization," Journal of Systems and Software, vol.129, pp.35–57, 2017.

[9] S. Pearson, J. Campos, R. Just, G. Fraser, R. Abreu, M.D. Ernst, D. Pang, and B. Keller, "Evaluating and Improving Fault Localization," 2017 IEEE/ACM 39th International Conference on Software Engineering (ICSE), 2017.

[10] X. Li, W. Li, Y. Zhang, and L. Zhang, "DeepFL: Integrating multiple fault diagnosis dimensions for deep fault localization," Proceedings of the 28th ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA 2019), pp.169–180, 2019.

[11] C. Parnin and A. Orso, "Are automated debugging techniques actually helping programmers?," International Symposium on Software Testing and Analysis (ISSTA 2011), pp.199–209, 2011.

[12] H. Xie, Y. Lei, M. Yan, Y. Yu, X. Xia, and X. Mao, "A universal data augmentation approach for fault localization," Proceedings of the 44th International Conference on Software Engineering, pp.48–60, 2022.

[13] Y. Lei, C. Liu, H. Xie, S. Huang, M. Yan, and Z. Xu, "BCL-FL: A data augmentation approach with between-class learning for fault localization," 2022 IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER), pp.289–300, IEEE, 2022.

[14] Y. Lei, T. Wen, H. Xie, L. Fu, C. Liu, L. Xu, and H. Sun, "Mitigating the effect of class imbalance in fault localization using context-aware generative adversarial network," 2023 IEEE/ACM 31st International Conference on Program Comprehension (ICPC), 2023.