IEICE TRANS. INF. & SYST., VOL.E102–D, NO.9 SEPTEMBER 2019

LETTER

TFIDF-FL: Localizing Faults Using Term Frequency-Inverse Document Frequency and Deep Learning

Zhuo ZHANG[†], ***Nonmember***, **Yan LEI**[††a)], ***Member***, **Jianjun XU**[†b)], **Xiaoguang MAO**[†], ***and*** **Xi CHANG**[†], ***Nonmembers***

SUMMARY Existing fault localization based on neural networks utilize the information of whether a statement is *executed* or *not executed* to identify suspicious statements potentially responsible for a failure. However, the information just shows the binary execution states of a statement, and cannot show how important a statement is in executions. Consequently, it may degrade fault localization effectiveness. To address this issue, this paper proposes TFIDF-FL by using term frequency-inverse document frequency to identify a high or low degree of the influence of a statement in an execution. Our empirical results on 8 real-world programs show that TFIDF-FL significantly improves fault localization effectiveness.

key words: *debugging, fault localization, term frequency, inverse document frequency, deep learning*

1. Introduction

In the process of software development, debugging usually requires much manual involvement of debugging engineers. Researchers have developed many fault localization techniques to reduce the cost of debugging [1]. In recent years, deep learning has witnessed a rapid development and shows its promising ability of providing tremendous improvement in robustness and accuracy [2].

Thus, some researchers have preliminarily used deep neural networks with multiple hidden layers to discuss and evaluate the potential of deep learning in fault localization [3], [4]. They found that with the capability of estimating complicated functions by learning a deep nonlinear network's structure and attaining distributed representation of input data, deep neural networks exhibit strong learning ability from sample data sets. However, the existing analysis is still preliminary and needs much further study. For example, it utilizes a matrix as the training samples, among which the value of each element is either 1 meaning a statement is *executed* or 0 denoting a statement is *not executed*. We can observe that the binary information of a statement just whether a statement is executed or not, whereas it cannot show what degree of the influence of a statement in an execution. The existing analysis also uses small-sized programs (*i.e.* hundreds of lines of code) with all seeded faults. The recent research [5] has revealed that small-sized programs with artificial faults are not useful for predicting which fault localization techniques perform best on real faults. Furthermore, the previous research [6] has shown there are unique features in test cases related to faults, *e.g.* the execution frequency of each statement. However, the current approaches use this feature of each statement in just one test case, and do not consider their features from the view of all test cases. Consequently, it may cause some bias, posing a negative effect on fault localization effectiveness [7].

Therefore, this paper explores more about deep learning in improving fault localization, *i.e.*, we aim at obtaining more insights by proposing an approach to identify the impact of each statement in all test cases by using the features from the view of all test cases, rather than a binary status, and evaluating our results with large-scale programs. Specifically, we propose TFIDF-FL: an effective fault localization approach using term frequency-inverse document frequency (*TF-IDF*) [8] to reflect how important of a statement in the executions of a test suite. TFIDF-FL abstracts a statement as a word and uses *TF-IDF* to construct a matrix as the training samples, which reflect how important a word (*i.e.* a statement) in the executions of a test suite. Then, it uses the architecture of Muti-Layer Perceptrons (MLPs) to learn a model from the training samples. Finally, TFIDF-FL evaluates the suspiciousness of each statement of being faulty by testing the trained model using a virtual test set. We designed and performed an empirical study on 8 large real-world programs. The results show that TFIDF-FL can significantly improves fault localization effectiveness.

2. Approach

2.1 Overview

In information retrieval, *TF-IDF* is a numerical statistic that is intended to reflect how important a word is to a document in a collection. It is one of the most popular term-weighting schemes and is often used in searches of information retrieval, text mining, and user modeling [8]. The *TF-IDF* is the product of two statistics, *TF* means term frequency and *IDF* means inverse document frequency. The term frequency is the number of times a word occurs in a document while inverse document frequency is whether a word is common or rare across all documents. The term frequency of a word is low if it occurs few times in a document,

Manuscript received November 14, 2018.
Manuscript revised March 19, 2019.
Manuscript publicized May 27, 2109.
[†]The authors are with College of Computer, National University of Defense Technology, Changsha 410073, China.
[††]The author is with School of Big Data & Software Engineering, Chongqing University, Chongqing 400044, China.
a) E-mail: yanlei@cqu.edu.cn (Corresponding author)
b) E-mail: jianjun.xu@yeah.net (Corresponding author)
DOI: 10.1587/transinf.2018EDL8237

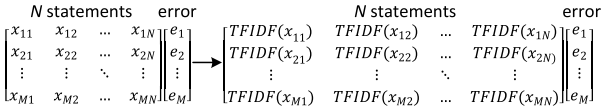

Fig. 1 The binary matrix and the TF-IDF matrix of M executions.

and is high if it occurs many times in a document. In contrast, the inverse document frequency of a word is low if it occurs in many documents, and is high if the word occurs in few documents.

The basic idea of TFIDF-FL is to adopt term frequency-inverse document frequency (*TF-IDF*) to build a matrix as the training samples reflecting the importance of a statement in the executions of a test suite, and utilize MLPs as the model for quantifying the suspiciousness of a statement of being faulty. In TFIDF-FL, *TF(s,t)* is the contribution of the statement s to test case t. The more statements are executed by the test case t, the lower value of *TF(s,t)* is. *IDF(s)* is whether the execution of the statement s is common or rare across all the test cases. *IDF(s)* is low if the execution of statement s occurs in many test cases, while it is high if the execution of statement s occurs in few test cases.

Given a program P with N statements (s_1, s_2, ..., s_N), it is executed by M test cases T (t_1, t_2, ..., t_M). In the binary matrix (see the left matrix of Fig. 1), x_{ij}=0 indicates that the statement j is not executed by the test case i, and we assign a value 0 to x_{ij}, and x_{ij}=1 otherwise. The error vector e represents the test results. The element e_i equals to 0 if the test case i passed, and 1 otherwise. Existing fault localization using deep learning [3] uses the left $M \times N$ matrix in Fig. 1 with the value of its element x_{ij} being 0 or 1. We can observe that the binary execution status of a statement cannot show how much influence of a statement in the executions of a test suite.

$$TF(x_{ij}) = x_{ij} * (1/log10(N(t_i))) \quad (1)$$

$$IDF(x_{ij}) = log10(M/(1 + DF(s_j))) \quad (2)$$

$$TFIDF(x_{ij}) = TF(x_{ij}) * IDF(x_{ij}) \quad (3)$$

Thus, based on x_{ij}, we leverage *TF-IDF* to define a new matrix reflecting different influence, rather than the binary status, of a statement in the executions of a test suite (see the right matrix of Fig. 1). Equation (1) calculates the value of $TF(x_{ij})$, where $N(t_i)$ means the number of executed statements in the test case t_i. In Eq. (1), we choose log10 because the empirical results show that the value of 10 is beneficial for fault localization effectiveness. Equation (2) calculates the value of $IDF(x_{ij})$, where $DF(s_j)$ indicates the number of test cases executing the statement s_j. Equation (3) calculates the value of $TFIDF(x_{ij})$ which is the multiplication of $TF(x_{ij})$ and $IDF(x_{ij})$.

The new matrix $M \times N$ with its element $TFIDF(x_{ij})$ can identify more different influence of a statement in comparison to the binary matrix. We use the new matrix as the training samples, and keep the error vector as their corresponding labels. We adopt mini-batch stochastic gradient descent to update network parameters with the batch size settled to h, namely, each time we feed a $h \times N$ matrix as input to the network and use its corresponding error vector as labels of these h training samples.

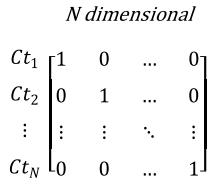

Fig. 2 Virtual test cases.

We use back propagation algorithm to fine-tune the parameters of the model. The goal is to minimize the loss between the training results and the error vector. The complex nonlinear relationship between the execution influence of a statement and the test results can be reflected after training. Finally, a set of virtual test cases (see Fig. 2) are constructed as the testing input, each time we choose one virtual test case and input it to the network, the output is the suspiciousness of the corresponding statement. As shown in Fig. 2, for a virtual test case Ct_i, only the *i-th* element of its row is 1, meaning that Ct_i just executes the statement s_i and does not execute the other statements. Since we have N statements, there are N corresponding virtual test cases. When the coverage vector of a virtual test case is inputted to the trained neural network, the output of the network is the estimation of the virtual test case of being failed by only covering one statement. The value of the result is between 0 and 1. The larger the value is, the more likely it is that the statement only covered the virtual test case is the buggy statement. For example, if we calculate suspiciousness of the statement s_i of being faulty, then we input Ct_i to the trained MLP, and the output of virtual test case Ct_i represents the probability of Ct_i of being failed by only covering the statement s_i. The probability value is the suspiciousness of the statement s_i.

2.2 An Illustrative Example

Figure 3 shows an example illustrating how our approach is to be applied with program P and a faulty statement s_6. The program P calculates the maximal value of three variables. The left 6 cells below each statement represent whether the statement is covered by the test case (1 for executed and 0 otherwise). The right 6 cells represent the *TF-IDF* values of each statement in each test case. The rightmost cells indicate whether the test case is failed or not (1 for fail and 0 otherwise). The concrete process is as follows: Firstly, TFIDF-FL constructs the MLP model with the number of input layer nodes being 8, 3 hidden layers with the number of each one's nodes being 10, and the number of output layer nodes being 1. Secondly, we input the vector t_1 (0,0,0,0,0.7,0,0,0) and its result 0, then vector t_2 (0,0,0,0,0.7,0,0,0) and its result 0 into the input layer until the coverage data and execution results are all inputted into the network. After that, we train the network iteratively to get the relationship between the exe-

Program P(maximal value of a,b,c)											Bug information								
S_1:Read(a) S_2:Read(b) S_3:Read(c) S_4:if(c > a) and (c > b){ S_5:max = c; } S_6:else if (a < b){								S_7:max = a; S_8:else {max = b;}				S_6 is faulty. Correct form: else if (a > b){							
T	a,b,c	S_1	S_2	S_3	S_4	S_5	S_6	S_7	S_8	T	S_1	S_2	S_3	S_4	S_5	S_6	S_7	S_8	R
t1	1,2,3	1	1	1	1	1	0	0	0	t1	0	0	0	0	0.7	0	0	0	0
t2	-2,-7,5	1	1	1	1	1	0	0	0	t2	0	0	0	0	0.7	0	0	0	0
t3	5,-6,-8	1	1	1	1	0	1	0	1	t3	0	0	0	0	0	0.2	0	0.6	1
t4	5,4,3	1	1	1	1	0	1	0	1	t4	0	0	0	0	0	0.2	0	0.6	1
t5	4,7,1	1	1	1	1	0	1	1	0	t5	0	0	0	0	0	0.2	0.6	0	1
t6	-1,2,1	1	1	1	1	0	1	1	0	t6	0	0	0	0	0	0.2	0.6	0	1
ZhengFL	value	0.62	0.64	0.61	0.69	0.57	0.59	0.60	0.58	TFIDF-FL	0.57	0.39	0.16	0.48	0.79	0.96	0.98	0.88	
	rank	3	2	4	1	8	6	5	7		5	7	8	6	4	2	1	3	

Fig. 3 Example illustrating our approach.

cution influence of a statement and the test results. Thirdly, we construct the virtual test set which is a 8 dimensional unit matrix, then put it into the network, and finally obtain the suspiciousness values.

Based on these information, The existing approach using binary information [3] (referred as ZhengFL) and TFIDF-FL using *TF-IDF* information both output a ranking list of all statements in descending order. The results show that the faulty statement s_6 is ranked *2nd* by TFIDF-FL and ranked *6th* by the one using binary information. We can observe that the statements s_1, s_2, s_3 and s_4 are executed by all the 6 test cases and however their *TF-IDF* scores are all 0. Consequently, TFIDF-FL identifies those less influential statements, and reduce their suspiciousness of being faulty. This binary information cannot capture such influential information. Therefore, our approach obtains a better localization result than the one using binary information.

3. An Experimental Study

3.1 Experimental Setup

The fault localization approach using Muti-Layer Perceptrons (MLPs), proposed by Zheng et al. [3], shows better performance over the representative and promising fault localization techniques (*e.g.* BP neural network [9], PPDG [10] and Tarantula [11]). However, ZhengFL still uses the binary information. Due to its promising results and its use of neural network, we mainly compare their approach (referred as ZhengFL) with our approach using *TFIDF* to demonstrate the effectiveness and potential of TFIDF-FL. The deep learning model used in our experiment is identical to that of ZhengFL. Furthermore, to obtain reliable experimental results, we choose those widely used real subject programs from the development of large-sized programs varying from 5 KLOC to 491 KLOC.

Table 1 lists the program name, the program function, the number of faulty versions used, the number of thousand lines of code, and the number of test cases. The first four programs are real faults, among which *python*, *gzip* and *libtiff* are collected from ManyBugs[†], and *space* is acquired from the SIR[††]. The last four programs are seeded faults of the four sperate releases of *nanoxml* acquired from the SIR. The physical environment on which we conducted the experiments was a computer containing a CPU of Intel I5-2640 with 128G physical memory and two 12G GPUs of NVIDIA TITAN X Pascal. The operating system was Ubuntu 16.04.3.

Table 1 The characteristics of subject programs.

Program	Description	Versions	KLOC	Test
python	General-purpose language	8	407	355
gzip	Data compression	5	491	12
libtiff	Image processing	12	77	78
space	ADL interpreter	35	6.1	13585
nanoxml_v1	XML parser	7	5.4	206
nanoxml_v2	XML parser	7	5.7	206
nanoxml_v3	XML parser	10	8.4	206
nanoxml_v5	XML parser	7	8.8	206

To evaluate the effectiveness of TFIDF-FL, we utilize fault localization accuracy (referred as *EXAM*[12]). *EXAM* is defined as the percentage of executable statements to be examined before finding the actual faulty statement. A lower value of *EXAM* indicates better performance. Then we adopt relative improvement (referred as *RImp*) [13]. It is to compare the total number of statements that need to be examined to find all faults using TFIDF-FL versus the number that need to be examined by using ZhengFL. A lower value of *RImp* shows better improvement [12] of TFIDF-FL over ZhengFL.

3.2 Data Analysis

Figure 4 illustrates the *EXAM* score of TFIDF-FL over ZhengFL. For each subplot, the horizontal axis represents the percentage of executable statements examined in all versions of subjects. Along the vertical axis, we can seek out the percentage of faults located in all faulty versions.

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

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

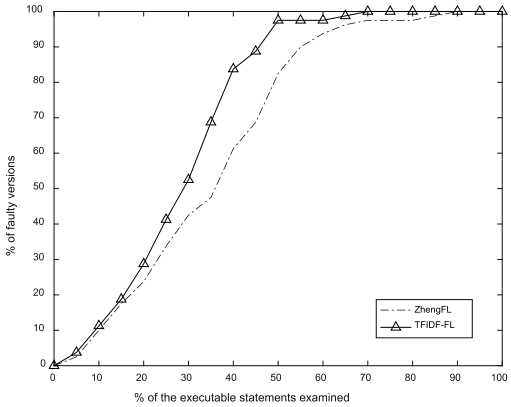

Fig. 4 *EXAM* of TFIDF-FL over ZhengFL.

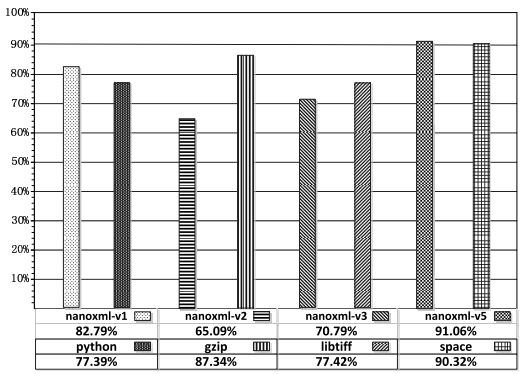

Fig. 5 *RImp* of TFIDF-FL over ZhengFL.

A point in Fig. 4 denotes when a percentage of executable statements is examined in each faulty version, the percentage of faulty versions has located their faults. We can observe that the curve of ZhengFL is always beneath that of TFIDF-FL. Thus, TFIDF-FL outperforms ZhengFL.

For detailed improvement in each program, we evaluate the *RImp* score of TFIDF-FL over ZhengFL. Figure 5 shows *RImp* score of TFIDF-FL over ZhengFL in each program. In comparison to ZhengFL, TFIDF-FL reduces the statements that need to be examined ranging from 65.09% (in *nanoxml_v2*) to 91.06% (in *nanoxml_v5*). This means that we need to examine from 65.09% to 91.06% of the statements that ZhengFL needs to examine of. The maximum saving is 34.91% (100% − 65.09% = 34.91%) on *nanoxml_v2* while the minimum saving is 8.94% (100% − 91.06% = 8.94%) on *nanoxml_v5*, which means that TFIDF-FL could reduce the checking number of statements from 8.94% to 34.91% over ZhengFL. The average saving is 19.72%. It shows that TFIDF-FL can reduce an average of 19.72% effort when using TFIDF-FL versus ZhengFL. Hence, TFIDF-FL significantly improves fault localization.

Although *RImp* can show more detailed improvement, the analysis using *RImp* evaluates TFIDF-FL from the overview of the results, and may miss other detailed view of the results. For example, suppose that TFIDF-FL has higher effectiveness than ZhengFL in several faulty versions of a program. Furthermore, ZhengFL has moderately higher effectiveness in most faulty versions of the programs. The sheer high effectiveness of TFIDF-FL in several faulty versions may make its *RImp* score lower than ZhengFL, showing that TFIDF-FL performs better than ZhengFL. However, in such case, we cannot conclude that TFIDF-FL performs better than ZhengFL. Thus, we need a more rigourous method to obtain a detailed result and adopt Wilcoxon-Signed-Rank Test [14] to achieve this goal, which is a non-parametric statistical hypothesis test for testing the differences between pairs of measurements F(x) and G(y). Table 2 shows the statistical results on this relationship (TFIDF-FL vs. ZhengFL) at the significant level of 0.05. We use *EXAM* scores as the measurements. Take *python* as an example. The p values of 2-tailed, 1-tailed(right) and 1-tailed(left) are 2.73E-02, 8.99E-01, and 1.81E-02 respectively. It means that the *EXAM* score of TFIDF-FL is significantly less than that of ZhengFL. Therefore, we obtain a BETTER conclusion, that is, TFIDF-FL performs better than ZhengFL in *python*. We can observe that TFIDF-FL obtains BETTER results in all the 8 programs.

Table 2 Wilcoxon-Signed-Rank test of the effectiveness relationship (TDIDF-FL vs ZhengFL).

Program	2-tailed	1-tailed(right)	1-tailed(left)	Conclusion
nanoxml_v1	2.85E-02	9.09E-01	2.11E-02	Better
nanoxml_v2	4.93E-02	7.89E-01	3.95E-02	Better
nanoxml_v3	2.80E-02	9.89E-01	1.73E-02	Better
nanoxml_v5	4.86E-02	7.05E-01	9.39E-01	Better
gzip	1.38E-02	9.47E-01	8.88E-02	Better
libtiff	3.10E-02	8.64E-01	1.76E-02	Better
python	2.73E-02	8.99E-01	1.81E-02	Better
space	3.74E-02	9.82E-01	1.91E-02	Better
Total	2.28E-04	1.00E+00	1.16E-04	Better

Thus, based on all the results and analysis, we can safely conclude that TFIDF-FL performs better than ZhengFL, showing that *TF-IDF* is useful to capture more subtle influence[†].

4. Conclusion

This paper proposes an effective fault localization approach using *TF-IDF* and deep learning method. We design and conduct an empirical study on the large-scale programs. The results show that TFIDF-FL significantly improves fault localization effectiveness In the future, we plan to improve the accuracy of TFIDF-FL. Moreover, we will seek the way to extend our current work to multiple-bugs cases.

Acknowledgments

This work is partially supported by the National Natural Science Foundation of China (Nos. 61602504, 61502296, 61379054, and 61672529), the Fundamental Research Funds for the Central Universities (No.

[†]Complete data of *RImp* on each faulty version and the source code are available at https://github.com/xiaofengwo/TF-IDF-fault-localization.

2019CDXYRJ0011), and the Scientific Research Fund of Hunan Provincial Education Department (No. 15A007).

References

[1] W.E. Wong, R. Gao, Y. Li, A. Rui, and F. Wotawa, "A survey on software fault localization," IEEE Transactions on Software Engineering, vol.42, no.8, pp.707–740, 2016.

[2] Y. Lecun, Y. Bengio, and G.E. Hinton, "Deep learning," Nature, vol.521, no.7553, pp.436–444, 2015.

[3] W. Zheng, D. Hu, and J. Wang, "Fault localization analysis based on deep neural network," Mathematical Problems in Engineering, vol.2016, pp.1–11, 2016.

[4] Z. Zhang, Y. Lei, Q. Tan, X. Mao, P. Zeng, and X. Chang, "Deep learning-based fault localization with contextual information," IEICE Transactions on Information and Systems, vol.E100-D, no.12, pp.3027–3031, 2017.

[5] S. Pearson, J. Campos, R. Just, G. Fraser, R. Abreu, M.D. Ernst, D. Pang, and B. Keller, "Evaluating and improving fault localization," Proceedings of the 39th International Conference on Software Engineering, pp.609–620, 2017.

[6] H.J. Lee, L. Naish, and K. Ramamohanarao, "Effective software bug localization using spectral frequency weighting function," Proceedings of the 34th Annual Computer Software and Applications Conference, pp.218–227, IEEE, 2010.

[7] Y. Lei, X. Mao, Z. Min, J. Ren, and Y. Jiang, "Toward understanding information models of fault localization: Elaborate is not always better," Proceedings of the 41st Annual Computer Software and Applications Conference, pp.57–66, 2017.

[8] A. Rajaraman and J.D. Ullman, "Mining of massive datasets," Cambridge University Press, 2011.

[9] W.E. Wong and Y. Qi, "Bp neural network-based effective fault localization," International Journal of Software Engineering and Knowledge Engineering, vol.19, no.4, pp.573–597, 2009.

[10] G.K. Baah, A. Podgurski, and M.J. Harrold, "The probabilistic program dependence graph and its application to fault diagnosis," International Symposium on Software Testing and Analysis, pp.189–200, 2008.

[11] J.A. Jones and M.J. Harrold, "Empirical evaluation of the tarantula automatic fault-localization technique," Proceedings of the 20th International Conference on Automated Software Engineering, pp.273–282, 2005.

[12] X. Mao, Y. Lei, Z. Dai, Y. Qi, and C. Wang, "Slice-based statistical fault localization," Journal of Systems and Software, vol.89, no.1, pp.51–62, 2014.

[13] V. Debroy, W.E. Wong, X. Xu, and B. Choi, "A grouping-based strategy to improve the effectiveness of fault localization techniques," International Conference on Quality Software, pp.13–22, 2010.

[14] G.W. Corder and D.I. Foreman, Nonparametric Statistics for Non-Statistician: A Step-by-Step Approach, International Statistical Review, 2010.