JITO: A Tool for Just-in-Time Defect Identification and Localization

Fangcheng Qiu[†]
Zhejiang University
China

Meng Yan[*†]
Chongqing University
China

Xin Xia
Monash University
Australia

Xinyu Wang
Yuanrui Fan
Zhejiang University
China

Ahmed E. Hassan
Queen's University
Canada

David Lo
Singapore Management University
Singapore

ABSTRACT

In software development and maintenance, defect localization is necessary for software quality assurance. Current defect localization techniques mainly rely on defect symptoms (e.g., bug reports or program spectrum) when the defect has been exposed. One challenge task is: can we locate buggy program prior to the appearance of the defect symptom. Such kind of localization is conducted at an early stage (e.g., when buggy program elements are being checked-in) which can be an early step of continuous quality control.

In this paper, we propose a **J**ust-**I**n-**T**ime defect identification and l**O**calization tool, named JITO, which can help developers to locate defective lines at check-in time. In summary, JITO contains two phases: (i) identify if a new change is buggy and (ii) locate suspicious buggy code lines in the identified buggy changes. We implement JITO as a plugin in an integrated development environment (i.e., Intellij IDEA). When developers using our plugin, JITO loads the local Git repository to build the JIT defect identification model and localization model based on historical changes. After submitting a new change to the local repository, developers apply JITO to identify whether it is a buggy change. If a buggy change is identified, JITO leverages JIT defect localization model to locate its suspicious buggy lines and highlight them in Intellij IDEA. Experimental results show that JITO outperforms two baselines (i.e., random guess and a static bug finder (i.e., PMD)) by a substantial margin in terms of four ranking measures.

Demo URL: https://youtu.be/tvnYs62FkEQ

Plugin download: https://git.io/Jf5r1

CCS CONCEPTS

• **Software and its engineering → Software development techniques**.

[*]Corresponding author.
[†]also with Pengcheng Laboratory, Shenzhen, China.

ESEC/FSE '20, November 8–13, 2020, Virtual Event, USA

ACM ISBN 978-1-4503-7043-1/20/11…$15.00
https://doi.org/10.1145/3368089.3417927

KEYWORDS

Defect Localization, Just-in-Time, Defect Identification, Software Naturalness

ACM Reference Format:
Fangcheng Qiu, Meng Yan, Xin Xia, Xinyu Wang, Yuanrui Fan, Ahmed E. Hassan, and David Lo. 2020. JITO: A Tool for Just-in-Time Defect Identification and Localization. In *Proceedings of the 28th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE '20), November 8–13, 2020, Virtual Event, USA.* ACM, New York, NY, USA, 5 pages. https://doi.org/10.1145/3368089.3417927

1 INTRODUCTION

In software development and maintenance, developers often spend much effort and resources for debugging [8]. Defect identification and localization aim to help developers save time in finding and locating suspicious defective program elements, such as defective lines of code. Previous work shows that defect localization can use information retrieval (IR) based techniques [13, 15, 24, 25] and spectrum-based techniques [1, 11, 14]. However, one disadvantage for the these localization techniques is that they require defect symptoms (e.g., from bug reports or execution traces). When we receive the defect symptoms, the defect has been exposed and caused negative impacts. In addition, as for defect identification, JIT defect identification is a well-known technique for identifying defects at check-in time. A number of studies have proposed various techniques for JIT defect identification [5, 9, 12, 18, 19, 21–23]. Although those JIT defect identification approaches can help developers identify buggy changes earlier, it is still challenging for locating the exact buggy positions (e.g., line-level) for a buggy change. A buggy change may introduce many lines of code (e.g., the average number of the introduced lines across all changes in Jmeter project is 180). In such case, it would cost a large amount of inspection effort if we inspect all the introduced lines for all the buggy changes.

In our previous work, to inspect a buggy change in order to locate the exact buggy lines with less inspection, we propose a two-phase framework of JIT defect Identification and Localization [20]. In the JIT defect identification phase, we use 14 change-level features to build a classifier to identify whether the change is a buggy change. In the JIT defect localization phase, we leverage software naturalness with the N-gram model to locate the buggy lines in a buggy change.

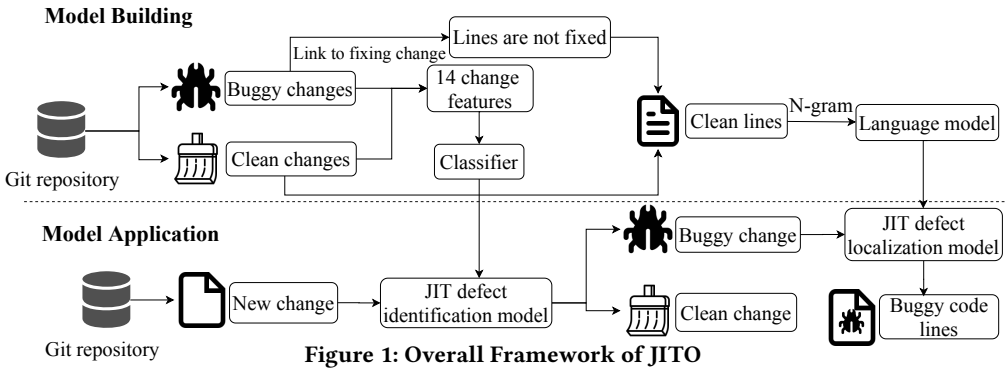

Figure 1: Overall Framework of JITO

In this paper, we present JITO, a tool based on our framework of JIT defect Identification and Localization. JITO is able to (i) identify if a new change is buggy and (ii) locate suspicious buggy code lines in the identified buggy changes. This tool is in a form of an integrated development environment (IDE) (i.e., Intellij IDEA[1]) plugin. When a model building action is called, JITO will load the local Git repository. After that, JITO extracts the 14 change-level features [4, 5, 9, 12, 23] of the Git repository's historical changes and leverages RA-SZZ algorithm to label each change as buggy or clean to build training set. Then it trains a classifier to construct the JIT defect identification model based on such training set. Next, JITO leverages software naturalness with the N-gram model [7] based on historical clean source code to construct a JIT defect localization model. When using JITO in practice, after submitting a new change to the local repository, developers can call JITO Analyze Change function in the plugin. JITO extracts the new change's features as extracted in the model building phase, and then applies the JIT defect identification model to identify it. If it is identified as buggy change, JIT defect localization model will locate suspicious buggy lines and highlight them in Intellij IDEA.

To evaluate JITO, we compare it with two baselines (i.e., random guess and a static bug finder (i.e., PMD)) on 14 open source projects with a total of 177,250 changes. JITO outperforms the two baselines by a substantial margin in terms of four ranking measures.

2 APPROACH

Figure 1 illustrates the overall framework of JITO. Our framework consists of two main phases: a model building phase and a model application phase. These two phases integrate JIT Defect Identification and JIT Defect Localization.

Model Building Phase. There are 3 steps in the model building phase: extracting data from historical changes, building JIT defect identification model, and building JIT defect localization model.

Extracting data from historical changes. Firstly, JITO loads the local Git repository. Then, it identifies the bug fixing changes by checking the keywords in change log and bug report ids. With such bug fixing changes, JITO identifies the buggy and clean changes by applying Refactoring Aware SZZ (RA-SZZ) algorithm [17]. Using the labeled changes, JITO identifies and collects clean lines. Clean lines are the lines introduced by clean changes, and the lines introduced by buggy changes but were not fixed later. In addition, we extract 14 change-level features from historical changes based on our prior work [20]. These features are grouped into five dimensions: diffusion (NS, ND, NF and Entropy) that characterizes the distribution of a change, size (LA, LD and LT) that demonstrates lines of code affected by a change, purpose (FIX) that indicates whether or not this commit is a bug fixing commit, history (NDEV, AGE and NUC) that characterizes how developers modify the files within the change in the code history and experience (EXP, REXP and SEXP) that captures a developer's experience.

Building JIT defect identification model. We train a logistic regression classifier learning from historical labeled changes by following prior studies [9, 10]. A new change would be identified as buggy if its predicted likelihood score is larger than 0.5; otherwise it will be classified as clean.

Building JIT defect localization model. Since N-gram model has been proved to be effective in modeling source code [7], we leverage N-gram language model to construct the JIT defect localization model. Language model is a probability distribution over sequences of words. Given a code fragment s of length $|s|$ and its code sequence $S = t_1 t_2..t_s$. A language model estimates the probability of this sequence occurring as a product of a series of conditional probabilities for each token. Because it is impossible to deal with a large amount of possible prefixes, the commonly used language model is N-gram language model, which assigns a probability to a sequence of words based on the Markov assumption. Additionally, to solve the probabilities may vary by orders of magnitude, we apply the logarithm of the phrase probability to arrive at the information-theoretic measure of entropy. Entropy represents the number of required bits to encode the phrase (aka., a token) given the language model. Entropy measures how "surprised" a model is by the given document. The higher entropy of a new code fragment indicates that new code fragment is more unnatural as compared to code in the training code corpus. Besides, based on Hellendoorn and Devanbu's study [6], we set N to be 6 and adopt the JM smoothing method [2] for modeling source code. We choose clean lines provided in previous step as training corpus. In detail, we break each line into separate words and use them to train the language model by applying tokenization tool that delimits code based on the Java grammar used by a prior study [16].

Model Application Phase. After submitting a new change to the local repository, the plugin calls the Analyze Change function. The built JIT defect identification model will identify the new change as buggy or clean. If the change is identified as buggy, the

[1]https://www.jetbrains.com/idea/

JIT defect localization model will compute the entropy of each token. Because our model aims to sort lines based on line entropy to find more unnatural lines which may be buggy lines, we compute the line entropy according to the entropy of its tokens and choose the summary of maximum entropy and average entropy of line as the line entropy. The line's maximum entropy is the maximum token entropy values in the line. The line's average entropy is the average tokens' entropy values in the line. Suppose there is a code lines of length $|s|$ with code sequence $S = t_1 t_2..t_s$, and the entropy of each token is denoted as $H_p(t_1), H_p(t_2), \ldots, H_p(t_s)$. We compute the line entropy ($H_p(s)$) as

$$H_p(s) = max(H_p(t_1), \ldots, H_p(t_s)) + \frac{1}{|s|}\sum_{n=1}^{|s|} H_p(t_i) \quad (1)$$

This approach captures both the most unnatural token sequences and the entire naturalness of a line. Finally, JITO will highlight the Top 10% of the most buggy lines of the change ranked by the line entropies in Intellij IDEA. Top 10% buggy lines can provide a short hint for locating the bug with limited inspection effort. Developers also can also specify other ratios on demand in JITO.

3 TOOL IMPLEMENTATION

We implement JITO in the form of an Intellij IDEA plugin. The source code can be found in our Github repository[2].

Tool Implementation. In JITO, developers will need to set the local Python interpreter path, JITO-Identification part path (e.g., /usr/a/JITO-identification), time period of training changes set (e.g., five months), and highlighted lines ratio (e.g., top 10%). When developers call the Build Model function, JITO will extract 14 change features of each changes from the project repository. Then JITO uses this data set to train the JIT defect identification model as well as collect clean line set. After that, JITO builds the JIT defect localization model based on the clean line set. This process will be performed in the background until developers are prompted after completion. When a developer submits a new change to the local Git repository, they can call the Analyze Change function. As a result, JITO leverages the JIT defect identification model to identify it as buggy or clean. If the new change is identified as buggy, JITO will invoke the JITO defect localization model, which will locate the suspicious buggy code lines and highlight them in Intellij IDEA. If it is identified as clean, the JITO will inform developers that this change is likely to be clean.

User Interface. Figure 2 shows the user interface of the JITO. Developers need to press the *Set Properties* button ② in the *Tools-JITO* menu ① to set Python interpreter path, JITO-identification part path, train set period, highlighted ratio, training set start time, and training set end time. Then they can press the *Build Model* button ③ in the *Tools-JITO* menu ① to build the defect identification and localization model. This process will be performed in the background. Developers can check the status in the status bar below. If a model is built successfully, it will send out a reminder to the developers. After submitting a new change to the local Git repository, developers can press the *Analyze Change* button ④ in the *Tools-JITO* menu ①. If JITO predicts there are buggy code lines, it will remind the developers that the change is buggy and highlight suspicious buggy code lines ⑥. Else, it will inform developers the change is clean. When the inspection is completed, developers can press *Unhighlight* button ⑤ to remove highlights.

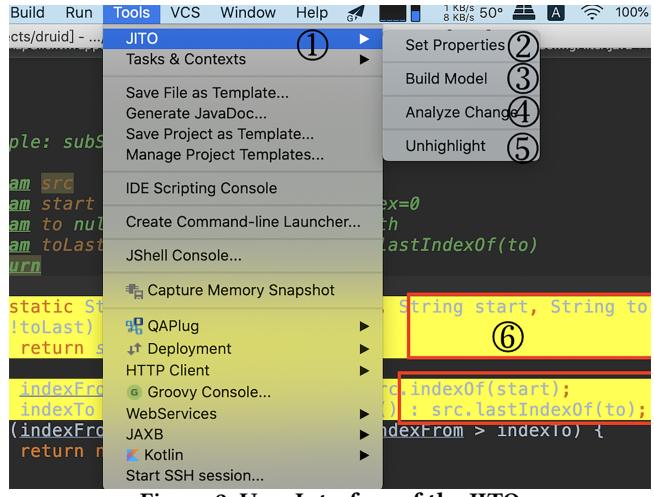

Figure 2: User Interface of the JITO

Table 1: The efficiency of the model building and application with different number of training changes for each project. MBT represents the Model Building Time. MAT represents the Model Application Time.

Project	1,000 changes		3,000 changes		6,000 changes	
	MBT /s	MAT /s	MBT /s	MAT /s	MBT /s	MAT /s
Deeplearning4j	63.83	40.41	125.45	48.12	250.49	66.43
Jmeter	24.39	3.71	87.42	5.27	228.70	8.23
H2o	26.42	9.50	93.22	31.76	237.78	93.91
Libgdx	26.98	6.36	82.26	13.25	379.24	131.61
Jetty	27.42	5.27	85.62	14.11	253.57	20.82
Robolectric	23.37	6.00	81.71	22.55	222.92	93.47
Storm	23.29	4.69	82.83	17.11	210.86	61.50
Jitsi	23.60	3.20	85.14	4.75	232.47	9.46
Jenkins	25.43	3.55	94.65	4.72	242.44	6.86
Graylog2-server	23.40	4.33	78.63	9.11	198.68	18.79
Flink	25.06	10.34	88.27	31.23	236.32	53.12
Druid	23.47	3.30	84.12	29.58	244.42	118.50
Closure-compiler	23.86	5.19	68.06	9.12	346.23	25.32
Activemq	26.04	5.26	98.99	11.60	240.42	18.02
average	*27.61*	*7.93*	*88.31*	*18.02*	*251.75*	*51.86*

4 EVALUATION

Data Collection. The dataset was collected from Github. We selected 14 projects which are written in Java, cover different application, and have many contributors. Besides, all the projects have over 5,000 changes and over 1,000 stars to ensure the studied projects have sufficient samples and are non-trivial ones. We collect the changes of studied from the creation data of the projects to March 1, 2018. Since we need change features to identify defect-introducing changes and to ensure most of the studied changes are correctly labeled, we use the changes until October 1, 2017. Too

[2]https://github.com/Lifeasarain/JITO

long intervals may cause lack of instances for our study. Too short intervals could introduce noise in our data. In our dataset, above 80% of the buggy changes were fixed within five months. Thus, five months interval is more appropriate. In total, there are 177,250 changes in the studied projects.

The efficiency of our tool is related to the numbers of training changes in the local Git repository. Therefore, we measure the efficiency by running our tool on the studied projects as described in data collection part. We measure the efficiency of JITO on each project by setting the number of training changes as 1,000, 3,000 and 6,000 changes for simulating small-size, medium-size and large-size projects. We run the experiments on the i9-9900k and 16GB RAM platform. Table 1 shows the result of our test.

In most small-size projects (with 1,000 changes), our tool can complete Model Building phase within 30 seconds and complete Model Application phase within 10 seconds. We choose to apply a total of 10 changes from 1,001 to 1,010 to run the JITO's Analyze Change function, and the average time is counted as model application time for each change. The following 3,000 changes and 6,000 changes are counted in the same way. In medium-size projects (with 3,000 changes), our tool takes an average of 88.31 seconds to complete Model Building phase and an average of 18.02 seconds to complete Model Application phase. In large-size projects (with 6,000 changes), our tool spends slightly more time. On average, it takes 257.75 seconds to complete Model Building phase and 51.86 seconds to complete Model Application phase. Because the submission interval of the first 3000 changes in Deeplearning4j is smaller than other projects and each change contains more modifications than other projects on average, the Deeplearning4j costs more time in both Model Building phase and Model Application phase.

In addition, we conduct an empirical study on the studied 14 projects to evaluate the effectiveness of our tool. We build a JIT defect localization model on each project. For each change in testing set, we first classify it as buggy or clean using the JIT defect identification model learning on our training set. Then, for a likely buggy change identified by the JIT defect identification model, we perform the JIT defect localization model. We also implement two baselines, i.e., random guess and a static bug finder baseline, PMD [3]. We compare the effectiveness of the proposed approach with these baselines.

Baseline 1: Random Guess (RG). RG randomly sorts the introduced lines. Since the performance of RG relates to the order of lines, we sort the introduced lines randomly and repeat 100 times to get the median performance.

Baseline 2: PMD. PMD is a popular static bug finder. It produces line-level warnings and assigns a priority for each warning. We use the PMD tool to scan the changed files then record the warning priority (i.e., 1-6, 6 means clean) of each introduced line by the change [3, 20]. Additionally, since some lines might have equal priority, we add a small random amount from [0, 1] to all line priority values for sorting. Then we sort the introduced lines according to these computed priority values (i.e., "1-6" + "[0,1]") in ascending order. The lines sorted at the top of the list are more likely to be the defect location. we repeat this process for 100 times and get the median performance for each change.

We choose identification ratio and misidentification ratio to measure tool's JIT defect identification performance. Identification ratio is the recall of our tool, which shows the ratio of correctly identified buggy changes among all buggy changes in our testing set. Misidentification ratio is the false positive rate of our tool, which shows the misidentified clean changes in our testing set. Besides, we choose MRR, MAP [16] and Top-k Accuracy to measure the tool's performance. MRR measures how far we need to check down a sorted list of added lines of a buggy change to locate the first buggy line. MAP considers the ranks of all buggy lines in that sorted list. Top-k Accuracy measures whether Top-k most likely buggy lines returned by our approach is actually the buggy location. In this paper, we set k = 1 and 5. We apply a time-aware validation setting that divides the training and testing sets. We choose the first 60% of the changes as our training set, and the remaining 40% of the changes as testing set according to the change time.

In JIT defect identification phase, JITO achieves an average identification ratio of 0.843, and a misidentification ratio of 0.284. Since this phase is not the main contribution of our work, we simply implement a prior approach which provides a reasonable performance already.

In JIT defect localization phase, JITO achieves a reasonable and better performance than the baselines on average across the 14 projects. The tool achieves an MRR of 0.396, an MAP of 0.353, a top-1 accuracy of 0.265 and a top-5 accuracy of 0.544 considering the identified buggy changes. It means that our tool can successfully locate the first buggy line in about 3 lines on average and locate at least the buggy line at top-1 position with a probability of 26.5%, and at top-5 positions with a probability of 54.0% of the identified-buggy changes. Our tool outperforms the two baselines in all of 14 projects in terms of MRR and MAP, and in most cases in terms of Top-k accuracy. More details of our evaluation results can be found in our journal paper [20].

5 CONCLUSION AND FUTURE WORK

In this paper, we present a tool named JITO, which combines JIT defect identification and JIT defect localization. JITO uses project historical changes to train a JIT defect identification model that can identify the buggy change. Then JITO leverages project historical clean code to build a JIT defect localization model that can locate the buggy lines. JITO is implemented as Intellij IDEA plugin, and it provides highlight on suspicious buggy code lines introduced by a buggy change at check-in time in an integrated development environment. In the future, we will do further research on investigating whether or not the entropy of the changed lines could be used to enhance the JIT defect identification model. Moreover, further research can investigate whether or not the different language model (i.e., cache model, nested model) can improve the effectiveness of JITO. Besides, we will enhance JITO to support more programming languages (e.g., Python, C++) and code editors (e.g., VS Code).

ACKNOWLEDGMENTS

This research was partially supported by the National Key R&D Program of China (No.2019YFB1600700), NSFC Program (No.61972339), the Australian Research Council's Discovery Early Career Researcher Award (DECRA) (DE200100021), the Fundamental Research Funds for the Central Universities (No.2020CDJQY-A021) and Alibaba-Zhejiang University Joint Institute of Frontier Technologies.

REFERENCES

[1] Rui Abreu, Peter Zoeteweij, Rob Golsteijn, and Arjan JC Van Gemund. 2009. A practical evaluation of spectrum-based fault localization. *Journal of Systems and Software* 82, 11 (2009), 1780–1792.

[2] Stanley F Chen and Joshua Goodman. 1999. An empirical study of smoothing techniques for language modeling. *Computer Speech & Language* 13, 4 (1999), 359–394.

[3] Tom Copeland. 2005. *PMD applied*. Vol. 10. Centennial Books Arexandria, Va, USA.

[4] Yuanrui Fan, Xin Xia, David Lo, and Shanping Li. 2018. Early prediction of merged code changes to prioritize reviewing tasks. *Empirical Software Engineering* 23, 6 (2018), 3346–3393.

[5] Wei Fu and Tim Menzies. 2017. Revisiting unsupervised learning for defect prediction. In *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering*. 72–83.

[6] Vincent J Hellendoorn and Premkumar Devanbu. 2017. Are deep neural networks the best choice for modeling source code? , 763–773 pages. https://doi.org/10.1145/3106237.3106290

[7] Abram Hindle, Earl T Barr, Zhendong Su, Mark Gabel, and Premkumar Devanbu. 2012. On the naturalness of software. In *2012 34th International Conference on Software Engineering (ICSE)*. IEEE, 837–847.

[8] Thong Hoang, Richard J. Oentaryo, Tien-Duy B. Le, and David Lo. 2019. Network-Clustered Multi-Modal Bug Localization. *IEEE Transactions on Software Engineering* 45, 10 (2019), 1002–1023. https://doi.org/10.1109/tse.2018.2810892

[9] Qiao Huang, Xin Xia, and David Lo. 2017. Supervised vs unsupervised models: A holistic look at effort-aware just-in-time defect prediction. In *2017 IEEE International Conference on Software Maintenance and Evolution (ICSME)*. IEEE, 159–170.

[10] Qiao Huang, Xin Xia, and David Lo. 2019. Revisiting supervised and unsupervised models for effort-aware just-in-time defect prediction. *Empirical Software Engineering* 24, 5 (2019), 2823–2862.

[11] James A Jones and Mary Jean Harrold. 2005. Empirical evaluation of the tarantula automatic fault-localization technique. In *Proceedings of the 20th IEEE/ACM international Conference on Automated software engineering*. 273–282.

[12] Yasutaka Kamei, Emad Shihab, Bram Adams, Ahmed E Hassan, Audris Mockus, Anand Sinha, and Naoyasu Ubayashi. 2012. A large-scale empirical study of just-in-time quality assurance. *IEEE Transactions on Software Engineering* 39, 6 (2012), 757–773.

[13] An Ngoc Lam, Anh Tuan Nguyen, Hoan Anh Nguyen, and Tien N Nguyen. 2017. Bug localization with combination of deep learning and information retrieval. In *2017 IEEE/ACM 25th International Conference on Program Comprehension (ICPC)*. IEEE, 218–229.

[14] Chao Liu, Xifeng Yan, Long Fei, Jiawei Han, and Samuel P Midkiff. 2005. SOBER: statistical model-based bug localization. *ACM SIGSOFT Software Engineering Notes* 30, 5 (2005), 286–295.

[15] Stacy K Lukins, Nicholas A Kraft, and Letha H Etzkorn. 2010. Bug localization using latent dirichlet allocation. *Information and Software Technology* 52, 9 (2010), 972–990.

[16] Christopher D Manning, Prabhakar Raghavan, and Hinrich Schütze. 2008. *Introduction to information retrieval*. Cambridge university press.

[17] E. Neto, D. da Costa, and U. Kulesza. 2018. The impact of refactoring changes on the SZZ algorithm: An empirical study. In *2018 IEEE 25th International Conference on Software Analysis, Evolution and Reengineering (SANER)*. IEEE Computer Society, Los Alamitos, CA, USA, 380–390. https://doi.org/10.1109/SANER.2018.8330225

[18] Christoffer Rosen, Ben Grawi, and Emad Shihab. 2015. Commit guru: analytics and risk prediction of software commits. In *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering*. 966–969.

[19] Emad Shihab, Ahmed E Hassan, Bram Adams, and Zhen Ming Jiang. 2012. An industrial study on the risk of software changes. In *Proceedings of the ACM SIGSOFT 20th International Symposium on the Foundations of Software Engineering*. 1–11.

[20] Meng Yan, Xin Xia, Yuanrui Fan, Ahmed E. Hassan, David Lo, and Shanping Li. 2020. Just-In-Time Defect Identification and Localization: A Two-Phase Framework. *IEEE Transactions on Software Engineering* (2020), 1–1. https://doi.org/10.1109/tse.2020.2978819

[21] Xinli Yang, David Lo, Xin Xia, and Jianling Sun. 2017. TLEL: A two-layer ensemble learning approach for just-in-time defect prediction. *Information and Software Technology* 87 (2017), 206–220.

[22] Xinli Yang, David Lo, Xin Xia, Yun Zhang, and Jianling Sun. 2015. Deep learning for just-in-time defect prediction. In *2015 IEEE International Conference on Software Quality, Reliability and Security*. IEEE, 17–26.

[23] Yibiao Yang, Yuming Zhou, Jinping Liu, Yangyang Zhao, Hongmin Lu, Lei Xu, Baowen Xu, and Hareton Leung. 2016. Effort-aware just-in-time defect prediction: simple unsupervised models could be better than supervised models. In *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering*. 157–168.

[24] Klaus Changsun Youm, June Ahn, and Eunseok Lee. 2017. Improved bug localization based on code change histories and bug reports. *Information and Software Technology* 82 (2017), 177–192.

[25] Jian Zhou, Hongyu Zhang, and David Lo. 2012. Where should the bugs be fixed? more accurate information retrieval-based bug localization based on bug reports. In *2012 34th International Conference on Software Engineering (ICSE)*. IEEE, 14–24.