2015 IEEE International Conference on Software Quality, Reliability and Security

Is learning-to-rank cost-effective in recommending relevant files for bug localization?

Fei Zhao Yaming Tang Yibiao Yang Hongmin Lu Yuming Zhou Baowen Xu
State Key Laboratory for Novel Software Technology, Nanjing University, Nanjing, China
njumrz@gmail.com tangyaming_nju@sina.com yangyibiao@smail.nju.edu.cn
hmlu, zhouyuming, bwxu@nju.edu.cn

***Abstract*—Software bug localization aiming to determine the locations needed to be fixed for a bug report is one of the most tedious and effort consuming activities in software debugging. Learning-to-rank (LR) is the state-of-the-art approach proposed by Ye et al. to recommending relevant files for bug localization. Ye et al.'s experimental results show that the LR approach significantly outperforms previous bug localization approaches in terms of "precision" and "accuracy". However, this evaluation does not take into account the influence of the size of the recommended files on the efficiency in detecting bugs. In practice, developers will generally spend more code inspection effort to detect bugs if larger files are recommended. In this paper, we use six large-scale open-source Java projects to evaluate the LR approach in the context of effort-aware bug localization. Our results, surprisingly, show that, when taking into account the code inspection effort to detect bugs, the LR approach is similar to or even worse than the standard VSM (Vector Space Model), a naïve IR-based bug localization approach.**

Keywords-bug localization, bug reports, effort-aware, learning-to-rank, empirical study

I. INTRODUCTION

A software bug is a fault in a computer program or system that causes it to produce an incorrect or unexpected result, or to behave in unintended ways [1]. Bugs are inevitable during the development and maintenance procedure. As software is growing in its volume, a project team may receive a large number of bug reports in a short period of time. For example, the Eclipse project team can receive about 115 new bug reports in a day. For each bug report, developers must spend a large amount of time and effort to identify which source code entities (e.g., files and methods) should be modified in order to fix the bug.

The task to locate buggy source code entities is known as Bug Localization. To reduce the high cost in bug localization, currently information retrieval (IR) techniques have been used to locate source code files that are textually similar to a given bug report [2-4]. In these works, bug localization is considered as a ranking problem: Given a bug report and n source code entities, each entity will get a relevancy score according to its textual similarity with the bug report and then a descending ranked list of these entities will be fed back. An entity is relevant if it needs to be modified to fix the bug and an entity with a higher relevancy score is considered more likely to be relevant, vice versa. By checking the source code entities in the ranked list successively, developers are expected to save a lot of time and effort.

However, bug reports usually miss essential information [5] and could be textually mismatched with source code entities due to the difference between the natural language used in bug reports and the programming language employed in the code. Both the above two reasons will lower the accuracy of IR-based bug localization methods. In [6], Ye et al. argued that with the use of domain specific knowledge, the performance of a ranking method would be greatly improved. In particular, their study showed that a project specific API documentation could be used to bridge the lexical gap between the natural and programming language while bug-fixing history as well as the code change history was also helpful in locating a bug.

Ye et al. showed that their learning-to-rank (LR) approach outperformed BugLocator [7] and BugScout [3], as well as two baseline approaches (the standard VSM method and The Usual Suspects method [6]). This work is significant, as domain specific knowledge is introduced into IR-based bug localization field. However, we observe that the performance metrics they used for evaluation are all related to the "hit location" of a relevant file in the ranked list. That is, a model which ranks the buggy files higher in the ranked list will have a better performance, with no concern on how much more effort will their model save for developers than other models when detecting bugs. In [11], Arisholm et al. pointed out that, when evaluating the effectiveness of defect prediction models, it would be of vital importance to take into account the cost-effectiveness indicator, as it can lead to more realistic prediction evaluations. Recently, this viewpoint has been taken by several defect prediction studies [12-16].

In this paper, we focus on investigating the ranking ability of the LR approach in the context of effort-aware bug localization. The subject projects used in this study consist of six large-scale open source Java projects. Based on the data sets collected from these six projects, we first introduce three effort-aware evaluation criteria (i.e., Effort@k, MAE, and MFE). Then, we compare the ranking performance of LR with the standard VSM approach and the Usual Suspect approach for ranking files with respect to these three metrics. We found that when taking code inspection effort into consideration, in most cases the LR approach has a much better performance than the Usual Suspects approach. But surprisingly, VSM outperforms LR in terms of Effort@k and has a similar performance when it comes to MAE and MFE. As the LR approach needs more information and is more complicated than VSM, LR seems to be not of practical value when code inspection effort is taken into account.

The rest of this paper is organized as follows. Section II introduces the models used in our evaluation. Section III introduces the design of our study, including the research questions of our study and the evaluation metrics we used in this paper. Section IV describes the experimental results in detail. Section V summaries the related work. Section VI concludes the paper and outlines the directions for the future work.

II. Ranking Models

The Vector Space Model (VSM) is a standard technique used in information retrieval, an algebraic model based on the term-document matrix [9]. In the bug localization scenario, queries (i.e. bug reports) and documents (i.e. source code files) are represented as vectors of term weights. The vectors make up an m×n matrix whose rows represent individual documents and columns represent individual terms (i.e. words). The *i*-th, *j*-th element w_{ij} in the matrix is the weight of term t_i in document d_j. Thus the VSM similarity can be calculated as:

$$f(r,s) = sim(r,s) \quad (1)$$

In Eq. (1), *r* represents the bug report, *s* represents a source code file, while sim(r,s) calculates the cosine similarity between *r* and *s*.

Ye et al. introduced their LR approach based on the classic VSM method, which not only used the content of a bug report but also the domain knowledge of the software project. Their ranking approach captured useful relationships between a bug report and source code files from three aspects: API specifications, bug-fixing history, and the code change history. In Section 3 of prior work [6] Ye et al. defined their ranking function as a weighted combination of six features:

$$f(r,s)=\sum_{i=1}^{6} w_i \times \emptyset_i (r,s) \quad (2)$$

In Eq. (2), w_i represent the feature weights which are trained using the ranking SVM technique [6]. $\emptyset_i(r,s)$ represents six features respectively (i.e., surface lexical similarity, API-enriched lexical similarity, collaborative filtering score, class name similarity, bug-fixing recency and bug-fixing frequency).

Usual Suspects is a simple approach that recommends the most frequently fixed files:

$$f(r,s) = |br(r,s)| \quad (3)$$

In Eq. (3), br(r,s) represents the bug reports for which file *s* was fixed before report *r* is committed.

III. Study Design

In this section, we first present the three research questions we attempt to answer in this study. Then, we describe the experimental setup. Finally, we introduce the performance indicators for evaluating the effectiveness of LR in the context of effort-aware bug localization.

A. Research Questions

The objective of this study is to empirically investigate the effectiveness of the LR approach in the context of effort-aware bug localization, especially compared with two baseline bug localization approaches, the standard VSM method and the Usual Suspect method. In order to determine the practical value of LR, we formulate the following three research questions:

- **RQ1**: *Under the top k recommendation, on average how much code inspection effort is required to find relevant files for a bug report?* This question aims at investigating the average effort needed to find out the real buggy source code files if top *k* files are recommended by the bug localization methods. With the recommendation of top *k* files, compared with two baselines, if LR saves developers less effort, we immediately know that LR is indeed useless for them. In contrast, if the baselines show a better localization performance, it is probably more worthwhile to apply them in practice.
- **RQ2**: *How much code inspection effort should be cost for detecting all the relevant files for a bug report on average?* In practice, it is often the case that one bug report has several relevant buggy files. The purpose of RQ2 is to find out if LR will be more effort-saving than the two baselines if we want to detect all relevant files.
- **RQ3**: *How much code inspection effort should be cost for detecting the first relevant file for a bug report on average?* In this research question, we assume that once one relevant file has been detected, developers can easily identify any other files that also need to be modified. Under this assumption, this research question is to determine the performance of LR for detecting the first relevant file.

B. Experimental Setup

To investigate the effectiveness of the LR approach in the context of effort-aware bug localization, we re-implement LR and then evaluate the ranking performance using the same benchmark datasets that used by Ye et al. in Section 4 of prior work [6]. These datasets are collected from six projects, including AspectJ, Birt, Eclipse Platform UI, JDT, SWT, and Tomcat. Table I describes the data sets used in this study. In Table I, the first column is the project name. The second to the fifth columns are respectively the age of the project, the number of bug reports, and the number of API. From Table I, we can see that the age of these projects are more than 8 years. This indicates they are long-term projects.

Ideally, to obtain an accurate and realistic evaluation performance, for each bug report, the bug localization approach should be built based on the code repository for which a bug report is reported. However, the exact source code versions are generally difficult to obtain. In our study, to be consistent with Ye et al., we use the code version just before the bug report is committed, whereas most of previous the bug localization methods barely use just one or a few code versions to evaluate on all bug reports for convenience. What's more, we use the same way to create disjoint training and test data as described in Section 5 of prior work [6].

TABLE I. BENCHMARK DATASETS

Project	Age	# of bug reports	# of API
AspectJ	12	593	54
Birt	8	4,198	957
Eclipse[a]	13	6,495	1,314
JDT	13	6,274	1,329
SWT	12	4,151	161
Tomcat	12	1,056	389

a. Eclipse refers to Eclipse Platform UI

C. Performance indicators

The number of lines of code (LOC) is a typical indicator for measuring code inspection effort [11-15]. Source code files that contain more code lines usually take more effort for code inspection. In this study we evaluate the effectiveness of the IR-based bug localization approaches using the following three metrics:

- *Effort@k* measures the average code inspection effort required to find relevant files for a bug report:

$$\text{Effort@k}=\frac{\sum_{j=1}^{n}\sum_{i=1}^{k}\text{LOC}(f_{ij})}{|R|} \quad (4)$$

In Eq. (4), f_{ij} represents the *i*-th file in the ranked list for the *j*-th bug report, LOC(f_{ij})represents the number of lines of code in the file f_{ij}. *n* is total number of bug reports. |R| is the number of unique bug reports that have at least one relevant file in the top *k* ranked files.

Given any bug localization approach *m* and *n* bug reports, for the *j*-th bug report, *m* recommend top *k* files (i.e. the $f_{1j}...f_{kj}$ files) for code inspection. Then the code inspection effort required to inspect these *k* files is $\sum_{i=1}^{k}\text{LOC}(f_{ij})$. The total effort required to inspect these top *k* files for all these *n* bug reports is $\sum_{j=1}^{n}\sum_{i=1}^{k}\text{LOC}(f_{ij})$. If one file in the top *k* files is a relevant file (i.e. the file need to be fixed for the *j*-th bug report), then the *j*-th bug report belongs to *R* and vice versa. Therefore, Effort@k represents, under the top *k* recommendation, the average code inspection effort required to find relevant files for each bug report. Here, *k* could be 1, 2, 3, and so on. For example, the Effort@3 represents the average code inspection effort required to find relevant files for each bug report under the top 3 recommendation.

- *Mean Average Effort (MAE)* is derived from the standard metric Mean Average Precision (MAP), which is widely used in information retrieval [8]. It is defined as the Mean of the Average Effort (MAE) that required for inspecting all relevant files for each bug report. In order to define the MAE, we first introduce the AvgE@p and the AvgE$_j$. Given any bug localization approach *m* and *n* bug reports, for the *j*-th bug report, all files are ranked by *m*. If the *p*-th file in the ranked list is a relevant file, we use AvgE@p to denote the average effort required to inspect each of the top *p* files. More specifically, AvgE@p is defined as follow:

$$\text{AvgE@p}=\sum_{i=1}^{p}\frac{\text{LOC}(f_{ij})}{p} \quad (5)$$

For the *j*-th bug report, it might have several relevant files. Let P_j be the set of the order numbers of all relevant files in its ranked file list. Then, AvgE$_j$ is the average effort for inspecting all relevant files for the *j*-th bug report which is defined as follows:

$$\text{AvgE}_j=\frac{1}{|P_j|}\sum_{p\in P_j}\{\text{AvgE@p}\} \quad (6)$$

After that, for all *n* bug reports, the average effort MAE is defined as:

$$\text{MAE}=\frac{1}{n}\sum_{j=1}^{n}\text{AvgE}_j \quad (7)$$

Therefore, similarly to MAP, MAE is the mean of the average effort required to inspect all relevant files for each bug report.

- *Mean First Effort (MFE)* is the average effort required to find the first relevant file for each bug report. The MFE is formulated as:

$$\text{MFE}=\frac{1}{n}\sum_{j=1}^{n}\sum_{i=1}^{q_j}\text{LOC}(f_{ij}) \quad (8)$$

Here, q_j is the position of the first relevant file in the ranked list for the *j*-th bug report. Therefore, MFE measures the average effort required to find the first relevant file for each bug report.

IV. EXPERIMENTAL RESULTS

In this section, we report the experimental results in detail. In order to answer RQ1, RQ2, and RQ3, we use the procedure described in Section III-D to build the Learning to Rank (LR) approach, the Standard VSM approach, and the Usual Suspects (US) approach. Then we compare these three approaches with respect to the Effort@k, MAE, and MFE performance indicators. In section IV-A, we report the results from comparing the average code inspection effort required to find relevant files of the three approaches under the top k recommendation (RQ1). In Section IV-B, we report the results from comparing the effort for detecting all relevant files for each bug report (RQ2). In section IV-C, we report the results from comparing the effort for detecting the first relevant file for each bug report (RQ3).

A. Under the top k recommendation, on average how much code inspection effort is required to find relevant files for a bug report?

In this section, under the top k recommendation, we report the results from comparing the average code inspection effort required to find relevant files (i.e. the Effort@k in this study) for the LR approach, the VSM approach, and the US approach. Figure 1 to Figure 6 show the Effort@k for these three approaches under the top k recommendation on six data sets, in which k is ranging from 1 to 20 (RQ1). In these six Figures, the y-axis represents Effort@k (indicated by the number of thousands of lines of

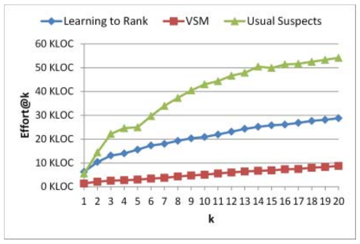


Figure 1. Effort graphs on Aspectj

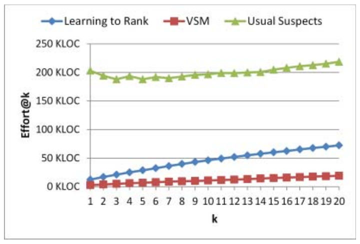


Figure 2. Effort graphs on SWT

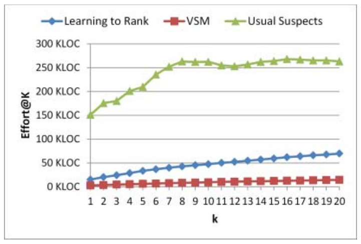


Figure 3. Effort graphs on Birt

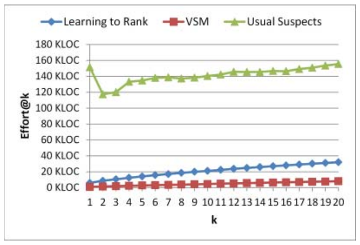


Figure 4. Effort graphs on Tomcat

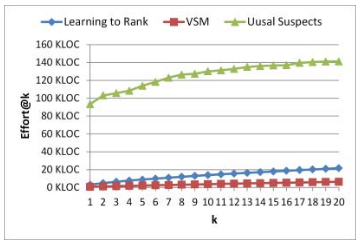


Figure 5. Effort graphs on Eclipse

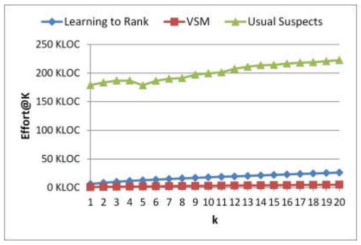


Figure 6. Effort graphs on JDT

code, i.e. the KLOC) and the x-axis represents k. k ranges from 1 to 20, indicating the top 1 to top 20 recommendations. The blue line, red line, and green lines respectively represent the Effort@k for the LR approach, the VSM approach, and the US approach. An approach with a larger Effort@k than other approaches indicates a worse performance.

From these six figures, we have the following observations:

1. Under any of the top 1 to top 20 recommendations, except under the top 1 recommendation in AspectJ, the US approach has a substantially larger Effort@k than the LR approach as well as the VSM approach in all six data sets. This indicates that, under any of the top 1 to top 20 recommendations, the US approach has the worst performance of the three models in finding the relevant files for each bug reports.

2. Under any of the top 1 to top 20 recommendations, the VSM approach has a smaller Effort@k than the LR approach in all six data sets. This indicates that, under the Effort@k evaluation, the VSM approach has the best performance of the three approaches in finding the relevant files for each bug reports.

For example, on the Tomcat project, for k=1, 3, 5, and 10, the LR approach has the Effort@k at around 6 KLOC, 10 KLOC, 14 KLOC, and 21 KLOC. This means that when the LR approach recommends one file for each bug report to developers, on average they need to inspect 6000 lines of code when they meet a correct recommendation. If we recommend 10 files for each bug report, the code inspection effort rises to 21 KLOC immediately. In comparison, the VSM approach achieves Effort@1 at around 1 KLOC and Effort@10 at around 5 KLOC. Meanwhile, the Usual Suspects achieves Effort@1 at around 150 KLOC and Effort@10 at around 140 KLOC.

We conjecture that the reason why the LR approach performs worse than the VSM approach may blame to the surface lexical similarity feature and the API-enriched lexical similarity feature. In these two features, to avoid the impact of file size on textual similarity between bug reports and source code files, Ye et al. split the files into methods. More specifically, given a bug report r and a large file s with n methods, the VSM approach only calculates the relevancy between r and s (i.e. the sim(r, s)). In addition to sim(r, s), the surface lexical similarity feature of the LR approach will also calculate the similarities between those methods belonging to m and r separately (i.e. the $\{sim(r, m) \mid m \in s\}$). According to formulae (4) in Section 3 of prior work [6], the final relevancy between r and s might result in a larger relevancy than sim(r, s). Therefore, for some large files, the LR approach might lead to a higher relevancy than the VSM approach. Consequently, larger files have a higher possibility to be ranked top in the LR approach. That is, when the LR approach improves the Accuracy@k (the percentage of bug reports for which a ranking model make at least one correct recommendation in the top k files [6]), it cannot save any effort yet.

Thus, for the first research question, under the top k recommendation, on average LR incurs more code inspection effort compared to the standard VSM approach.

B. *RQ2: How much code inspection effort should be cost for detecting all the relevant files for a bug report on average?*

In this section, we report the results from comparing the average effort required for detecting all the relevant files (i.e. the MAE) for the LR, VSM and US approaches.

Figure 7 shows the MAE for the LR, VSM, and US approaches on six data sets. The upper part in figure 7 employs the histogram to compare the MAE for these three approaches on six data sets. The lower part in Figure 7 is a table used for describing the numerical value of the MAE for these three approaches on six data sets. Note that Eclipse in this figure refers to the project Eclipse Platform UI.

From Figure 7, we have the following observations:

1. The US approach has the largest MAE than the other two approaches. This indicates that, in order to detecting all relevant files, the US approach requires more code inspection effort on average.

2. On AspectJ and SWT projects, the VSM approach has larger MAE than the LR approach. While on other projects, the VSM approach has a similar or even smaller MAE than the LR approach. This indicates that, in most cases, the VSM approach requires a similar or less code inspection effort than the LR approach for detecting all relevant files on average.

Interestingly, we find that the performance of a ranking approach may rely on the Accuracy@k results to some extent. For a project, the ranking approach that has the best Accuracy@k usually has the best MAE. MAE indicates the average code inspection effort for detecting all the relevant files. The higher the relevant files are listed in the ranked list, the smaller MAE the ranking approach will have. For example, in Tomcat, LR has the MAE around 21 KLOC, VSM has the MAE around 16 KLOC, and US has the MAE around 73 KLOC, which are consistent with the results of Accuracy@k reported by Ye et al. What's more, the MAE of all the three ranking approaches is extremely large for the Birt project, from the study by Ye we can see that the Birt indeed had a bad Accuracy@k performance.

In case of MAE, it appears that the LR approach has a performance similar to the VSM approach. However, the effort required for code inspection is still very expensive. For the best of the six projects, MAE is still around 20 KLOC, which is not as good as we imagine.

C. *RQ3: How much code inspection effort should be cost for detecting the first relevant file for a bug report on average?*

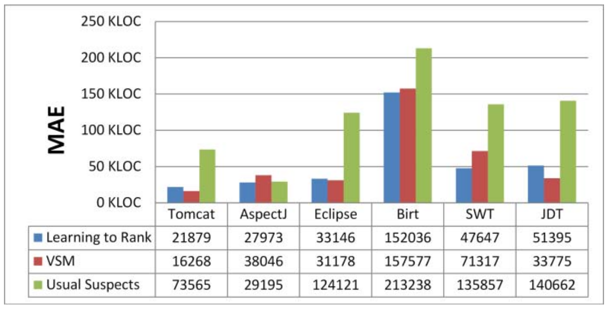
In this section, we report the results from comparing the average effort required for detecting the first relevant files (i.e. the MFE) for the LR, VSM and US approaches.

Figure 8 shows the MFE for the LR, VSM, and US approaches on six data sets. Again, the upper part in Figure 8 employs a histogram to compare the MFE for these three approaches on six data sets. The lower part in Figure 8 is a table used for describing the numerical value of the MFE for these three approaches on six data sets.

From Figure 8, we have the following observations:

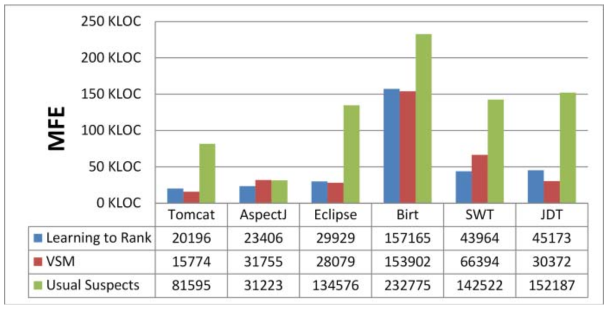
1. The US approach has a larger mean MFE than the LR approach and the VSM approach. This indicates the US approach requires more code inspection effort for detecting the first relevant file on average.

2. On AspectJ and SWT projects, the VSM approach has a larger mean MFE than the LR approach. While on other projects, the VSM approach has smaller mean MFE than the LR approach. This indicates that, in most cases, the VSM approach requires less code inspection effort than the LR approach for detecting the first relevant file on average.



	Tomcat	AspectJ	Eclipse	Birt	SWT	JDT
🟦 Learning to Rank	21879	27973	33146	152036	47647	51395
🟥 VSM	16268	38046	31178	157577	71317	33775
🟩 Usual Suspects	73565	29195	124121	213238	135857	140662

Figure 7. MAE Comparision



	Tomcat	AspectJ	Eclipse	Birt	SWT	JDT
🟦 Learning to Rank	20196	23406	29929	157165	43964	45173
🟥 VSM	15774	31755	28079	153902	66394	30372
🟩 Usual Suspects	81595	31223	134576	232775	142522	152187

Figure 8. MFE Comparision

Overall, the above observations suggest that the LR and VSM approaches incur similar code inspection effort to detect the first relevant file on average.

Based on the results from Section IV-A, Section IV-B, and Section IV-C, in the context of effort-aware bug localization, we find that, in most cases, the VSM approach performs similarly to or better than the LR approach. Since the VSM approach has a building cost much lower than the LR approach, we therefore suggest choosing the VSM approach for effort-aware bug localization in practice.

V. Related Work

Recent years have seen an increasing interest in information retrieval based bug localization approaches [2-4, 7]. These approaches treat bug reports as queries and rank the source code files by their textual relevance to the queries. Unlike spectrum-based bug localization techniques [17-19], IR-based bug localization methods do not need the runtime information such as passing and failing traces.

J. Zhou et al. [7] proposed BugLocator, a revised Vector Space Model to recommend files to fix. In their study, they first took information of similar bugs into consideration. In addition to measure the textual similarity between bug reports and source code files, they considered that larger files and files that had been fixed in similar bugs were more likely to contain bugs. In [4], Rao et al. measured the textual similarity between bug reports and source code fragments instead of source code files. In their experiment, they found that complex IR approaches such as LDA and LSI did not outperform simple models like VSM. In [2], Lukins et al. built their ranking approach with a combination of LDA and VSM. In their approach, topics generated from LDA were used for indexing the source code files while VSM was then applied to compute the textual similarities between bug reports and source file topics.

However, to the best of our knowledge, among all previous bug localization approaches, no one took into consideration the code inspection effort during their performance evaluation. Arisholm et al. [11] have pointed out that it is very important to take into account the effort required for code inspection when evaluating the performance of defect prediction models. The above viewpoint has been taken by several defect prediction studies recently [12-16].

Software bug localization is one of the most tedious and effort consuming activities in software debugging. In this paper, we evaluate the LR approach in the context of effort-aware bug localization. To the best of our knowledge, it is the first time to introduce the effort of code inspection into the area of bug localization. From this point of view, we can determine whether a bug localization approach such as LR is of practical value or not.

VI. Conclusion and Future Work

With the need of locating buggy files automatically, Ye et al. proposed the LR approach that first introduced domain specific knowledge into the area of IR-based bug localization. In their evaluation, they found that the proposed LR approach outperformed two state-of-the-art approaches and two baselines. However, we found that they did not take into consideration code inspection effort when evaluating the effectiveness of their approach.

In this paper, we examine the utility of the LR approach in the perspective of effort-aware bug localization. With the three effort-aware metrics we defined, we find that the LR approach is useful as it performs much better than the Usual Suspect in overall. However, LR performs worse than VSM for all the projects in terms of Effort@k and has a similar result when it comes to MAE and MFE. The results suggest that VSM, the simple information retrieval method, may be more suitable for practitioners.

In the future, we would like to replicate the study to validate the findings on more different software projects and bug localization methods.

Acknowledgment

This work is supported by the National Key Basic Research and Development Program of China (2014CB340702), the National Natural Science Foundation of China (61432001, 91318301, 61272082, 61300051), and the Natural Science Foundation of Jiangsu Province (BK20130014). We especially thank Ye Xin, the author of prior work [6], for his valuable assistance in the implementation of the LR approach.

References

[1] H. Chaudhary. C Programming::The Definitive Beginner's Reference. Createspace, 2014.

[2] S. Lukins, N. Kraft, L. Etzkorn. Bug Localization Using Latent Dirichlet Allocation. Information and Software Technology, 52(9), 2010: 972-990.

[3] A. Nguyen, T. Nguyen, J. Al-Kofahi, H. Nguyen, T. Nguyen. A Topic-Based Approach for Narrowing the Search Space of Buggy Files from a Bug Report. 26th IEEE/ACM International Conference on Automated Software Engineering, 2011: 263-272.

[4] S. Rao, A. Kak. Retrieval from Software Libraries for Bug Localization: A Comparative Study of Generic and Composite Text Models. In: Proceedings of the 8th Working Conference on Mining Software Repositories, 2011: 43-52.

[5] S. Breu, R. Premraj, J. Sillito, T. Zimmermann. Information needs in bug reports: Improving cooperation between developers and users. In: Proceedings of the 2010 ACM conference on Computer supported cooperative work, 2010: 301-310.

[6] X. Ye, R. Bunescu, C. Liu. Learning to Rank Relevant Files for Bug Reports using Domain Knowledge. In: Proceedings of the 22nd ACM SIGSOFT International Symposium on the Foundations of Software Engineering, 2014: 66-76.

[7] J. Zhou, H. Zhang, D. Lo. Where should the bugs be fixed? more accurate information retrieval-based bug localization based on bug reports. 34th International Conference on Software Engineering, 2012: 14-24.

[8] C. Manning, P. Raghavan, H. Schütze. Introduction to Information Retrieval. Cambridge: Cambridge university press, 2008.

[9] G. Salton, A. Wong, C. Yang. A Vector Space Model for Automatic Indexing. Communications of the ACM, 18(11), 1975: 613-620.

[10] F. Rahman, P. Devanbu. How, and why, process metrics are better. In: Proceedings of the 2013 International Conference on Software Engineering, 2013: 432-441.

[11] E. Arisholm, L. Briand, and E. Johannessen. A systematic and comprehensive investigation of methods to build and evaluate fault prediction models. Journal of Systems and Software, 83(1), 2010: 2-17.

[12] T. Menzies, Z. Milton, B. Turhan, B. Cukic, Y. Jiang, A. Bener. Defect prediction from static code features: current results, limitations, new approaches. Automated Software Engineering, 17(4), 2010: 375-407.

[13] Y. Zhou, B. Xu, H. Leung, L. Chen. An In-depth Study of the Potentially Confounding Effect of Class Size in Fault Prediction. ACM Transactions on Software Engineering and Methodology, 23(1), 2014: 10:1–10:51.

[14] Y. Yang, Y. Zhou, H. Lu, L. Chen, Z. Chen, B. Xu, H. Leung, Z. Zhang. Are slice-based cohesion metrics actually useful in effort-aware post-release fault-proneness prediction? An empirical study. To Appear in IEEE Transactions on Software Engineering, 2015.

[15] F. Rahman, D. Posnett, A. Hindle, E. Barr, P. Devanbu. BugCache for Inspections: Hit or Miss? In: Proceedings of the 19th ACM SIGSOFT Symposium and the 13th European Conference on Foundations of Software Engineering, 2011: 322–331.

[16] F. Rahman, P. Devanbu. How, and Why, Process Metrics Are Better. In: Proceedings of the 2013 International Conference on Software Engineering, 2013: 432–441.

[17] R. Abreu, P. Zoeteweij, R. Golsteijn, A. Gemund. A practical evaluation of spectrum-based fault localization. Journal of Systems and Software, 82(11), 2009: 1780-1792.

[18] B. Liblit, M. Naik, A. Zheng, A. Aiken, M. Jordan. Scalable statistical bug isolation. In: Proceedings of the 2005 ACM SIGPLAN Conference on Programming Language Design and Implementation, 2005: 15-26.

[19] C. Liu, L. Fei, X. Yan, S. Midkiff, J. Han. Statistical debugging: a hypothesis testing-based approach. IEEE Transactions on Software Engineering, 32 (10), 831-848.