Bug Report Recommendation for Code Inspection

Shin Fujiwara*, Hideaki Hata*, Akito Monden*, Kenichi Matsumoto*
*Graduate School of Information Science
Nara Institute of Science and Technology, Japan
Email: {fujiwara.shin.fe5, hata, akito-m, matumoto}@is.naist.jp

***Abstract*—Large software projects such as Mozilla Firefox and Eclipse own more than ten thousand bug reports that have been reported but left unresolved. To utilize such a great amount of unresolved bug reports and accelerate bug detection and removal, we propose to a way recommend programmers a bug report that is likely to contain failure descriptions related to a source file being inspected. We employ the vector space model (VSM) to make a relevancy ranking of bug reports to a given source file. The result of an experiment using data of three open source software projects showed that the accuracies of recommendations ranged from 21.74% to 60.05% in terms of the percentage of recommendations that contained relevant bug reports in a top 10 recommended list.**

I. Introduction

Code inspection has been considered as a most cost effective way to discover bugs in source code. It has been shown in many articles that inspection can discover and remove much more bugs per man-hour than other competing methods such as testing and static code analysis [1]–[3]. On the other hand, inspection is a hard and brain-exhausting work, and it eats up busy peoples' time. Indeed, many companies do not do many inspections although everybody knows their value [4]. Therefore, there still exists a great need for an effective support to make inspections more easy and feasible. To support inspection processes, various methods have been proposed including reading techniques [5]–[7], supporting distributed meetings [8], face-to-face meetings [9] and phased inspections [10], supporting a decision as to whether or not a detected defect is really a defect [11], and so on.

As a new way to support inspections, this paper focuses on accumulated (unresolved) bug reports, which have not been noticed in past studies, but we consider them to have a great potential to support finding bugs during inspections. Generally, in a large software project, many bug reports are posted and accumulated everyday but most of them remain unresolved although they may contain useful information to find bugs. For example, in case of Mozilla Firefox, more than ten thousands bug reports are reported but still remain resolved.

To utilize such a great amount of unresolved bug reports, we propose to recommend an inspector a bug report that is relevant to a source file being inspected. Obviously, finding bugs becomes much easier if the bug report describes a failure caused by an existing bug (fault) in the source file. To identify a relevant bug report, this paper employs the vector space model (VSM) to represent both bug reports and a source file, and to make a relevancy ranking of bug reports based on a query vector of a given source file. We use a standard natural language processing procedure such as tokenizing, reserved word removal and stemming.

Bug report recommendation is a proposal of new scenario of code inspection with related bug reports, but to whom and when is this technique needed? Currently, bug reports are consumed with the process of manual selection, assignment, working on code, reviewing, and so on. On the other hand, developers work on code in other development tasks, and they should be familiar with such code. We propose a new bug report processing cycle by asking such developers to solve related bug reports.

As a first step to evaluate the feasibility of our proposal, we conducted an experiment focusing on source files that actually had a relevant (unresolved) bug report, and evaluated the accuracy of recommendation derived from available unresolved bug reports. We used data from three open source software projects, assuming that recommendation takes place every one month for each project. The results show that our technique can provide appropriate bug reports for 21.74% to 60.05% of source codes in the top ten rankings.

II. Related work

Although the links between source code and bug reports are not often recored explicitly in repositories, such linking information is valuable for understanding the context of source code and bugs. For the purpose of mining software repositories (MSR), that is, for further research, several techniques have been proposed to finding such links in the existing source code and bug repositories [12]–[14]. These techniques are executed as batch processes. Since all bug reports and source code are recorded in repositories, various information can be utilized, such as times, authors (developers), commit messages, and other meta-data of bug reports.

In the development phase, such linking information is expected to be beneficial too. IR (information retrieval)-based bug localization returns relevant source code when users give bug reports as queries. Relevant source code can be considered as bug-related source code. Since this task runs with evolving repositories, it can be online processing, that is, the task runs with incomplete bug repositories and comments sometimes without comments. Because of the lacks of complete information, TR (text retrieval) techniques have been proposed for IR-based bug localization. TR techniques are used for finding similar documents in corpus.

Rao and Kak compared several types of TR techniques including unigram model (UM), vector space model (VSM), latent semantic analysis model (LSA), latent Dirichlet allocation (LDA), and cluster based document model (CBDM), and reported that simple text models such as UM and VSM are more effective than other models [15]. Zhou et al. proposed

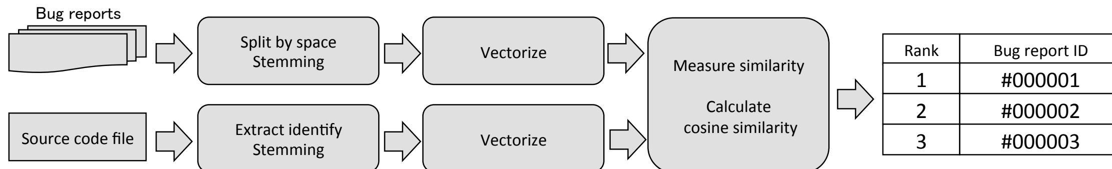

Fig. 1. Overview of bug report recommendation

BugLocator that ranks files based on the text similarity between the given bug report and source code with revised vector space model (rVSM) [16]. The results showed BugLocator can find relevant buggy files for more than 60% Eclipse bugs in the top ten ranking. Tantithamthavorn et al. studied the impact of the granularity on bug localization, and reported that function-level bug localization outperforms class-level bug localization and required 7 times less effort to find the first relevant source code entity [17]. For improving the performance of TR-based bug localization, recent studies made use of various information, such as structured information on code constrtucts [18], segmentation and stack-trace information [19], and API documentation [20].

In this paper, we propose bug report recommendation, which is also an online processing task. Similar to bug localization, it adopts TR techniques to find links between source code and bug reports. Bug report recommendation finds relevant bug reports when a file of source code is given contrary to bug localization. To the best of our knowledge, this is the first study to propose the approach of bug report recommendation.

III. Bug report recommendation

A. Overview

Given a source code file, bug report recommendation ranks bug reports based on the textual similarity between the initial source code and bug reports. Figure 1 shows the overview of our approach. From both bug reports and source code, text information is extracted. We collect titles, descriptions and discussions from bug reports, and split the text by space. Stop words are removed. We perform lexical analysis for each source code file, and collect only lexical tokens. After these preprocessing, both text data are converted to the feature vectors. Similarities are calculated with feature vectors, and the ranks of bug reports are presented in descending order of similarities.

For bug report recommendation, queries of source code files are selected from the latest versions in selected snapshots, and relevant bug reports are searched from the groups of bug reports that has been reported but not been fixed.

B. Similarity Measurement

We use vector space model (VSM) for calculating the similarities between source code and bug reports since VSM is reported to be effective in bug localization. In VSM, each source code file or bug report is represented as an *n*-dimensional vector, where *n* is the number of unique index terms appearing in all the documents (d) and query (q), and w_t is the weight of the *i*-th index term in the vector $< w_1, w_2, ..., w_n >$ defined as follows.

$$w_{t \in d} = tf_{td} \times idf_t = (\log f_{td} + 1) \times \log \frac{\#docs}{n_t}$$

In the above Equation, *tf* refers to the frequency of index term occurrences in a document and if refers to the frequency of index term occurrences over the entire collection of documents. Among many variations of weights, the logarithmic variant was used because it can lead to better performance. A typical formula for *tf* and *idf* are represented as follows.

$$tf(t,d) = \frac{f_{td}}{\#terms}, idf(t) = \log(\frac{\#docs}{n_t})$$

where *t* represents an index term, *d* represents a particular document, f_{td} is the number of term *t* occurs in document *d*. *N* is the total number of documents, and n_t is the number of documents in which term *t* occurs. After transforming source code file and bug reports into vectors, we calculate the degree of similarity between a given source code file and bug report corpus as follows.

$$Similarity(q,d) = cos(q,d) = \frac{\vec{V}_q \cdot \vec{V}_d}{|\vec{V}_q||\vec{V}_d|}$$

With this equation, bug reports with the highest scores are considered as the most textually similar to a given source code file.

IV. Experiment

A. Target Projects

We selected three open-source projects for our study: Eclipse Communication Framework (ecf), Eclipse Pulug-in Development Enviroment UI (eclipse.pde.ui), Eclipse Java Development Tools Core (eclipse.jdt.core). All projects are written in java and have relatively long development histories. Table I presents the number of commits (from 10k to 21k), first commit date (from 2001-5 to 2004-12), and the number of bug report (from 400 to 3k).

TABLE I. Target project

Project	#commit	First commit date	#Bug report
ecf	10786	2004/12/3	1912
eclipse.jdt.core	21055	2001/6/5	51687
eclipse.pde.ui	11270	2001/5/24	14973

B. Data Collection

We collected bug reports from the Eclipse bug tracking system, and obtained source code from their Git repositories. Correct links between source code files and bug reports were identified based on the SZZ algorithm, which is designed to identify bug-introducing commits by mining version control repositories and bug report repositories [12]. Linked source code files were used as the queries of this experiment.

For each month, we collected a set of source code files and bug reports. Source code files were selected at the first snapshot of each month. Selected source code files were buggy files to be fixed later. Candidate bug reports are those that have been reported before but not fixed at the time of the snapshot.

C. Evaluation Metrics

We used the following two metrics to evaluate the effectiveness of our bug report recommendation.

Mean Reciprocal Rank (MRR) is a statistic for evaluating a process that produces a list of possible responses to a query. The reciprocal rank of a query is the multiplicative inverse of the rank of the first correct answer. The mean reciprocal rank is the average of the reciprocal ranks of results of a set of query Q:

$$\mu = \frac{1}{|Q|}[\sum_{q \in Q} \frac{1}{r_q}]$$

Top N Rank is the number of source code files whose associated bug reports are ranked in the top N (N = 1, 5, 10) of the returned results. Given a source code file, if the top N result contains at least one appropriate bug report, we consider the bug report can be recommended. The higher the metric value, the better the bug report recommendation performances.

D. Results

RQ1: How many source code files can be successfully found relevant bug reports with bug report recommendation?

For each source code, the bug report recommendation provide the list of relevant bug reports. We checked the ranks of this list whether there are appropriate bug reports that are fixed bug reports related to the given source code. We performed the experiment for all source code files that will be fixed later, and measured the percentages of source code files that have been successfully recommended.

TABLE II. RESULT OF RQ1

Project	MRR	Top1	Top5	Top10	Median of # Bug report
ecf	0.272	15.90%	38.07%	60.05%	230
eclipse.jdt.core	0.108	6.86%	15.80%	21.74%	5860
eclipse.pde.ui	0.182	10.81%	28.15%	39.19%	812

Table II shows the summary of the results. The difficulties of finding the appropriate bug reports depends on the number of bug reports. For the ecf project, which has relatively small number of bug reports, we could recommend more than 60% of appropriate bug reports in Top 10 lists. For the eclipse.jdt.core project, which has many bug reports to be searched, the performance remained 20% in Top10. The result of the project eclipse.pde.ui was intermediate.

RQ2: How does bug report recommendation perform with different periods?

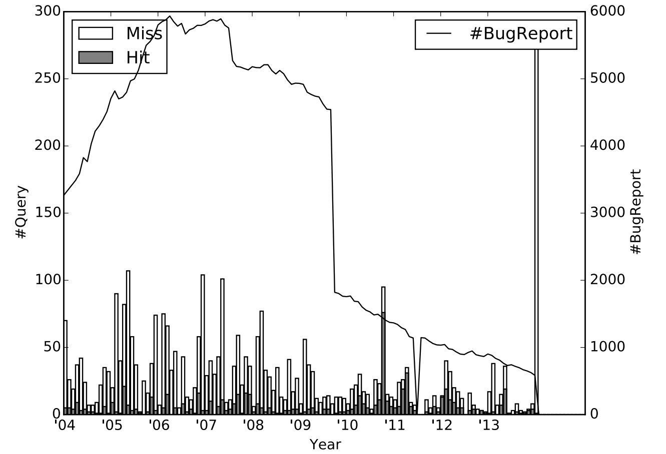

Fig. 2. Top 10 performance of every month (eclipse.jdt.core)

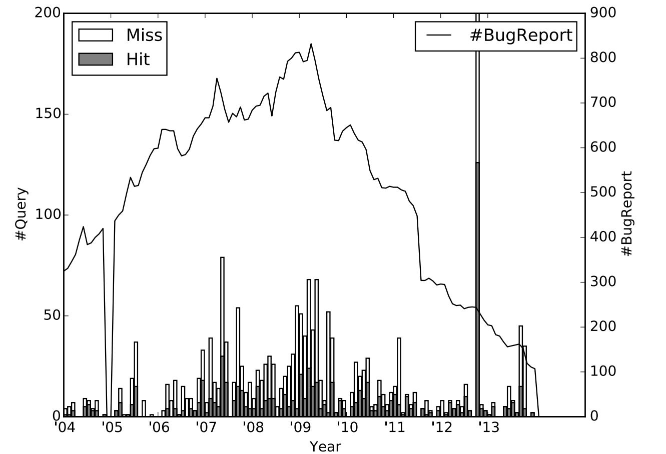

Fig. 3. Top 10 performance of every month (eclipse.pde.ui)

The number of target source code as query and the number of bug reports for search vary depending on the periods. So there is a question how bug report recommendation works in different contexts. To answer this question, we measured the performance of bug report recommendation in every month of research data.

Figure 2 and Figure 3 summarize the performance in each month. The number of source code (queries) is presented as a bar. In the bar, successfully recommended bug reports in top 10 (hit) is represented as gray and fails (miss) are represented as white. In addition, the number of bug reports is plotted as a line. For eclipse.pde.ui, the number of bug reports is almost less than 1,000 in all the periods, and we can see that the ratio of hit is relatively high. On the contrary, for eclipse.jdt.core, the number of bug reports is more than 1,000 in most periods,

and the performance is relatively low. For large-scale project with many bug reports, we need to improve our technique of filtering out irrelevant bug reports.

V. Discussions

A. Threats to Validity

The target projects were limited to open-source software written in Java. For external validity, there is a threat of generalization of our result. Projects we targeted are only open-source software projects written in Java.

Collection of correct bug report information has problems. For construct validity, the main threat is in the phase of collecting bug information.

B. Future Work

Our result suggests that our technique works relatively well. Future work includes evaluating our technique in practical scenario, and improving performance of bug report recommendation by using the similar techniques of recent bug localization, such as structured information on code constrtucts [18], segmentation and stack-trace information [19], and API documentation [20].

VI. Conclusion

This paper proposed a way to recommend to programmers a prioritized list of unresolved bug reports to accelerate bug detection in a given source file. Through an empirical evaluation using data from three open source software projects, we confirmed that recommendation is feasible for all three projects.

The major limitation of our current work is that we put an assumption that there exist at least one unresolved bug report related to the given source file. However, in an actual situation, there is no guarantee that all source files have a related bug report. Therefore, our future work is to automatically classify source files into two groups: one having related bug reports, and the other with no bug reports. If a source file is likely to have no related bug reports, then we just do not recommend anything. Otherwise, we conduct recommendation.

Acknowledgment

This study has been supported by JSPS KAKENHI Grant Number 26540029, and has been conducted as a part of Program for Advancing Strategic International Networks to Accelerate the Circulation of Talented Researchers.

References

[1] L. A. Franz and J. C. Shih, "Estimating the value of inspections and early testing for software projects," *HEWLETT PACKARD JOURNAL*, vol. 45, pp. 60–60, 1994.

[2] J. C. Kelly, J. S. Sherif, and J. Hops, "An analysis of defect densities found during software inspections," *J. Syst. Softw.*, vol. 17, no. 2, pp. 111–117, Feb. 1992. [Online]. Available: http://dx.doi.org/10.1016/0164-1212(92)90089-3

[3] E. F. Weller, "Lessons from three years of inspection data," *IEEE Softw.*, vol. 10, no. 5, pp. 38–45, Sep. 1993. [Online]. Available: http://dx.doi.org/10.1109/52.232397

[4] R. L. Glass, "Practical programmer: Inspections—some surprising findings," *Commun. ACM*, vol. 42, no. 4, pp. 17–19, Apr. 1999. [Online]. Available: http://doi.acm.org/10.1145/299157.299161

[5] V. Basili, S. Green, O. Laitenberger, F. Lanubile, F. Shull, S. Srumgrd, and M. Zelkowitz, "The empirical investigation of perspective-based reading," *Empirical Software Engineering*, vol. 1, no. 2, pp. 133–164, 1996. [Online]. Available: http://dx.doi.org/10.1007/BF00368702

[6] T. Thelin, P. Runeson, and C. Wohlin, "An experimental comparison of usage-based and checklist-based reading," *IEEE Trans. Softw. Eng.*, vol. 29, no. 8, pp. 687–704, Aug. 2003. [Online]. Available: http://dx.doi.org/10.1109/TSE.2003.1223644

[7] M. Halling, S. Biffl, T. Grechenig, and M. Kohle, "Using reading techniques to focus inspection performance," in *Proc. of 27th Euromicro Conf.*, 2001, pp. 248–257.

[8] V. Mashayekhi, J. M. Drake, W.-T. Tsai, and J. Riedl, "Distributed, collaborative software inspection," *IEEE Softw.*, vol. 10, no. 5, pp. 66–75, Sep. 1993. [Online]. Available: http://dx.doi.org/10.1109/52.232404

[9] L. Brothers, V. Sembugamoorthy, and M. Muller, "Icicle: Groupware for code inspection," in *Proc. of 1990 ACM Conf. on Comput. supported Cooperative Work*, ser. CSCW '90. New York, NY, USA: ACM, 1990, pp. 169–181. [Online]. Available: http://doi.acm.org/10.1145/99332.99353

[10] J. C. Knight and E. A. Myers, "An improved inspection technique," *Commun. ACM*, vol. 36, no. 11, pp. 51–61, Nov. 1993. [Online]. Available: http://doi.acm.org/10.1145/163359.163366

[11] K. El Emam, O. Laitenberger, and T. Harbich, "The application of subjective estimates of effectiveness to controlling software inspections," *J. Syst. Softw.*, vol. 54, no. 2, pp. 119–136, Oct. 2000. [Online]. Available: http://dx.doi.org/10.1016/S0164-1212(00)00032-7

[12] J. Śliwerski, T. Zimmermann, and A. Zeller, "When do changes induce fixes?" in *Proc. of 2nd Int. Workshop on Mining Softw. Repositories*, ser. MSR '05. New York, NY, USA: ACM, 2005, pp. 1–5. [Online]. Available: http://doi.acm.org/10.1145/1082983.1083147

[13] R. Wu, H. Zhang, S. Kim, and S.-C. Cheung, "ReLink: recovering links between bugs and changes," in *Proc. of 8th Joint Meeting of the European Softw. Eng. Conf. and the ACM SIGSOFT Symp. on the Found. of Softw. Eng.*, ser. ESEC/FSE '11. New York, NY, USA: ACM, 2011, pp. 15–25. [Online]. Available: http://doi.acm.org/10.1145/2025113.2025120

[14] A. T. Nguyen, T. T. Nguyen, H. A. Nguyen, and T. N. Nguyen, "Multi-layered approach for recovering links between bug reports and fixes," in *Proc. of 20th ACM SIGSOFT Int. Symp. on Found. of Softw. Eng.*, ser. FSE '12. New York, NY, USA: ACM, 2012, pp. 63:1–63:11. [Online]. Available: http://doi.acm.org/10.1145/2393596.2393671

[15] S. Rao and A. Kak, "Retrieval from software libraries for bug localization: A comparative study of generic and composite text models," in *Proc. of 8th Work. Conf. on Mining Softw. Repositories*, ser. MSR '11. New York, NY, USA: ACM, 2011, pp. 43–52. [Online]. Available: http://doi.acm.org/10.1145/1985441.1985451

[16] J. Zhou, H. Zhang, and D. Lo, "Where should the bugs be fixed? - more accurate information retrieval-based bug localization based on bug reports," in *Proc. of 34th Int. Conf. on Softw. Eng.*, ser. ICSE '12. Piscataway, NJ, USA: IEEE Press, 2012, pp. 14–24. [Online]. Available: http://dl.acm.org/citation.cfm?id=2337223.2337226

[17] C. Tantithamthavorn, A. Ihara, H. Hata, and K. Matsumoto, "Impact analysis of granularity levels on feature location technique," in *Proc. of 1st Asia Pacific Requirements Engineering Symposium*, ser. APRES '14, 4 2014, pp. 135–149.

[18] R. Saha, M. Lease, S. Khurshid, and D. Perry, "Improving bug localization using structured information retrieval," in *Proc. of 28th IEEE/ACM Int. Conf. on Automated Softw. Eng.*, Nov 2013, pp. 345–355.

[19] C. Wong, Y. Xiong, H. Zhang, D. Hao, L. Zhang, and H. Mei, "Boosting bug-report-oriented fault localization with segmentation and stack-trace analysis," in *Proc. of 30th IEEE Int. Conf. on Softw. Maintenance and Evolution*, ser. ICSME '14, October 2014, pp. 181–190.

[20] X. Ye, R. Bunescu, and C. Liu, "Learning to rank relevant files for bug reports using domain knowledge," in *Proc. of 22nd ACM SIGSOFT Int. Symp. on Found. of Softw. Eng.*, ser. FSE 2014. New York, NY, USA: ACM, 2014, pp. 689–699. [Online]. Available: http://doi.acm.org/10.1145/2635868.2635874