Mining A Change History to Quickly Identify Bug Locations : A Case Study of the Eclipse Project

Chakkrit Tantithamthavorn[†], Rattamont Teekavanich[‡], Akinori Ihara[†], Ken-ichi Matsumoto[†]

[†]Graduate School of Information Science,
Nara Institute of Science and Technology, Japan.
{chakkrit-t,akinori-i,matumoto}@is.naist.jp

[‡]Department of Computer Engineering,
Faculty of Engineering, Kasetsart University, Thailand.
b5310546537@ku.ac.th

***Abstract*—In this study, we proposed an approach to mine a change history to improve the bug localization performance. The key idea is that a recently fixed file may be fixed in the near future. We used a combination of textual feature and mining the change history to recommend source code files that are likely to be fixed for a given bug report. First, we adopted the Vector Space Model (VSM) to find relevant source code files that are textually similar to the bug report. Second, we analyzed the change history to identify previously fixed files. We then estimated the fault proneness of these files. Finally, we combined the two scores, from textual similarity and fault proneness, for every source code file. We then recommend developers examine source code files with higher scores. We evaluated our approach based on 1,212 bug reports from the Eclipse Platform and Eclipse JDT. The experimental results show that our proposed approach can improve the bug localization performance and effectively identify buggy files.**

***Index Terms*—Software Debugging, Bug Localization, Mining Change History, Information Retrieval**

I. Introduction

In the software debugging process, finding a bug in a large and complex software project is time-consuming and painstaking. A developer requires a deep understanding of the software structure to manually locate suspicious entities based on a bug report in an issue tracking system such as BugZilla. To help developers find those entities quickly, automated bug localization techniques have been proposed and received much attention in the last few years.

Automated bug localization refers to a process of finding suspicious entities based on a bug report. Several approaches [1]–[4] have been built around modern information retrieval (IR) to identify source code files that are textually similar to a given bug report. However, the accuracy of these IR-based technique is far from perfect.

Recently, much research has shown promise that using additional sources of information can significantly improve the performance of IR-based techniques. Several studies [2]–[4] have exploited past bug report information. Intuitively, similar bugs tend to need fixes in similar files. While these studies do an adequate job on the bug report history, they do not make use of another wealth of stored information in the form of *the Change History* [5].

In this paper, we present an approach to using the change history of a software project to improve the improve IR-based bug localization. As a baseline IR model for our study, we use the Vector Space Model (VSM) because VSM is the most effective IR model [6]. In our approach, we also analyze the change history to identify previously fixed files. The key idea is that a recently fixed file may also need to be fixed in the near future. Inspired by a previous study [7], we use the notion of a "*Cache*" to hold the list of previously fixed files in chronological order. We then estimated the fault proneness of these files. Finally, for every source code file, we combine the similarity scores and the fault proneness scores. We then recommend the source code files with the higher scores to developers as likely places to correct the bugs.

In our evaluation of our approach, we used 1,212 bug reports from two open source software projects, the Eclipse Platform and the Eclipse JDT. To validate our approach, we investigated these three research questions: **RQ1:** What is the performance of our approach? **RQ2:** Does our approach improve the bug localization performance? **RQ3:** Does our approach effectively identify buggy files?

The main contributions of this paper are:

- We propose an approach to mine a change history to improve the bug localization performance.
- The experimental results show that our approach can improve the bug localization performance and more effectively identify buggy files.

II. Background

In this section, we describe the background behind our approach. First, we introduce the background of mining change histories. Second, we introduce the workflow of IR-based bug localization using the Vector Space Model (VSM).

A. Mining Change History

The change history of a software system refers to the history of all changes that have been made to the software [5]. The

978-1-4799-2552-0/13/$31.00 ©2013 IEEE

change history can be acquired from version control tools. Previous studies [7]–[9] have demonstrated that the change history stores a wealth of information that could potentially be used to predict the future fault proneness of software entities such as classes, files, methods, and so on. For example, Sliwerski et al. [8] showed that fixed entities are likely to induce a later change. Hassan and Holt [7] used the number of recently modified and fixed files to predict subsystems susceptible to faults. Kim et al. [9] also pointed out that previously fixed entities are a good indicator to predict near future faults.

These findings inspired us to mine the change history incorporating previously fixed files to improve the accuracy of existing IR-based bug localization approach. Intuitively, previously fixed files are likely to need to be fixed again soon.

B. Workflow of IR-based Bug Localization using The Vector Space Model

In traditional information retrieval, the Vector Space Model (VSM) is a widely used technique. Several approaches [1], [3] have adapt VSM to 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 queries (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:

$$w_{t \in d} = tf_{td} \times idf_t \tag{1}$$

In Equation 1, tf refers to the frequency of index term occurrences in a document and idf 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 shown in Equation 2.

$$\begin{aligned} tf(t,d) &= log(f_{td}) + 1 \\ idf(t) &= log(\frac{N}{1+n_t}) \end{aligned} \tag{2}$$

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 files and bug reports into vectors, we calculate the degree of similarity between a given bug report and source code corpus as shown in Equation 3.

$$Similarity(q,d) = \frac{\overrightarrow{V_q} \bullet \overrightarrow{V_d}}{|\overrightarrow{V_q}||\overrightarrow{V_d}|} = \frac{\sum_{i=1}^{n} (w_{t \in q} \times w_{t \in d})}{\sqrt{\sum_{i=1}^{n} w_{t \in q}^2 \times \sum_{i=1}^{n} w_{t \in d}^2}} \tag{3}$$

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

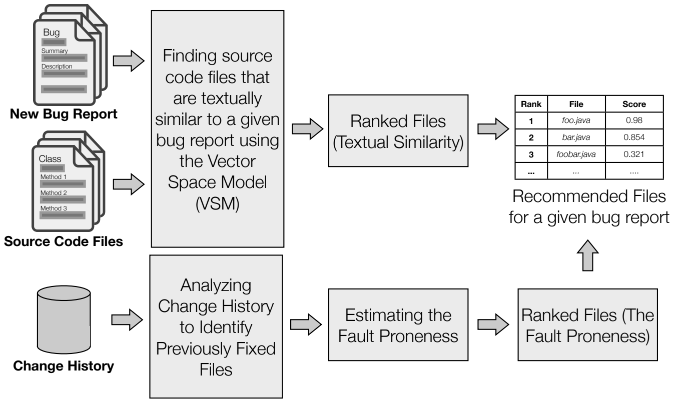
Bug
Summary
Description
New Bug Report
Class
Method 1
Method 2
Method 3
Source Code Files
Change History
Finding source code files that are textually similar to a given bug report using the Vector Space Model (VSM)
Analyzing Change History to Identify Previously Fixed Files
Ranked Files (Textual Similarity)
Estimating the Fault Proneness
Rank
File
Score
1
foo.java
0.98
2
bar.java
0.854
3
foobar.java
0.321
...
...
....
Recommended Files for a given bug report
Ranked Files (The Fault Proneness)

Fig. 1. Overview of our proposed approach

III. Our Proposed Approach

Figure 1 shows the overview of our proposed approach. For a new bug report, we adopted the Vector Space Model (VSM) to find source code files that are textually similar to the title and description of the bug report. We analyzed the change history to identify previously fixed files. We then estimated the fault proneness of these files. Finally, we combined the scores and recommended source code files with higher score to developers as most likely to contain bugs. We describe the details of this approach in the following subsections.

A. Finding Textual Similar Files

Our approach adopted VSM to find source code files that are textually similar to the bug report. First, we extracted semantic words from bug reports and source code files. We removed all punctuations and digits. We then split all words into tokens and normalized them by transforming to lower case. For multiple-word identifiers e.g. *GetInitialValue()*, we did not separate them into single words to retain the original meaning. We also removed common English words (e.g. a, an, the) and general programming language words (e.g. int, double, char). Second, we obtained one vector for every bug reports and source code files by applying Equation 1 and 2. Third, we used Equation 3 to calculate $VSMScore(q,d)$ for every bug report and source code file. Finally, we returned the ranked source code files for every bug report.

B. Mining Change History

1) Analyzing Change History to Identify Previously Fixed Files: Generally, developers commonly include a bug report number in the comment of commit logs whenever they fix a bug associated with it [8]. To identify previously fixed files, we used logical mappings between the commit logs of the software version archives and the fixed bug reports (see Figure 2). This allowed us to obtain a list of previously fixed files in chronological order.

Inspired by a previous study [7], we used the notion of a "*Cache*" to hold the potential files for these faults. The cache is dynamically updated as the software system evolves.

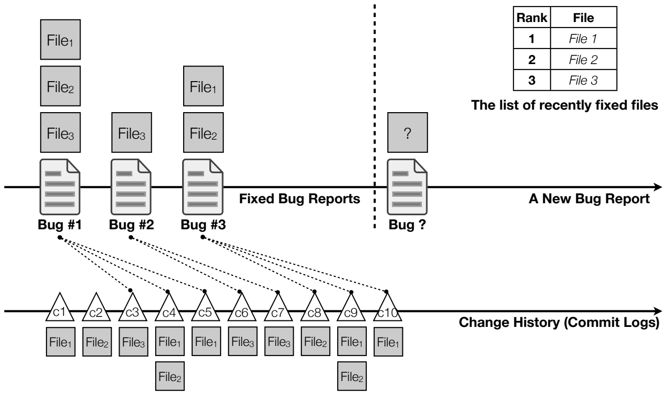

Fig. 2. An illustration of change history analysis to identify previously fixed files

A *hit* occurs when a file in the cache undergoes a bug fix; a *miss* occurs when a file outside the cache receives a bug fix. Intuitively, when a file hits in the cache, the file will be automatically moved to the top of the cache. We used the cache size to determine the upper bounds on how many files were loaded in the cache. When the cache is full, the least recently used file is removed from the cache. In this research, we called this the cache-based approach.

2) Estimating Fault Proneness: Previous studies have shown promise that previously fixed files are more likely to need to be fixed in the near future [7]–[9]. From our observation, we found that recently fixed files are likely to need to be fixed again. Therefore, we should rank recently fixed files higher in the case of bug localization. We used the logistic function for estimating the fault proneness for every file (f) corresponding to the rank in the cache (C). This allowed us to ensure that higher ranks are given higher scores during ranking. The function is defined as follows:

$$FaultProneness(f) = \frac{1}{1 + e^{Norm(rank_f)}} \quad ;\forall f \in C \quad (4)$$

We computed the $FaultProneness$ value for each file according to the rank number. In Equation 4, we use the normalized value of $rank_f$ as the input to the exponential function e^x. The normalization function is defined as follows:

$$Norm(rank_f) = \frac{rank_f - \mu}{\sigma} \quad (5)$$

where μ is the average rank of buggy files that hit in the cache, and σ is the standard deviation of the rank of buggy files that hit in the cache.

C. Combining Ranks

After calculated the scores obtained from querying similar source code files ($VSMScore$) and from the change history analysis ($FaultProneness$), we combined these two scores for each file as follows:

$$FinalScore(q,d) = VSMScore(q,d) + FaultProneness(d) \quad (6)$$

Files that are ranked higher are the more relevant ones, i.e., more likely to contain bugs.

IV. Experimental Design

In our evaluation, we addressed the following research questions:

RQ1: What is the performance of our approach?

RQ2: Does our approach improve the bug localization performance?

RQ3: Does our approach effectively identify buggy files?

We used two open source software projects, from the Eclipse Platform and the Eclipse JDT, as shown in Table I.

TABLE I
Statistics Summary of Studied Project

Project	Study Period	# Bugs	# Files
Eclipse Platform	Apr 2002 - Jan 2013	744	1,758
Eclipse JDT	Jun 2002 - Mar 2013	468	4,222

A. Dataset Collection and Ground-Truth Data Preparation

We obtain bug report information from the Eclipse bug tracking system. We selected only bug reports which labeled as "FIXED". We excluded all false-positive bug numbers. We obtained source code information from the Git version control system. We took snapshots of each system's source code at six month intervals over the duration of the system. During evaluation, given a bug report, we determine the previous nearest snapshot and perform our approach to obtain a ranked list of source code files.

To prepare ground-truth data, we identified changes from the commit logs using the SZZ algorithm [8]. The algorithm parses the commit log messages from the Git source code repository, looking for messages such as "Fixed Bug #137088" or similar variations. If found, the algorithm establishes a link between all the source code files in the commit transaction with the identified bug ID. The result is a set of links between bug reports and source code entities, which we use to evaluate our approach.

B. Evaluation Metrics

To evaluate the performance of our approach compared to traditional bug localization methods, we use the following metric:

1) Top-N Rank Accuracy: This metric measure the accuracy by calculating the percentage of bug reports that have at least one buggy files that was returned from the top N rank result (N) = 1, 5, 10, 20, and 30). This accuracy is calculated by Equation 7.

TABLE II
SUMMARY OF THE PERFORMANCE OF OUR APPROACH COMPARED TO VSM APPROACH AND CACHE-BASED APPROACH ALONE

Project	Approach	Top 1	Top 5	Top 10	Top 20	Top 30	MRR	MAP
Eclipse Platform	**Our Approach**	**14.25%**	**40.46%**	**53.36%**	**64.92%**	**71.77%**	**0.2692**	**0.2394**
	Vector Space Model Approach	12.10%	32.39%	43.15%	55.38%	63.58%	0.2247	0.1890
	Cache-based Approach	11.56%	23.66%	31.18%	44.62%	53.36%	0.1810	0.1762
Eclipse JDT	**Our Approach**	**11.97%**	**32.91%**	**44.66%**	**53.42%**	**58.76%**	**0.2208**	**0.1611**
	Vector Space Model Approach	6.17%	19.23%	26.70%	35.90%	41.45%	0.1327	0.0940
	Cache-based Approach	9.19%	25.00%	35.04%	45.73%	54.49%	0.1680	0.1573

$$Accuracy(C_j) = \frac{1}{|Q|}\sum_{i=1}^{|Q|} F(q_i) \tag{7}$$

where $|Q|$ is the total number of queries (Bug reports) and q_i represents for each query. $F(q_i)$ is a function for each query, such that if at least one buggy files was appeared in the top N rank this function will return 1 and otherwise will return 0.

2) MRR (Mean Reciprocal Rank): is a statistical measure for evaluating any process that produces a list of possible responses to a query. The reciprocal rank of a query response 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 queries Q:

$$MRR = \frac{1}{|Q|}\sum_{i=1}^{|Q|} \frac{1}{rank_i} \tag{8}$$

3) MAP (Mean Average Precision): is a measure to evaluate performance of bug localization that is most commonly used in research papers. MAP calculates the mean of the average precision scores for each query (Bug report), when one query may have more than one relevant document (Buggy file). The Average Precision for one query ($AvgP$) can be computed using Equation 9.

$$AvgP(q_i) = \frac{\sum_{k=1}^{n}(P(k) \times Rel(k))}{number\ of\ relevant\ documents} \tag{9}$$

where k is the rank, n is the number of retrieved documents (number of bugs from ranked result), $Rel(k$ returns 1 if the retrieved document is a relevant document, 0 otherwise. $P(k)$ is the precision at the given cut-off rank j. After that MAP can be calculated as follows:

$$MAP = \frac{1}{|Q|}\sum_{i=1}^{|Q|} AvgP(q_i) \tag{10}$$

A higher MAP means that for each query, the bug localization technique can return relevant documents at a higher rank.

4) Average Coverage Ratio: This metric measure the percentage number of relevant documents that a bug localization method can returns from the result. Because of one query may have more than one relevant document, $CoverageRatio$ is the percentage for one query as calculated by Equation 11.

$$CoverageRatio(\%) = \frac{\#SuccessfullyLocalizedBugs}{\#TotalBugs} \tag{11}$$

Finally, we then calculated the average of $CoverageRatio$ for all queries Q:

$$AverageCoverageRatio(\%) = \frac{1}{|Q|}\sum_{i=1}^{|Q|} CoverageRatio_i \tag{12}$$

V. EXPERIMENTAL RESULTS

This section reports the experimental results. We report the performance of our approach (RQ1) in the first section and compare the results with the Vector Space Model approach and the cache-based approach (RQ2) in the second section. Finally, we show the effectiveness at identifying buggy files of our approach (RQ3).

A. Performance

We first addressed **RQ1:** What is the performance of our approach? Table II shows the Top-N Accuracy, MRR and MAP values respectively. Since our approach recommends the Top N files, the performance of our approach depends on the value of N. Overall, the value of Top-N Accuracy grows as N increases.

For the Eclipse Platform, there are total of 744 bug reports. As Table II shows in bold, our approach successfully located relevant files for 106 bug reports (14.25%) in the Top 1 and 534 bug reports (71.77%) in the Top 30 of the returned results. Similarly, the Eclipse JDT had 468 bug reports total, and our approach successfully located relevant files for 56 bug reports (11.97%) in the Top 1 and 275 (58.76%) in the Top 30 of the returned results.

These results indicate that for a large percentage of bugs, our approach can identify a small number of source files that need to be examined.

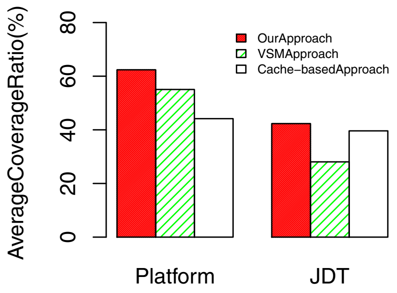

Fig. 3. The effectiveness of identifying buggy files

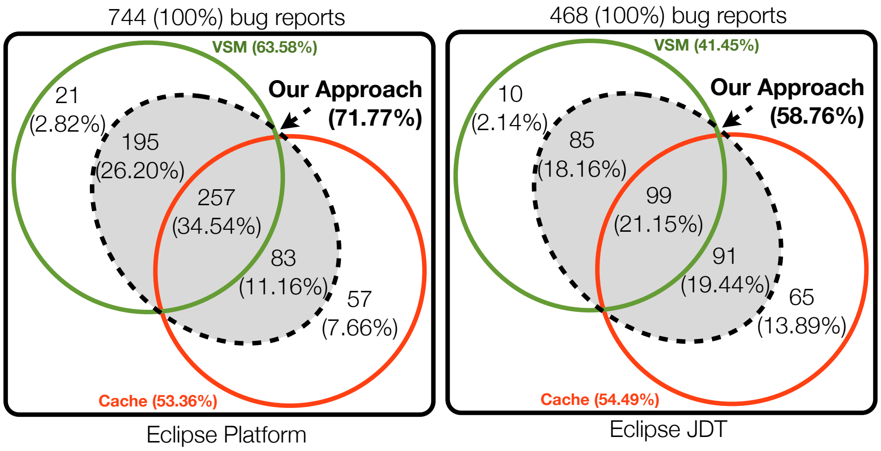

Fig. 4. A comparison of variations in the returned results of the approaches (Top 30 accuracy)

B. Comparison

We compared the results of our approach with the VSM and cache-based approaches to address **RQ2:** Does our approach improve the bug localization performance? Table II shows the summary results. For fair comparisons, we used the same comparative size of Top N files. The experimental results show that our approach outperformed the VSM approach and cache-based approaches. For example, for the Eclipse Platform, when considering at the Top 30 accuracy, of the total bug reports, our approach can successfully localized 71.77%, the VSM approach 63.58%, and the cache-based approach only 53.36%. Similarly, for the Eclipse JDT, when considered at the Top 30 accuracy, of the total bug reports, our approach can successfully localized 58.76%, the VSM approach 41.45%, and the cache-based approach only 54.49%. In the same way, the MRR and MAP values of our approach are also higher than the other two approaches.

C. Effectiveness at identifying buggy files

We calculated $AverageCoverageRatio$ to address **RQ3:** Does our approach effectively identify buggy files? We computed this for every subject system and for every approach and plotted the comparative results as shown in Figure 3. The x-axis indicates the $AverageCoverageRatio$ values. For the Eclipse Platform, our approach successfully identified 62.38%, the VSM approach 55.05%, and the cache-based approach 44.17% of the relevant buggy files. Similarly, for the Eclipse JDT, our approach successfully identified 42.33%, the VSM approach 28.05%, and the cache-based approach 39.61% of the relevant buggy files. Overall, the completeness from our approach is better than the other two approaches.

VI. Discussions

A. What are the variations in the returned results of the various approaches?

In this section, we performed an analysis to address the question: what are the variations in the returned results of the various approaches? As shown in Figure 4, we used a Venn diagram to show the variations in the returned results at Top 30 Accuracy. For the Eclipse Platform, our approach localized 71.77%, VSM 63.58%, and cache-based 53.36% of the total bug reports. As the figure shows, the VSM and cache-based approaches shared 257 bug reports (34.54%), the overlapping region in the middle. While our approach localized a larger percentage than either VSM or cache-based approaches, there were small subsets that the other approaches localized which were not localized by our approach. Specifically, 21 bug reports (2.82%) were identified by VSM but not by our approach, and 57 (7.66%) were identified by the cache-based approach but not by our approach. Similarly, for the Eclipse JDT, our approach localized 58.76%, VSM 41.45%, and cache-based 54.49% of the total bug reports. the VSM and cache-based approaches shared 99 bug reports (21.15%). The VSM approach identified 10 bug reports (2.14%) and the cache-based approach identified 65 (13.89%) not identified by our approach at the Top 30 Accuracy level.

Based on the results, we concluded that our method of combining the scores of the VSM and cache-based approaches has some negative effects because our approach does not completely cover all bugs localized by the VSM and cache-based approaches. Therefore, the change history mining technique can be further improved.

B. Why can our approach improve the performance of IR-based bug localization?

As shown in Figure 4, there were a large number of bug reports that are not shared between the VSM and cache-based approaches. Our approach takes advantage of this finding. In the proposed approach, Equation 6 combines rankings from the two models. With this combination, our approach can localize bugs more accurate and more effective than the VSM approach and cache-based approach alone. The experimental results also help us confirm that by incorporating textual features and the change history of previously fixed files can significantly improve the performance of IR-based bug localization.

C. Threats to Validity

There are several potential threats to the validity of our work, including:

- Our ground-truth data relies on the SZZ technique [8] that can bias the results of our approach. This is because there is a linking bias in identifying bugs with revision logs and bug reports [10].
- In this experiment, we used only two systems from the Eclipse Project, so our results may not be generalized to other software projects. More experiments are required to obtain more reliable results.

VII. Related Works

Debugging a failure in a large and complex program in current software projects is cost intensive. There is much research on proposed automated bug localization approaches to help developers quickly locate the bug locations based on a given bug report [1]–[4]. These approaches have been built on modern information retrieval (IR) to identify source code files that are textually similar to a given bug report. Rao et al. [1] compared several generic and composite IR models. Their results showed that VSM is the most effective model among others. However, the accuracy of IR-based techniques is far from perfect.

Several studies have shown promise that using additional source of information can significantly improve the performance of IR-based bug localization. Nichols et al. [2] used information from past bug reports to improve the latent semantic indexing (LSI) model. Zhou et al. [3] proposed an approach to using past similar bug reports to improve bug localization performance. The intuitive idea of their approach is that similar bugs tend to be fixed in similar files. In the same year, Davies et al. [4] also exploited similar bug reports to identify buggy methods. In contrast, our approach mined the change history to improve bug localization performance.

The mining of the change history has been widely used in several aspects. Several related studies in the literature of software defect prediction worked to predict fault incidence [11], to detect logical couplings [12], and to predict change entities in the future [13]. In contrast, our approach exploited the change history as an aspect of bug localization research.

Buckley et al. [5] have extensively studied the taxonomy of software change. Several dimensions of software changes have been introduced, such as, temporal properties, object of change, system properties, and change support. In this research, we considered only the temporal properties of previously fixed files. Although the results of our approach achieved an accuracy of up to 71.77% at the Top 30, in our future work, we will include other aspects of change history.

VIII. Conclusion and Future Works

In this paper, we proposed a way to use the change history of a software project to improve the performance of IR-based bug localization. We used the Vector Space Model (VSM) as a baseline IR model of our study. We analyzed the change history to identify previously fixed files. Intuitively, a recently fixed file might need to be fixed in the near future. We then estimated the fault proneness of these files. Finally, for every source code file, we combined the similarity scores and the fault proneness scores. We then recommended source code files with higher scores to developers.

In our evaluation of this approach, we used a number of traditional IR-based metrics such as Top-N Accuracy, MRR, and MAP. Based on our datasets, the experimental results showed that our approach performed better than the VSM approach and cached-based approach alone. We also studied the effectiveness of our approach to in identifying bug locations. The results showed that our approach effectively identified more bug locations than the compared approaches. These results indicate that a system using our approach can help developers quickly identify possible bug locations to examine based on bug reports. Such a system would allow developers to spend less time locating bugs and more time fixing them. This means more bugs can be fixed at the same amount of maintenance effort and cost.

Based on this study, future research will focus on validating our approach by doing extensive experiments on other large software projects.

Acknowledgment

We would like to thank Prof. Mike Barker from NAIST and anonymous reviewers for their valuable comments. Thanks to the internship program between Kasetsart University and NAIST. This research was conducted as part of Grant-in-Aid for Young Scientists (B), 25730045 and for Scientific Research (B) 23300009 by Japan Society for the Promotion of Science (JSPS).

References

[1] S. Rao and A. Kak, "Retrieval from Software Libraries for Bug Localization : A Comparative Study of Generic and Composite Text Models," in *MSR'11*, 2011, pp. 43–52.

[2] B. D. Nichols, "Augmented Bug Localization Using Past Bug Information," in *ACM SE'10*, 2010, pp. 61:1–61:6.

[3] J. Zhou, H. Zhang, and D. Lo, "Where Should the Bugs Be Fixed ?" in *ICSE'12*, 2012, pp. 14–24.

[4] S. Davies, M. Roper, and M. Wood, "Using Bug Report Similarity to Enhance Bug Localisation," in *WCRE'12*, 2012, pp. 125–134.

[5] J. Buckley, T. Mens, M. Zenger, A. Rashid, and G. Kniesel, "Towards a taxonomy of software change," *J of Softw Maint and Evol: Research and Practice*, vol. 17, no. 5, pp. 309–332, Sep. 2005.

[6] S. W. Thomas, M. Nagappan, D. Blostein, and A. E. Hassan, "The Impact of Classifier Configuration and Classifier Combination on Bug Localization," *IEEE Trans. Softw. Eng*, pp. 1–20, 2013.

[7] A. E. Hassan and R. C. Holt, "The top ten list: dynamic fault prediction," *ICSM'05*, pp. 263–272, 2005.

[8] J. Śliwerski, T. Zimmermann, and A. Zeller, "When do changes induce fixes?" *ACM SIGSOFT Softw. Eng. Notes*, vol. 30, no. 4, p. 1, Jul. 2005.

[9] S. Kim, T. Zimmermann, E. J. Whitehead Jr., and A. Zeller, "Predicting Faults from Cached History," in *ICSE'07*, 2007, pp. 489–498.

[10] C. Bird, A. Bachmann, E. Aune, J. Duffy, A. Bernstein, V. Filkov, and P. Devanbu, "Fair and Balanced ? Bias in Bug-Fix Datasets Categories and Subject Descriptors," in *FSE'09*, 2009, pp. 121–130.

[11] T. L. Graves, A. F. Karr, J. S. Marron, and H. Siy, "Predicting fault incidence using software change history," *IEEE Trans. Softw. Eng*, vol. 26, no. 7, pp. 653–661, 2000.

[12] H. Gall, M. Jazayeri, and J. Krajewski, "CVS Release History Data for Detecting Logical Couplings," in *IWPSE'03*, 2003, p. 13.

[13] T. Zimmermann, P. Weiß gerber, S. Diehl, and A. Zeller, "Mining Version Histories to Guide Software Changes," in *ICSE'04*, 2004, pp. 563–572.