2012 Fourth International Workshop on Empirical Software Engineering in Practice

Locating Source Code to be Fixed based on Initial Bug Reports -A Case Study on the Eclipse Project-

Phiradet Bangcharoensap*, Akinori Ihara†, Yasutaka Kamei‡, Ken-ichi Matsumoto†
* *Dept. of Computer Engineering, Faculty of Engineering, Kasetsart University*
Email: b521050134@ku.ac.th
† *Graduate School of Information Science, Nara Institue of Science and Technology*
Email: (akinori-i, matumoto)@is.naist.jp
‡ *Graduate School and Faculty of Information Science and Electrical Engineering, Kyushu University*
Email: kamei@ait.kyushu-u.ac.jp

***Abstract*—In most software development, a Bug Tracking System is used to improve software quality. Based on bug reports managed by the bug tracking system, triagers who assign a bug to fixers and fixers need to pinpoint buggy files that should be fixed. However if triagers do not know the details of the buggy file, it is difficult to select an appropriate fixer. If fixers can identify the buggy files, they can fix the bug in a short time. In this paper, we propose a method to quickly locate the buggy file in a source code repository using 3 approaches, text mining, code mining, and change history mining to rank files that may be causing bugs. (1) The text mining approach ranks files based on the textual similarity between a bug report and source code. (2) The code mining approach ranks files based on prediction of the fault-prone module using source code product metrics. (3) The change history mining approach ranks files based on prediction of the fault-prone module using change process metrics. Using Eclipse platform project data, our proposed model gains around 20% in TOP1 prediction. This result means that the buggy files are ranked first in 20% of bug reports. Furthermore, bug reports that consist of a short description and many specific words easily identify and locate the buggy file.**

***Keywords*-Bug localization; Text mining; Code mining; Change history mining;**

I. Introduction

Most open source software projects provide a Bug Tracking System (BTS) to unify management of bugs reported by developers and users in their projects. This system is important to improve software quality. When developers and users detect a bug, they report their situation (e.g. the kind of PC/OS, function name that caused the bug, and bug reproduction method) in a bug report to the BTS managed by the project. OSS developers (triagers and fixers) then locate the buggy files that should be fixed by reference to the bug reports in the BTS.

Triagers should assign the bug to an appropriate fixer based on the initial bug report. However, because triagers are not always experts with a buggy file, it is difficult to assign an appropriate fixer every time [6]. As the result the fixer often needs to identify the location of buggy files from their source code repository, even if she/he is not familiar with the bug report. In large open source software projects with from several dozen to several hundred bug reports filed on a daily basis, developers often take a long time to identify appropriate triagers and fixers to perform fixes.

In this paper, we propose a method to identify the location of buggy files from a source code repository based on an initial bug report. This study uses 3 approaches, text mining, code mining, and change history mining. (1) The text mining approach ranks files based on textual similarity using Term Frequency-Inverse Document Frequency (TF-IDF) between a bug report and source code. (2) The code mining approach ranks files that may be causing bugs based on the output of a bug prediction model [7] using source code product metrics. (3) The change history approach ranks buggy files that may be causing bugs based on the output of the bug prediction model using the change process metrics. Using Eclipse project data, we answer the following research question.

RQ: *How accurately can our proposed algorithm identify the source code that should be fixed?*

In our experiment, we describe the difference in accuracy between approaches. However, the accuracy of our approach will be affected by the documents of a bug report. In this study, we describe the difference in accuracy produced by the features of documents in bug reports.

This paper is organized as follows. Section II describes related work and background of this study. Section III describes the details of our proposed approach. Section IV describes the design of our experiments. Section V describes the way we answered the research question and the answer. Section VI discusses our experimental results. Finally, section VIII concludes this paper and presents our future work.

II. Related Works

There are many studies of bug fixing processes with the BTS in large software projects. Some of them have proposed methods to automatically identify and locate buggy files

978-0-7695-4866-1/12 $26.00 © 2012 IEEE
DOI 10.1109/IWESEP.2012.14

based on bug reports to improve the process from bug report to the start of the bug fix[1][9]. Rao et al. [9] applied five information retrieval models, the Vector Space Model (VSM), Unigram Model (UM), Latent Semantic Analysis Model (LSA), Latent Dirichlet Allocation Model (LDA), and Cluster Based Document Model (CBDM), on 369 bugs from ASPECTJ. Zhou et al. [12] revised the Vector Space Model by using the length of a description in a bug report and taking similar bug information into account. Their approach identified 29.14% of the buggy files. The accuracy of these approaches differed according to features of the bug report document (e.g., the length of the description or specific word to identify buggy code). In this study, we analyze the effect of both of the number of words and the number of specific words in a bug report description on the bug localization studies. As part of achieving the goal of our study, we will also help follow up these existing research studies.

How to make a good bug report: The most helpful information for developers is different with the information provided by reporters. According to Bettenburg et al. [2], their analysis, a test cases and a code examples are the most helpful information for developers. When bug reporters submit these, developers can easily start fixing bugs. However, reporters actually provide only the product name and version. This study will give some feedback to how to easily understand a bug report description by the analysis of accuracy produced in each the features of documents in bug reports.

How to assign an appropriate fixer: To assign a bug fix to appropriate fixers, triagers need experience with the reported buggy module. However, because triagers do not always have this experience, triagers sometimes assign bug fixes to inappropriate fixers. Anvik et al. [1] proposed a method to automatically identify fixers who have changed the buggy source code and have ever fixed similar bugs. This study should help to improve the accuracy of their method, because our method identifies the location of the buggy source code.

III. Approach

A. Overview

Our proposed method aims to automatically locate the buggy file. Figure 1 shows an overview of our method. Our method consists of 3 modules, text mining, code mining, and change history mining.

1) **Text mining approach**: The textual similarities between a bug report and source code are measured. The similarities are used to rank the buggy code.
2) **Code mining approach**: This approach determines the chance that a file is a defected source-code. The prediction model was trained using product metrics.
3) **Change history approach**: This approach also measures the chance that a file is a defected source-code.

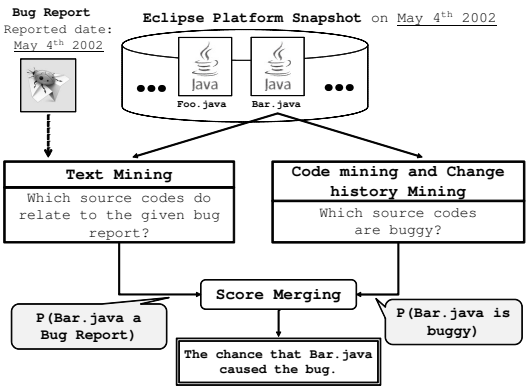

Figure 1. Overview of the proposed approach.

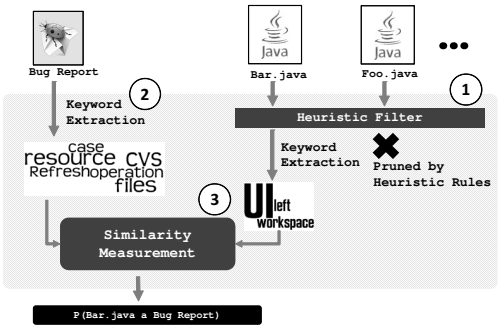

Figure 2. Overview of the text mining approach.

But, this model was built using change history process metrics such as the number of revisions of a file and the number of times a file was involved in a bug fix transaction. The code mining and change history approaches both use a similar process, but the models are constructed from those different metrics type.

B. Text mining approach

The text mining approach is composed of 3 submodules (which are*Heuristic Pruning*, *Keyword Extraction*, *Similarity Measurement*). Figure 2 shows an overview of this approach.

Heuristic Pruning Module aims to prune some irrelevant files away. In this scenario, irrelevant files are files that have a low chance to be a cause of a given bug. This module reduces the number of suspicious files to be checked. Consequently, this one reduces the time consumed by text mining, and boosts the accuracy.

Most related source code files contain a common component string in their file path. According to our observation, 91.5% of the bug reports in version 3.0 had changed source code in the component that the reporter noted in the bug report. In this study, we filtered the

source code based on the component name that was written in a bug report. The component filed is the requirement in the Eclipse bug report submission. For example, a reporter submitted bug report #251469 of Eclipse Project. The reporter informed that it happened at the *SWT* component. From this bug report, a developer made several changes to 11 files. All of their file paths contained the string *SWT* for example, org.eclipse.swt/Eclipse SWT/cocoa/org/eclipse/swt/widgets/Composite.java.

Keyword Extraction Module aims to extract significant keywords from the bug report and source code to identify associated source code that used the same significant keywords as the bug report.

Generally, a bug report contains the bug information in specific text fields (e.g. title, description, and comment). We gather keywords from the documents without using the comment field, because when triagers get a bug report, the developers have not submitted any comments yet. We applied the TreebankWordTokenizer provided by the NLTK python library [3] to segment sentences into words. TreebankWordTokenizer is a word tokenizer that tokenizes sentences using the conventions used by the Penn Treebank. Next, we get a list of terms in the bug report.

In the case of source code, we collect keywords from comments, class name, method name, and filename. However, most identifiers follow the CamelCase identifier naming convention (e.g. JFaceUtil, DebugUIViewsMessages, getLocation). Hence, additional tokenization of identifier names is necessary to avoid a vocabulary mismatch between the source code and bug report. Beside the original form of identifier names, they are normalized. "DebugUIViewsMessages" will be collected as DebugUIViewsMessages (original), Debug, UI, Views, and Messages. This tokenization produces a list of terms in the source code.

Each list of terms needs to be cleaned and pre-processed to reduce noisy data. We tried to remove function words that have little lexical meaning using a function words list provided by [5]. In addition, we applied stemming, which is an algorithm for changing derived words back to their base (e.g. "simulated", "simulation", and "simulates" all derive from "simulate"). This research adopted the Porter Stemmer [8], which is the de-facto standard for English stemming, via the NLTK python library [4].

Finally, the TF-IDF score in each document is calculated as follows.

$$tf(t,d) = \frac{f_{t,d}}{\#terms}, idf(t) = log(\frac{\#docs}{n_t}) \qquad (1)$$

f_{td} means the number of occurrences of a term t in document d. n_t means the number of documents that contain the term t. $\#terms$ mmeans the total number of terms in document d, and $\#docs$ means the total number of documents in the corpus.

Similarity Measurement aims to calculate the textual similarity between the source code and a bug report. This paper proposes a novel similarity measurement algorithm, called Bug to Code algorithm. In addition, we perform the experiment with the traditional Cosine Similarity too.

Cosine Similarity is represented a document as a vector using its TF-IDF score. This approach measures the similarity score of documents by determining the cosine of angle between them. Precisely, the similarity between two documents is defined by equation 2.

$$CosineSim(b,s) = \frac{\sum_{j=1}^{|V|} w_{b,j} \times w_{s,j}}{\sqrt{\sum_{j=1}^{|V|} w_{b,j}^2 \times \sum_{j=1}^{|V|} w_{s,j}^2}} \qquad (2)$$

b means the TF-IDF vector of targeted bug report, s means the TF-IDF vector of a source code, $|V|$ means the vocabulary size, and $w_{x,j}$ means the TF-IDF weights of j^{th} vocabulary in document x.

Bug to Code is a novel approach for determining textual similarity. The central idea of this algorithm is to look for some top keywords in the source code. First, all terms in bug report are ranked by the significant. Their TF-IDF scores are calculated and the highest β terms are selected. Next, each of the top β terms are re-calculated TF-IDF based on a java file. The summation of the re-calulated TF-IDF scores reveals how many degrees of relationship exist between the bug report and the java file. Precisely, the calculation procedure as follows

$$BugToCode(b,s) = \sum_{t \in T_\beta} tf \times idf(t,s) \qquad (3)$$

where

$$T_\beta = argmax_{\beta, t \in b}(tf \times idf(t,b)) \qquad (4)$$

where $tf \times idf(t,s)$ are TF-IDF score of term t based on the source code document, and $argmax_{\beta,t}(tf \times idf(t,b))$ returns terms that are ranked as the β highest TF-IDF score in bug report. As the experiment result, this algorithm performs the best result when β is 15.

C. Code mining approach

This approach aims to rank the target source code based on the output of the bug prediction model built using code metrics. The bug prediction model predicts fault-prone modules. This model is to identify source code that may be causing bugs. This study applied 12 code metrics as shown in table I.

Then, we combined the scores, which calculated by the bug prediction model and text mining, as in equation 5.

$$Score(b,s) = (1-\alpha) \cdot TM(b,s) + \alpha \cdot LR(CodeMetrics, s) \qquad (5)$$

$TM(b,s)$ means the text mining result of the bug report b and the source code s, $LR(CodeMetrics, s)$ means *Logistic Regression*'s prediction result of the source code based on code metrics.

Table I
LISTS OF CODE METRICS

Metrics	Metrics name	Description
code	SLOC	Source Lines of Codes
	MLOC	LOC executable
	PAR	Number of parameter
	NOF	Number of attributes
	NOM	Number of methods
	NSF	Number of static attributes
	NSM	Number of static methods
	NBD	Nested block depth
	VG	Cyclomatic complxity
	DIT	Depth of inheritance tree
	LCOM	Lack of cohesion of methods
	WMC	Number of weighted methods per class
change history	CodeChurn	Sum of (added lines of code - deleted lines of code)
	LOCAddes	Sum over all revisions of the lines of code added to a file
	LOCDeleted	Sum over all revisions of the lines of code deleted from a file
	Revision	Number of revisions of a file
	Age	Age of a file in weeks
	BugFixes	number of times a file was involved in a bug-fix transaction
	Refactorings	number of times a file has been refactored
	LastModDate	number of days a file has been latest modified

D. Change history mining approach

According to Kamei et al. [7], the process metrics outperformed product metrics. Then, this study performed change history mining to determine the chance that a source code could be buggy. As mentioned in the previous section, we adopted *Logistic Regression* to combine several process metrics. The targeted metrics are shown in table I.

The output probability was added to the combination of two previous approach's score. We studied two methods for combining the results together, shown in equations 6 and 7.

$$FinalScore(b,s) = \alpha \cdot TM(b,s) + (1-\alpha) \cdot LR(ProcMetrics, s) \quad (6)$$

$LR(ProcMetrics, s)$ means the predicted results of the source code s. The Logistic Regression was trained with only the process metrics values.

$$FinalScore(b,s) = \alpha \cdot TM(b,s) + (1-\alpha) \cdot LR(Proc\&CodeMetrics, s) \quad (7)$$

$LR(Proc\&CodeMetrics, s)$ is the predicted results of the source code, and the Logistic Regression includes both process metrics and code metrics values. As far as we can observe, the most suitable α is 0.2. All source code was ranked by $FinalScore(b,s)$. The higher $FinalScore(b,s)$ indicates that the file is more relevant.

IV. EXPERIMENT SETTING

This study performed the experiments on Eclipse Platform 3.0 and 3.1, which is a well-known open source software development environment. The Eclipse Platform uses Bugzilla as a bug tracking systems and CVS as a version control system. 2,950 bug reports in XML format were acquired from MSR Mining Challenge 2008. We focused on bug reports from ID #1 until ID #205497. Consequently, we checked out 475 code snapshots from 21st June 2004 until 29 th June 2006, a total of 48,764 source code files. All 2,950 bug reports were separated into two equal data sets for training and testing. We split the data along the time axis. All bug reports from 21 st June 2004 to 28th June 2005 were training data, the others were testing data.

To collect links between bug reports in the bug tracking system and committed changes in the source code repository, we obtained data using the familiar SZZ algorithm [10]. We investigated the result of linking with SZZ. About 57% of the bug reports linked with only one java file.

V. RESEARCH QUESTION

RQ: ***How accurately can our proposed algorithm identify source code that should be fixed?***

Motivation. Basically, triagers assign a bug to a developer who has changed the source code. However, triagers do not always assign a bug to the right developer, so the assigned developer has to identify source code which should be fixed based on an initial bug report. In this study, we propose a method to automatically identify the source code that caused bugs.

Approach. This research studied three approaches. One determines the textual similarity between a bug report and source code as mentioned in section III-B. Another one predicts fault-prone modules based on code metrics. The third one also predicts fault-prone modules, but based on change history metrics. To measure the scores achieved by these 3 approaches, first, the corresponding parameters (β) of Bug To Code were optimized based on the training dataset. Next, we run our two algorithms on the training dataset as well as compared the output with linkage of a bug report and a source code file, as described in Section IV. We combined the code mining approach and change history approach with equations 5, 6, and 7. In this experiment, we evaluated the accuracy of approaches via these metrics as follows :

Top N Rank: The percentage of bug report that the algorithm ranks at least one of actual buggy files that related given bug report. The higher value reveals the better accuracy. This metric can be explicitly outlined in terms of following equation:

$$Top\,N\,Rank = \frac{|\,\{b \in B | \lambda_N(b) \cap R_b \neq \phi\}\,|}{|B|} \times 100 \quad (8)$$

B means a set of bug reports, $\lambda_N(b)$ means the top N output of the algorithm given b, and R_b the set of files that developer changed to resolve the bug report b.

Mean Reciprocal Rank (MRR):A statistic for evaluating a process that produces a list of possible responses to a query

Table II
ACCURACY OF OUR PROPOSED APPROACH

Algorithm	TOP1	TOP5	TOP10	MRR
bug2code	20.09	37.65	45.25	0.29
bug2code with codeonly	20.14	37.71	45.20	0.29
bug2code with processonly	19.94	37.91	45.35	0.29
bug2code with code and process	**20.20**	37.96	45.25	0.29

[11]. It can be calculate as follows:

$$MRR = \frac{1}{|B|}\sum_{i=1}^{|B|}\frac{1}{rank_i} \quad (9)$$

B means a set of bug reports, and $rank_i$ means the rank of the first correct answer of bug report i.

Results. To answer the research question, we designed two experiments. *Experiment 1* compared two textual similarities measurement algorithms first. Next, we studied three combinations of text mining and code-change history mining. *Experiment 2* aims to study the effect of characteristics of bug report on the accuracy as shown in Table 3.

Experiment 1: Locating source codes

1) *Selection of Text Mining Approaches*: First, we investigated the effect of β on the performance of the Bug to Code approach. According to the experiment, our system show the best result when β is 15. It is obvious that we could obtain the best Top1 result (20.51%). The classical Cosine Similarity approach showed the lower precision at 12.82% Top1 precision. Based on this, we believe that our proposed text mining algorithm, Bug to Code, is better than Cosine Similarity.
2) *Selection of Score Merging Approach*: This experiment conducted using our three proposed score merging approaches (equation 5, 6, and 7). The score merging approach was utilized to combine the scores of text mining, which is Bug to Code, code mining, and change history mining. The details of this merging approach was mentioned in section III-C and III-D. The results are shown in table II. Unfortunately, a bug prediction based on code metrics and process metrics did not contribute much to identify a location of buggy code files.

Experiment 2: Locating source code according to the description

In this study, we proposed a method to identify the buggy source code that includes specific keywords extracted from a bug report document. Most of the specific words are found in the description of a bug report. Bettenburg et al. [2] studied how to write good bug report for developers. They revealed that test code and example code are helpful for developers. Thus , we expected a difference in accuracy based on features of the bug report description.

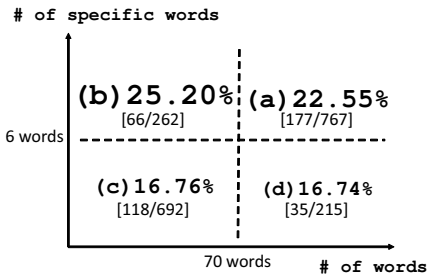

Figure 3. The accuracy of our model according to features of the description (*[66/262]* means 262 bug reports contain more than 6 specific words. The length of these 262 bug reports is more that 70 words. And, the related buggy file is ranked at first in 66 bug reports.)

Figure 3 shows Top1 precision related to the number of words and the number of specific words. To calculate the accuracy due to each feature of the target bug reports, we classified them into the following 4 groups according to the number of words and the number of specific words in the description of the bug report.

As a result, our study showed that our proposed model had higher accuracy if there were many specific words in a bug report. Also, the model had higher accuracy if the bug report description was short.

VI. DISCUSSION

To identify the buggy source code based on an initial bug report, we proposed a method using 3 approaches. Our approach based on a text mining approach has about the same accuracy as the model proposed by Zhou et al. [12]. One of the reasons is the low accuracy of a bug prediction. At present, The accuracy of the bug prediction is only about 53%. The weighting of the code mining approach and change history approach shown in equation 5 was also low because the accuracy of prediction of the fault-prone module was low. In future work, we will try to improve the accuracy of prediction of the fault-prone module to improve the accuracy of our approach.

Our approach identifies specific source code that caused a bug. However, developers do not always change a single source file to fix bugs. According to our analysis, for 43% bug reports developers changed over 2 source files to fix the bugs. In future work, we will try to identify if there are several source code files depending on the buggy source code file.

VII. THREATS TO VALIDITY

Our study had three limitations. First, the focus of our study was on resolving software bugs by making changes to existing source code. Developers could deal with some kinds of bug reports by adding new files. Unfortunately, this case is beyond the scope of our research.

Second, the accuracy of our software depends on the quality of the bug report and the readability of the source code. If the source code consists of meaningless identifier names, it leads to lower accuracy. In the same way, if the bug report provided poor information, it will decrease our accuracy.

Finally, Zhou et al. [12] used similar bug reports to locate the buggy file. However, less than 30% of the bug reports linked the changed source code when the bug was fixed. So, using similar bug reports to identify source code is not so easy. In this study, we did not use this approach. In future work, if many bug reports were linked to the changed source codes when the bug was fixed, this approach might be helpful.

VIII. CONCLUSION AND FUTURE WORKS

In this paper, we proposed a method to identify buggy files in a source code repository based on 3 approaches, text mining , code mining, and change history mining. Using Eclipse platform project data, our model gained around 20% TOP1 precision. Furthermore, our study revealed that bug reports that contained a short description and many specific words was easier to use to locate the buggy files. Even when developers do not have enough knowledge of the reported bugs, they have to assign bugs and to fix bugs. Therefore developers often assign inappropriate fixers. Our method reduces time spent reassigning bugs. In our future work, we will try to improve the accuracy of prediction of fault-prone modules, and to identify when source code files depend on buggy files. However, it takes a long time to identify the buggy files from the large amount of software development data. Therefore, in addition to improving the accuracy of prediction, we will also try to improve the performance of the prediction system.

ACKNOWLEDGMENT

We would like to thank Dr.Mike Barker for helping us to write this paper. The first author is grateful to the internship program that is jointly supported between Kasetsart University, Thailand, and Nara Institute of Science and Technology, JAPAN. This research was conducted as part of the Grant-in-Aid for Young Scientists, 24680003, 2012 by the Japan Society for the Promotion of Science.

REFERENCES

[1] J. Anvik, L. Hiew, and G. C. Murphy. Who should fix this bug? In *Proceedings of the 28th International Conference on Software Engineering (ICSE'06)*, pages 361–370, 2006.

[2] N. Bettenburg, S. Just, A. Schröter, C. Weiss, R. Premraj, and T. Zimmermann. What makes a good bug report? In *Proceedings of the 16th International Symposium on Foundations of Software Engineering (FSE'08)*, pages 308–318, 2008.

[3] S. Bird. Nltk: the natural language toolkit. In *Proceedings of the COLING/ACL on Interactive presentation sessions (COLING-ACL'06)*, pages 69–72, 2006.

[4] S. Bird. Nltk: the natural language toolkit. In *Proceedings of the COLING/ACL on Interactive presentation sessions (COLING-ACL'06)*, pages 69–72, 2006.

[5] L. Gilner and F. Morales. Academic resources, September 2006.

[6] G. Jeong, S. Kim, and T. Zimmermann. Improving bug triage with bug tossing graphs. In *Proceedings of the 7th Joint Meeting of the European Software Engineering Conference and the Symposium on the Foundations of Software Engineering (ESEC/FSE'09)*, pages 111–120, 2009.

[7] Y. Kamei, S. Matsumoto, A. Monden, K.-i. Matsumoto, B. Adams, and A. E. Hassan. Revisiting common bug prediction findings using effort-aware models. In *Proceedings of the International Conference on Software Maintenance (ICSM'10)*, pages 1–10, 2010.

[8] M. F. Porter. Readings in information retrieval. pages 313–316. Morgan Kaufmann Publishers Inc., CA, USA, 1997.

[9] S. Rao and 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 (MSR'11)*, pages 43–52, 2011.

[10] J. Śliwerski, T. Zimmermann, and A. Zeller. When do changes induce fixes? In *Proceedings of the 4th International Workshop on Mining Software Repositories (MSR'07)*, pages 1–5, 2005.

[11] E. M. Voorhees. The trec-8 question answering track report. In *In Proceedings of the Text Retrieval conference (TREC-8)*, pages 77–82, 1999.

[12] J. Zhou, H. Zhang, and D. Lo. Who should fix this bug? In *Proceedings of the International Conference on Software Engineering (ICSE'12)*, pages 14–24, 2012.