Boosting Bug-Report-Oriented Fault Localization with Segmentation and Stack-Trace Analysis

Chu-Pan Wong*, Yingfei Xiong*, Hongyu Zhang†, Dan Hao*, Lu Zhang*, and Hong Mei*
*Key Laboratory of High Confidence Software Technologies (Peking University), MoE, China
School of EECS, Peking University, China
Email: {chupanwong,meih}@pku.edu.cn,{xiongyf04,haod,zhanglu}@sei.pku.edu.cn
†Microsoft Research, Beijing, China
Email: honzhang@microsoft.com

***Abstract*—To deal with post-release bugs, many software projects set up public bug repositories for users all over the world to report bugs that they have encountered. Recently, researchers have proposed various information retrieval based approaches to localizing faults based on bug reports. In these approaches, source files are processed as single units, where noise in large files may affect the accuracy of fault localization. Furthermore, bug reports often contain stack-trace information, but existing approaches often treat this information as plain text. In this paper, we propose to use segmentation and stack-trace analysis to improve the performance of bug localization. Specifically, given a bug report, we divide each source code file into a series of segments and use the segment most similar to the bug report to represent the file. We also analyze the bug report to identify possible faulty files in a stack trace and favor these files in our retrieval. According to our empirical results, our approach is able to significantly improve BugLocator, a representative fault localization approach, on all the three software projects (i.e., Eclipse, AspectJ, and SWT) used in our empirical evaluation. Furthermore, segmentation and stack-trace analysis are complementary to each other for boosting the performance of bug-report-oriented fault localization.**

Keywords*—*fault localization, bug report, feature location, information retrieval

I. Introduction

With the increasing size of software and the limited development resources, it is often inevitable to release software systems with bugs. A common way to deal with those post-release bugs is to set up a public bug repository to collect bug reports from users. For a popular software system, the number of bug reports in its bug repository may increase rapidly. For example, as counted by Zhou et al. [1], the Eclipse project received 4414 bug reports in 2009. Therefore, it is typically vital to the success of such a system whether its development team can fix the received bug reports in an efficient way. However, given a bug report, it is often painstaking for a developer to manually locate the buggy source code in the code base. It may become even worse for a new developer to handle a bug report for a very large code base (containing thousands of files).

To help developers handle bug reports, researchers have investigated approaches to bug-report-oriented fault localization (e.g., [2], [3], [4]), which try to find among the entire code base a small subset of source files that are directly related to fixing each bug report. These approaches typically rely on information retrieval [5], [6] where bug reports are treated as queries to locate a few related source files from the whole collection of source files. Both bug reports and source code are usually treated as text objects. Different information retrieve techniques, such as LDA [7], SUM [8], VSM [1], or combination of different techniques [9] have been used.

Although a large number of approaches have been proposed, we found that the existing approaches still do not deal with the following two issues well, leading to accuracy loss.

- **Large files.** Typically, existing approaches treat each source file as a single unit, and use IR-based approach to match the files. Some approaches, such as BugLocator [1], have further introduced parameters to favor large files as existing studies [10], [11] have shown that larger files are far more likely to be bug-prone. However, due to the fuzzy nature of information retrieval, textually matching bug reports against source files may have to concede noise (less relevant but accidentally matched words). Large files are more likely to contain noise as usually only a small part of a large file is relevant to the bug report. By treating files as single units or favoring large files, we are more likely to be affected by the noise in large files.
- **Stack traces.** Bug reports often contain stack-trace information, which may provide direct clues for possible faulty files. Most approaches directly treat the bug descriptions as plain texts and do not explicitly consider stack-trace information.

To deal with the two issues, we propose two novel heuristic techniques over existing approaches to bug-report-oriented fault localization, as follows.

- **Segmentation.** We divide each source code file into a series of segments and use the segment with the highest similarity to the bug report to represent a file. By using one segment to represent a source code file, we are able to reduce some noise that may exist in the other segments of the file.
- **Stack-trace analysis.** We identify files that are related to stack traces in bug reports, and increase the ranking of these files based on the observation that the files covered by the stack-trace are more likely to be bug-prone. Although some other approaches (e.g., [12]) have attempted to use stack-trace information, we use

Bug ID	87692
Summary	setConsoleWidth() causes Invalid thread access
Description	calling setConsoleWidth() outside UI thread causes Invalid thread access.

Fig. 1: The First Bug-Report Example

it in a novel way and make the first attempt to combine it with our segmentation technique.

We realize the two techniques on top of BugLocator [1], a state-of-art approach on bug-report-oriented fault localization. We call the integrated approach *BRTracer*.

We further performed an experimental evaluation of BRTracer together with BugLocator on three subject Java projects using three widely used metrics. Our empirical results demonstrate that our two techniques are able to significantly improve the performance of BugLocator with or without considering similar bug reports (similar bug reports are first used in BugLocator to boost bug localization). Furthermore, our experiments show that either segmentation or stack-trace analysis is effective in boosting bug-report-oriented fault localization. They are complimentary to each other and thus better used together.

II. Motivating Examples

In this section, we present two examples selected from Eclipse 3.1 project to further motivate our research.

A. Example One

Figure 1 depicts an example bug report for Eclipse, in which there is a clear clue to the source of the bug but only limited information available for information retrieval. Developers modified file `TextConsoleViewer.java` to fix this bug, but BugLocator ranks `Accessible.java` as first while this file at the 26th.

To understand the cause of the inaccuracy, we investigate the information retrieval process of BugLocator. According to the revised Vector Space Model (rVSM), the final similarity score between a file and a bug report is the multiplication of two factors: a similarity score calculated by the basic Vector Space Model (VSM)[1] and a length score calculated by a length function. As `Accessible.java` is several times larger than `TextConsoleViewer.java`, BugLocator assigns a larger length score to `Accessible.java` than to `TextConsoleViewer.java`. What really surprises us is that both files have similar similarity scores calculated by the basic VSM. In fact, the bug report perfectly matches `TextConsoleViewer.java` (where method `setConsoleWidth` is declared), but a large number of occurrences of "`access`", "`invalid`" and "`call`" in `Accessible.java` significantly increase its similarity score. As a result, the final similarity score for `Accessible.java` is significantly larger than that for `TextConsoleViewer.java` in BugLocator. This issue also accounts for many highly ranked files for this bug report.

[1]Here a logarithm function is used to calculate the term frequency.

In fact, the fuzzy nature of information retrieval determines that when matching a bug report against source code files, the number of code snippets accidentally matched to the bug report is inevitably large. Thus, those accidentally matched code snippets (i.e., noise) may become an important factor that impacts the effectiveness of rVSM in BugLocator. For `Accessible.java`, the noisy matches of "`access`", "`invalid`" and "`call`" are largely magnified in rVSM to push `Accessible.java` into the first position in the rank.

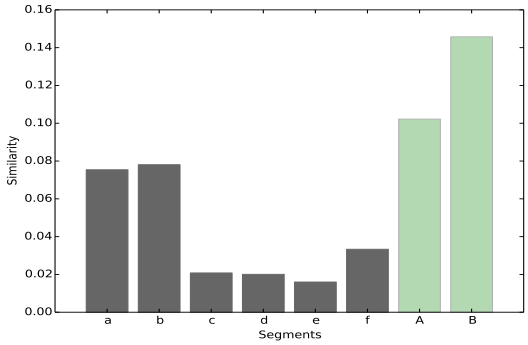

Fig. 2: Similarity between Bug Report and Each Segment

We further divide both `Accessible.java` and `TextConsoleViewer.java` into equally-sized segments. In particular, we divide `Accessible.java` into six segments (denoted as a, b, c, d, e, f), and `TextConsoleViewer.java` into two segments (denoted as A, B). Figure 2 depicts the similarity score calculated by the basic VSM in BugLocator for each segment. In this setting, one segment (i.e., B) from `TextConsoleViewer.java` overwhelms any segment from `Accessible.java` in similarity. If this similarity score is used to calculate the final similarity score in rVSM, `TextConsoleViewer.java` instead of `Accessible.java` would be ranked in the first position.

B. Example Two

Among 3459 bug reports in our dataset, we observe that bug reporters include stack-trace information in more than 17% bug reports. A previous study by Schröter et al. [13] shows that the top-10 functions in stack traces are likely to be the sources of bugs. Therefore, stack-trace information in bug reports can be a good information source for bug localization.

Figure 3 depicts an example bug report for Eclipse, which contains a very long stack trace observed by the bug reporter. In the real fix of this bug, file `Table.java` (which also appears in the stack trace) is actually modified. That is to say, we may be able to locate `Table.java` quite conveniently by analyzing the stack trace in the bug report.

However, existing approaches always treat bug descriptions as plain texts and thus may hardly be able to handle stack-trace information properly. First, a file name (which is also a class name in Java) in a stack trace may typically match only a few times in the file defining the class, but may match many times in other files using the class. Second, a stack trace may contain a large volume of information, resulting a large number of matches (including many noisy matches) in various files. For this example, noisy matches (e.g., "`java`", "`button`", and "`launcher`" etc.) overwhelm the matches in "`Table.java`",

Bug ID	87855
Summary	NullPointerException in Table.callWindowProc

Here is a stack trace I found when trying to kill a running process by pressing the kill button in the console view. I use 3.1M5a.
!ENTRY org.eclipse.ui 4 0 2005-03-12 14:26:25.58
!MESSAGE java.lang.NullPointerException
!STACK 0
java.lang.NullPointerException
at org.eclipse...Table.callWindowProc(**Table.java**:156)
at org.eclipse...Table.sendMouseDownEvent(**Table.java**:2084)
at org.eclipse...Table.WM_LBUTTONDOWN(**Table.java**:3174)
at org.eclipse...Control.windowProc(Control.java:3057)
at org.eclipse...Display.windowProc(Display.java:3480)
...
at org.eclipse.core.launcher.Main.run(Main.java:887)
at org.eclipse.core.launcher.Main.main(Main.java:871)

Fig. 3: The Second Bug-Report Example

resulting a very low position of "`Table.java`" in the rank (i.e., at the 252nd position). On the other hand, our stack-trace analysis can push it to the 5th position.

It should also be noted that it may not be appropriate to directly deem all files appearing in stack traces as faulty, because one stack trace may contain many file names and the real faulty file may not appear in the stack trace at all.

III. Approach

As mentioned before, we build our two heuristic techniques on top of BugLocator [1], resulting in a new tool BRTracer. That is to say, both BugLocator and BRTracer share the same retrieval model (i.e., rVSM). As mentioned previously, rVSM uses a length function as a factor to favor larger files. In BRtracer, we further introduce a parameter into the length function. Using this parameter (which we refer to as the *magnifier*), we are able to provide different degrees of favor that we give to larger files. Thus, on the basis of controlling the noise through segmentation, we may magnify the favor assigned to larger files to further take advantage of rVSM.

It should be noted that the two heuristic techniques we proposed can be easily migrated to other IR-based approaches. We develop BRTracer on top of BugLocator because the retrieval model implemented in BugLocator is basic and straightforward, which make it convenient to demonstrate the effectiveness of segmentation and stack-trace analysis.

A. Overview

In our approach, we first divide each source code file into segments and match the bug report against each segment, resulting an initial similarity score for each segment. Then, we calculate a length score for each source code file with our parameterized length function. Furthermore, we perform stack-trace analysis on the bug report and get a boost score for each source code file. For each source code file, we first use the segment with the highest initial similarity score to represent the file and multiply this initial similarity score with the length score of the file to calculate the similarity score of the file, then we add the corresponding boost score to this similarity score to calculate the final score. In the following, Sections III-B, III-C, III-D, and III-E present the details of our segmentation strategy, our parameterized length function, our stack-trace analysis, and the calculation of the final score, respectively.

B. Segmenting Source Code Files

For a bug report, as there may be many code snippets in a source code file (especially a large file) being textually similar to the bug report, the aim of segmentation is to use only the most similar code snippet to represent the file. Therefore, how to divide a source code file into segments may have significant impacts on the effectiveness of this segmentation approach. One straightforward method is to divide a file by lines. In addition, we could also divide a source code file by methods, treating each method in a Java class as a segment. However, as the primary goal of our segmentation is to prevent noise in larger files from being too much magnified, using segments with equal sizes should be more suitable for this goal. Therefore, in order to get segments that vary little in size, we evenly divide each source code file based on its code corpus.

Given a source code file, we first extract a corpus (i.e., a series of words) in a similar way to existing approaches[2]: First, we extract a series of lexical tokens from the file. Second, we remove keywords (e.g., $float$ and $double$ in Java) specific to the programming language and separate each concatenated word (e.g., $isCommitable$) into several words. Finally, we remove stop words (e.g., a and the) and stem each word into its root form[3].

Let us use Cor=$\{w_1, w_2, ..., w_n\}$ to denote the corpus of a source code file, where w_i is the i-th word in the corpus. Given the maximum number (denoted as l) of words in each segment, we divide Cor into k segments denoted as Seg_1, Seg_2,..., Seg_k, where $k = \lceil \frac{n}{l} \rceil$, Seg_i=$\{w_{(i-1)l+1}, ..., w_{il}\}$ for any $1 \leq i < k$, and Seg_k=$\{w_{(k-1)l+1}, ..., w_n\}$. In such a segmentation strategy, only the number of words in the last segment may be less than l. For example, if the corpus contains 11 words (i.e., $w_1, w_2, ..., w_{11}$) and each segment contains at most 4 words, we have the following 3 segments: $\{w_1, w_2, w_3, w_4\}$, $\{w_5, w_6, w_7, w_8\}$, and $\{w_9, w_{10}, w_{11}\}$. Note that, as some pieces of source code may contribute very little to the corpus, segmentation on the basis of the extracted corpus instead of the source file can ensure to produce evenly divided segments.

In fact, the segmentation strategy itself only provides us with a mechanism to avoid uneven segmentation. We still need to assign a proper value to l. If we assign a too small number to l, the preceding segmentation strategy may still divide the most similar code snippet in the file into different segments. Of course, if we assign a too large number to l, each segment itself may contain much noise.

C. Parameterizing the Length Function

In rVSM, a length function based on the logistic function[4] depicted in Formula 1 (where $\#terms$ denotes the number

[2]In fact, all existing approaches [7], [8], [1] to bug-report-oriented fault localization use very similar ways to build corpora from source code files.

[3]Currently, we use the Porter Stemming algorithm provided in http://tartarus.org/martin/PorterStemmer/.

[4]http://en.wikipedia.org/wiki/Logistic_function

of terms (words) in the corpus extracted from a source code file, and $Nor()$ is a normalization function) is used to give favor to larger files. Thus, the final similarity in rVSM is the initial similarity calculated with the basic Vector Space Model multiplied by the length function. According to experimental results reported by Zhou et al. [1], the length function in the form of the logistic function is more effective than length functions in other forms.

$$LenFunc(\#terms) = \frac{1}{1 + e^{-Nor(\#terms)}} \quad (1)$$

In our approach, we parameterize Formula 1 with a new factor β ($\beta > 0$) to form Formula 2. According to the property of the logistic function, given the same file, the larger the value of β is, the more favor is given to the size (represented by the number of terms) of the file. Therefore, parameter β actually serves as a $magnifier$ to adjust how much favor we give to larger files. By adjusting $magnifier$, we are able to develop a better balance between favoring large files and reducing noise in large files.

$$LenFunc(\#terms) = \frac{1}{1 + e^{-\beta \times Nor(\#terms)}} \quad (2)$$

To further utilize the property that the logistic function has the largest capability to distinguish different inputs when the input value is around zero, we use of the normalization function depicted in Formula 3, which is able to normalize the medium value to zero to make our length function better distinguish files with the numbers of terms close to the medium value.

$$Nor(t) = \frac{t - t_{med}}{t_{max} - t_{min}} \quad (3)$$

Given a set of values for variable t, t_{max}, t_{min}, and t_{med} in Formula 3 represent the maximum value, the minimum value, and the medium value among all the values in the set, respectively.

D. Analyzing Stack-Trace Information

As discussed in Section II-B, it is common that buggy files are directly mentioned in bug reports, usually in the form of stack traces. Therefore, we extract all the file names directly mentioned in the bug report using a regular expression. In the regular expression, we look for phrases ended with ".java" and we further require that what appears before ".java" should form a valid file name for Java (e.g., containing only letters, digits, underscores, and hyphens with the beginning symbol to be a letter). Then we check the file names against the code base to eliminate file names that do not exist in the code base. This checking would help filter out both mistakenly obtained file names and file names in libraries. Finally, we get a set of source code files (denoted as $\mathcal{D}$) directly mentioned in the bug report.

Moreover, if a stack trace indicates that a file is suspicious, the root cause may actually lie in some methods (of some classes) used in this file. Therefore, these classes may also deserve more attention. In our approach, we also analyze each file (denoted as f) in $\mathcal{D}$ to get a set of files (classes) directly used in f. For ease of presentation, we refer to the set of files, each of which is used by some file in $\mathcal{D}$, as the closely related file set (denoted as $\mathcal{C}$). In particular, for each Java source code file in $\mathcal{D}$, we extract all the import statements and record the corresponding files referred to in these statements. As for the import statements for packages, we put all the classes in the packages into $\mathcal{C}$. After filtering out files not existing in the code base, we obtain the closely related file set $\mathcal{C}$.

As our goal is to favor files in the two sets $\mathcal{D}$ and $\mathcal{C}$, we calculate an additional score (referred to as the $boost\ score$ in this paper) for each file in $\mathcal{D}$ or $\mathcal{C}$. Note that the use of a boost score makes it easy to incorporate the results of our stack-trace analysis into the retrieval process.

In particular, given a file (denoted as x), we use Formula 4 to calculate the boost score for x. For the convenience to incorporate the boost score into the retrieval process, we deem the boost score of any file neither in $\mathcal{D}$ nor in $\mathcal{C}$ as 0. A previous study by Schröter et al. [13] shows that a defect is typically located in one of the top-10 stack frames. Furthermore, the higher stack frame is more likely to be faulty than the lower one. Therefore, our calculation of the $boost\ score$ also depends on the rank of each source file in a stack trace. In the following formula, $rank = 1$ refers to the highest source file in a stack trace. For example, in Figure 3, $rank(Table.java) = 1$.

$$BoostScore(x) = \begin{cases} \frac{1}{rank} & x \in \mathcal{D}\ \&\ rank \leq 10 \\ 0.1 & x \in \mathcal{D}\ \&\ rank > 10 \\ 0.1 & x \in \mathcal{C} \\ 0 & otherwise \end{cases} \quad (4)$$

E. Calculating the Final Score

In our approach, we use Formula 5 to calculate the final score of a source code file. Given file x, $rVSM_{seg}(x)$ (i.e., the similarity score between file x and the bug report) is the multiplication of the similarity score of the most similar segment in x and the length score of x; and N is a normalization function to normalize $rVSM_{seg}(x)$ into range [0..1].

$$FinalScore(x) = N(rVSM_{seg}(x)) + BoostScore(x) \quad (5)$$

As BugLocator has the ability to utilize similar bug reports, we adopt the same mechanism to utilize similar bug reports in our approach. Formula 6 depicts the calculation of the final score considering similar bug reports, where $SimiScore(x)$ is the score calculated from similar bug reports and α is a value between 0 and 1 denoted as the weight of $SimiScore(x)$.

$$\begin{aligned} FinalScore(x) = (1 - \alpha) \times N(rVSM_{seg}(x)) \\ +\alpha \times N(SimiScore(x)) + BoostScore(x) \end{aligned} \quad (6)$$

In both BugLocator and our approach, $SimiScore(x)$ is calculated as follows. The bug report (denoted as BR) is matched against each previously fixed bug report. Let B_i denote the i-th previously fixed bug report and S_i denote the similarity between BR and B_i. Thus, Formula 7 calculates

$SimiScore(x)$, where F_i is the set of modified files for fixing B_i and $|F_i|$ is the number of files in F_i.

$$SimiScore(x) = \sum_{x \in F_i} \frac{S_i}{|F_i|} \quad (7)$$

IV. Empirical Evaluation

A. Research Questions

As we implemented our two heuristic techniques on top of BugLocator, we are interested in whether the new tool, BRTracer, is able to outperform BugLocator. In our comparison, we treated similar bug reports as an independent variable. This is because of the following reasons. First, the information of similar bug reports may not be fully available for some existing projects. Furthermore, collecting similar bug data requires recovering links between source code files and bug reports, which is not a trivial task. Therefore, we are interested in how BRTracer compares with BugLocator both with and without considering similar bug reports. Second, as both BugLocator and BRTracer share the same mechanism to deal with similar bug reports, treating similar bug reports as an independent variable would help us further investigate the strengths and weaknesses of both approaches. Therefore, there are two research questions in our empirical evaluation:

- **RQ1**: How does our approach compare to BugLocator either with or without considering similar bug reports?
- **RQ2**: How do segmentation and stack-trace analysis impact the effectiveness of our approach?

B. Independent Variables

To answer the preceding research questions, we were primarily concerned with the following two independent variables: the experimented approach and the consideration of similar bug reports. We compared the following four implementations in our evaluation.

1) BugLocator without considering similar bug reports.
2) BugLocator considering similar bug reports.
3) BRTracer without considering similar bug reports.
4) BRTracer considering similar bug reports.

C. Subject Projects

For better comparison with BugLocator, we evaluated BRTracer using the same subjects as Zhou et al. [1]. In our empirical evaluation, we used the following three projects: Eclipse[5] (v3.1), AspectJ[6] (v1.5), and SWT[7] (v3.1). All projects have a fairly large number of bug reports that help us achieve statistically sound empirical results. Eclipse, which is a large open source development platform widely used in various empirical studies, has been used by Poshyvanyk et al. [2], [3], Gay et al. [4], Lukins et al. [7], Bangcharoensap et al. [9], and Zhou et al. [1] to investigate bug-report-oriented fault localization. AspectJ, which is a famous aspect-oriented extension for Java and is provided by Dallmeier and Zimmermann [14] as a part of the iBUGs[8] benchmark for evaluating techniques for bug-report-oriented fault localization, has been used by Rao and Kak [8] and Zhou et al. [1]. SWT (v3.1), which has been used by Zhou et al. [1], is a library providing various graphical widgets used in Java. Table I depicts statistics related to the three projects.

TABLE I: Statistics of Studied Projects

Project	Studied Period	#Bug Reports	#Source Files
Eclipse 3.1	Oct 2004 - Mar 2011	3075	11892
AspectJ 1.5	Jul 2002 - Oct 2006	286	5487
SWT 3.1	Oct 2004 - Apr 2010	98	484

D. Metrics

In our empirical evaluation, we consider the following three metrics: top N rank of files, mean reciprocal rank, and mean average precision. The first one is a metric widely used in evaluating techniques for bug-report-oriented fault localization. The latter two are metrics widely used in information retrieval.

Top N Rank of Files (TNRF). Given a number N, the top N rank gives the percentage of bugs whose related files are listed in top N (which is set to 1, 5, and 10 in our evaluation) of returned files. Here, following how Zhou et al. [1] used the TNRF metric to evaluate BugLocator, we use the highest ranked one among all the related files when a bug report has more than one related files.

Mean Reciprocal Rank (MRR). The MRR metric [15] measures the overall effectiveness of retrieval for a set of bug reports. Let us denote the set of bug reports as BR and the rank of the first related file of the i-th bug report as $rank(i)$. The value of the MRR metric is defined in Formula 8.

$$MRR = \frac{\sum_{i=1}^{|BR|} 1/rank(i)}{|BR|} \quad (8)$$

Mean Average Precision (MAP). The MAP metric [16] provides a synthesized way to measure the quality of retrieved files, when there are more than one related file retrieved for the bug report. Supposing that m related files are retrieved, if we use f_1, f_2,..., f_m to denote the m related files in the same order as they are ranked and $Pos(i)$ to denote the position that f_i is ranked at, the average precision (denoted as AvgP) is defined in Formula 9. Given a set of bug reports, we use the mean value of the AvgP values of all bug reports in the set to define the MAP metric.

$$AvgP = \frac{\sum_{i=1}^{m} i/Pos(i)}{m} \quad (9)$$

Among the preceding three metrics, the TNRF metric does not consider the quality of retrieval outside the top N files, while the other two metrics consider the overall quality of retrieval. Furthermore, larger values for all the three metrics indicate better effectiveness.

E. Experimental Procedure

To use the three projects for our empirical evaluation, we need to prepare both the data related to source code files and the data related to bug reports. Since we would like to compare BRTracer with BugLocator, we follow the steps described in

[5]http://www.eclipse.org
[6]http://eclipse.org/aspectj/
[7]http://www.eclipse.org/swt/
[8]http://www.st.cs.uni-saarland.de/ibugs/

[1] to prepare our data. For Eclipse and SWT, we downloaded the source code of the corresponding versions from their project website. For AspectJ, we downloaded the source code from iBUGs. For each subject project, we collected a set of fixed bug reports from its bug tracking system and mined the links between bug reports and source code files following the same methodology[9] that Zhou et al. [1] used to evaluate BugLocator. As a result, we collected a total of 3459 bug reports with their links to source code files. We put both the data related to the bug reports and the used source code of the three projects on our project website[10].

In particular, we adopted the following classic heuristics suggested by Bachmann and Bernstein [17] to recover links between source code files and bug reports. With these recovered links, we were able to calculate the effectiveness of each approach under experimentation in terms of the three metrics.

First, we scanned through the change logs of the three projects to identify bug IDs (e.g., "issue 327"). As there might be several formats of recording bug IDs, we tried all the formats we could think of. Second, we further excluded false bug IDs (e.g., "r360") through manual examination. To ensure accuracy, we deemed as false bug IDs any ones that we were not certain to be real bug IDs. Third, we also searched in the bug tracking system to identify IDs of fixed bugs. Finally, we mined the version repository (e.g., SVN) to identify changed files for each identified bug ID. Note that each bug report is also associated with a date for submission and a date for fixing. Given a bug report (denoted as r), when considering similar bug reports for r, only those both whose submission date and whose fixing date are prior to the submission date of r can be candidates.

Our initial experience with our approach indicated that our approach with the segment size being 800 words (i.e., l=800), the magnifier in our length function being 50 (i.e., β=50) might typically achieve competitive effectiveness. In fact, we have tuned l ($l = 400, 600, 800, 1000, 1200$) and β ($\beta = 30, 50, 70, 90$) to explore how they affect the retrieval results. Our experiment shows that, BRTracer is able to achieve effectiveness no worse than BugLocator in most parameter settings, but the optimized parameter settings are different from project to project. It seems that l and β should be set according to some characteristics of the target project (e.g., project size, average source file size), and we leave that to our future work. In order to demonstrate the general effectiveness of our segmentation technique, throughout all experiments in our empirical evaluation, we set $l = 800$ and $\beta = 50$ as the default values of our approach. Furthermore, to balance the similarity acquired from similar bug reports and the similarity from direct matching the bug report with each source code file, we set the value of α in Formula 6 as 0.2 (which Zhou et al. [1] also used in their evaluation of BugLocator) for both BugLocator and our BRTracer.

F. Results and Analysis

1) RQ1: Overall Effectiveness: Table II depicts the overall effectiveness of BugLocator and our BRTRacer without considering similar bug reports (SBRs). From Table II, we have the following observations.

TABLE II: Overall Effectiveness without Considering SBRs

Subject	Approach	Top N (%)			MRR	MAP (%)
		N=1	N=5	N=10		
Eclipse	BugLocator	25.8	47.9	58.2	36.5	27.5
	BRTracer	29.6	51.9	61.8	40.2	30.3
AspectJ	BugLocator	24.1	47.2	60.4	35.1	19.8
	BRTracer	38.8	58.7	66.8	47.6	27.2
SWT	BugLocator	33.6	67.3	75.5	48.0	41.5
	BRTracer	37.8	74.5	81.6	53.6	46.8

TABLE III: Overall Effectiveness Considering SBRs

Subject	Approach	Top N (%)			MRR	MAP (%)
		N=1	N=5	N=10		
Eclipse	BugLocator	29.4	52.9	62.8	40.7	30.7
	BRTracer	32.6	55.9	65.2	43.4	32.7
AspectJ	BugLocator	26.5	51.0	62.9	38.8	22.3
	BRTracer	39.5	60.5	68.9	49.1	28.6
SWT	BugLocator	35.7	69.3	79.5	50.2	44.5
	BRTracer	46.9	79.6	88.8	59.5	53.0

First, our approach outperforms BugLocator in all circumstances. This observation indicates that our approach provides a clear improvement in the performance of bug localization.

Second, both BugLocator and our approach typically do not achieve very high percentage numbers in Eclipse project. We suspect the reason to be the intrinsic difficulty of the target problem. The fuzzy nature of different descriptions in different bug reports may prevent any approach from retrieving source code files very accurately. Furthermore, when there are thousands of or even tens of thousand files, it is almost impossible for any approach to retrieve the correct files by chance. Therefore, we argue that the improvement of our approach over BugLocator should be a substantial improvement although the improvement is small in magnitude.

Table III depicts the overall effectiveness of BRTracer considering similar bug reports (SBRs). From this table, we have the following observations.

First, our approach is also able to outperform BugLocator in all circumstances, when both approaches use the same way to deal with similar bug reports. In general, the differences between our approach and BugLocator when considering similar bug reports are similar to those without considering similar bug reports. In some cases, when considering similar bug reports, the differences become significantly larger. For example, compared with Table II, our approach gains more improvement in SWT project under all the three metrics. This observation indicates that our two techniques (i.e., segmentation and stack-trace analysis) for boosting bug localization may be quite compatible with existing heuristics (e.g., utilizing similar bug reports).

Second, for either BugLocator or our approach, considering similar bug reports would typically improve the corresponding approach. This observation confirms that similar bug reports could be a reliable information source for bug-report-oriented fault localization.

We further use a statistical test to check whether the improvement is significant. In particular, we use Sign Test on the basis of the TNRF metric. Let n_+ (n_-) be the number of cases where BRTracer (BugLocator) ranks the correct file

[9] In fact, we directly used the data related to bug reports that THE AUTHORS OF [1] provided for us.

[10] http://brtracer.sourceforge.net

higher than BugLocator (BRTracer) for a specific bug report, and $N = n_+ + n_-$. We further drop all the cases for which neither BRTracer nor BugLocator is able to rank the buggy file within top 10 because we regard both approaches as equally ineffective in these cases.

TABLE IV: Sign Test Result without Considering SBRs

Subject	n_+	n_-	N	p
Eclipse	718	533	1251	<0.0001 (reject)
AspectJ	91	33	124	<0.0001 (reject)
SWT	30	16	46	0.0541

TABLE V: Sign Test Result Considering SBRs

Subject	n_+	n_-	N	p
Eclipse	686	578	1264	0.00262 (reject)
AspectJ	84	41	125	0.000172 (reject)
SWT	33	12	45	0.00246 (reject)

The Sign Test results are depicted in Tables IV and V. As shown in the tables, the null hypothesis between our BRTracer and BugLocator are rejected at the α=0.05 level for 5 out of 6 cases. The only exception is found in SWT project. As reported in Table I, the number of bug reports in SWT project is 98, which is too small to derive statistically sound results. For example, if $n_+ = 30$ and $n_- = 15$, the resulted p would be 0.0357, which rejects the null hypothesis. Thus, we argue that our approach is able to significantly outperform BugLocator on all the three subjects either with or without considering similar bug reports.

2) RQ2: Effectiveness of Segmentation and Stack-Trace Analysis: Table VI depicts the effectiveness of segmentation without considering similar bug reports (SBRs). In this table, we use "Segmentation" to denote our approach using only segmentation. From Table VI, we observe that, although the improvement of using only segmentation is less than that of using both techniques, using only segmentation is still able to outperform BugLocator for all the three metrics without considering similar bug reports.

Table VII depicts the effectiveness of using only segmentation when considering similar bug reports. From Table VII, we have the following observations.

First, following the same trend for not considering similar bug reports, using segmentation is also able to outperform BugLocator when both approaches use the same way to deal with similar bug reports.

Second, compared with Table VI, the differences between our approach using only segmentation and BugLocator when considering similar bug reports become even bigger than the differences when not considering similar bug reports. For example, improvement in SWT under the metric of "Top 1 Rank of Files" changes from 1.0 percentage points ($34.6\% - 33.6\%$) to 10.2 percentage points ($45.9\% - 35.7\%$). This indicates that segmentation is compatible with and even a boosting factor for the use of similar bug reports.

Tables VIII and IX depict the effectiveness of using only stack-trace analysis, i.e., using $rVSM(x)$ (which is used in BugLocator) instead of $rVSM_{seg}(x)$ in Formula 5 and 6. In these two tables, we use "Stack-Trace" to denote our approach using only stack-trace analysis. From these tables, we have the following observations.

TABLE VI: Effectiveness of Segmentation without Considering SBRs

Subject	Approach	Top N (%)			MRR	MAP (%)
		N=1	N=5	N=10		
Eclipse	BugLocator	25.8	47.9	58.2	36.5	27.5
	Segmentation	27.2	49.9	60.6	38.0	28.6
AspectJ	BugLocator	24.1	47.2	60.4	35.1	19.8
	Segmentation	30.4	50.0	62.2	39.8	22.4
SWT	BugLocator	33.6	67.3	75.5	48.0	41.5
	Segmentation	34.6	72.4	79.5	50.7	44.4

TABLE VII: Effectiveness of Segmentation Considering SBRs

Subject	Approach	Top N (%)			MRR	MAP (%)
		N=1	N=5	N=10		
Eclipse	BugLocator	29.4	52.9	62.8	40.7	30.7
	Segmentation	30.5	54.2	64.0	41.6	31.1
AspectJ	BugLocator	26.5	51.0	62.9	38.8	22.3
	Segmentation	31.1	54.5	67.1	42.2	24.0
SWT	BugLocator	35.7	69.3	79.5	50.2	44.5
	Segmentation	45.9	76.5	85.7	58.2	51.6

First, similar to using only segmentation, using only stack-trace analysis would also improve BugLocator either with or without considering similar bug reports under all the three metrics.

Second, our stack-trace analysis is generally compatible with the use of similar bug reports, because when considering similar bug reports, our stack-trace analysis can achieve a similar improvement over BugLocator.

Furthermore, when comparing the overall effectiveness of our approach with the effectiveness of using only segmentation and the effectiveness of using only stack-trace analysis, we have the following observation. Segmentation and stack-trace analysis can typically complement each other for boosting bug localization, especially when not considering similar bug reports.

3) Summary of Main Findings: We summarize the main findings of our empirical evaluation as follows:

- Whether or not considering similar bug reports, our approach is able to significantly outperform BugLocator for all the three projects under all the three metrics.
- Either segmentation or stack-trace analysis is an effective technique to boost bug localization and both techniques are compatible with the use of similar bug reports.
- Segmentation and stack-trace analysis should better be used together, as they can complement each other.

G. Threats to Validity

The main threat to internal validity lies in the implementation of the experimented approaches. To reduce this threat, we used the version of BugLocator provided by its authors and carefully reviewed the code to implement our approach on top of BugLocator.

The main threat to external validity lies in the subject projects used in our empirical evaluation. As all the three projects are open source projects, our empirical results may be specific to these projects and thus not generalizable to projects other than open source projects. However, recent work by Ma et al. [18] shows that a large portion of issues are often reported by commercial developers, which indicates that defects in open

TABLE VIII: Effectiveness of Stack-Trace Analysis without Considering SBRs

Subject	Approach	Top N (%)			MRR	MAP
		N=1	N=5	N=10		(%)
Eclipse	BugLocator	25.8	47.9	58.2	36.5	27.5
	Stack-Trace	28.2	50.1	59.6	38.8	29.4
AspectJ	BugLocator	24.1	47.2	60.4	35.1	19.8
	Stack-Trace	34.2	56.2	65.7	44.2	25.2
SWT	BugLocator	33.6	67.3	75.5	48.0	41.5
	Stack-Trace	36.7	70.4	77.5	51.1	44.2

TABLE IX: Effectiveness of Stack-Trace Analysis Considering SBRs

Subject	Approach	Top N (%)			MRR	MAP
		N=1	N=5	N=10		(%)
Eclipse	BugLocator	29.4	52.9	62.8	40.7	30.7
	Stack-Trace	31.8	54.8	64.0	42.8	32.5
AspectJ	BugLocator	26.5	51.0	62.9	38.8	22.3
	Stack-Trace	37.4	59.0	68.5	47.7	27.7
SWT	BugLocator	35.7	69.3	79.5	50.2	44.5
	Stack-Trace	38.7	72.4	81.6	53.3	47.2

source projects are likely to stem from commercial projects. Further reduction of this threat may involve more projects, especially commercial projects. Furthermore, like most other IR-based fault localization approaches, we use the same set of source code in the evaluation of each project. Since the modifications between revisions are usually small, we deemed the threat posed by the differences between revisions to be acceptable.

The main threat to construct validity lies in the metrics used in our empirical evaluation. To reduce this threat, we used three different metrics in our empirical evaluation. All the three metrics are widely used in previous studies. As all the three metrics are based on recovered links between bug reports and source code files, inaccuracy in the recovery process may impact our empirical results. However, using recovered links between bug reports and source code files for evaluation has been widely adopted by existing studies on bug-report-oriented fault localization [7], [8], [9], [1]. Furthermore, we used the same data set of recovered links that has been studied in a previous paper [1] in our experiments.

V. Future Work

In our approach, we use empirically determined values for β, l. However, due to different characteristics of different projects, it might be better to use differentiated values of these parameters. In future work, we plan to investigate mechanisms to automatically choose suitable values for these parameters on the basis of analyzing the target project.

In our approach, our segmentation strategy does not consider the characteristics of source code (e.g., code structures). Further consideration of code structures in segmentation (such as putting highly related code portions into one segment) may further improve our approach. Thus, we also plan to extend our research in this direction in future work.

Our approach uses a function with respect to the length of each source code file to favor larger files. As the ultimate reason is that larger files are far more likely to contain faults, in future work, we plan to investigate functions in respect to some fault-prediction techniques to directly favor files more likely to contain faults.

We adopt a lightweight but coarse analysis of source-code files in our stack-trace analysis. In fact, heavier analysis (e.g., language-specific syntactic and semantic analysis) may further improve the precision of analysis to benefit our approach, but such an analysis may become too time-consuming due to a large number of files potentially requiring analysis. In future work, we plan to investigate better trade-offs between time-consumption and analysis precision.

Our approach contains two distinctive techniques, which are complementary to each other for boosting the retrieval process according to our empirical results. However, the straightforward combination of the two technique in our approach may not fully exploit the strengths of both of them. In future work, we plan to investigate other ways to combine the two techniques.

VI. Related Work

A. Progress after BugLocator

In parallel with our work, several approaches have been proposed to improve IR-based bug localization with different heuristics. For example, Ye et al. [19] defined six features measuring the relationship between bug reports and source files, and defined a ranking model to combine them, using a learning-to-rank technique. In one of their features, they utilized project API descriptions to narrow the lexical gap between bug reports and source files. Saha et al. [20] proposed an approach called BLUiR and gained significant improvement over BugLocator by replacing Vector Space Model with structured information retrieval model. Sisman et al. [21] proposed to use automatic query reformulation by injecting closely related words into the query. Le et al. [22] applied topic modeling to infer multiple abstraction levels from documents, and extended Vector Space Model to locate code units by computing similarities resulting from several abstraction levels. Tantithamthavorn et al. [23] proposed an approach to improve BugLocator by leveraging co-change histories, which relied on the assumption that files that have been changed together are prone to be fixed together again in the future. Davies and Roper [12] combined four kinds of information to locate bugs: (1) textual similarities between bug reports and source files, (2) similarities of bug reports, (2) number of previous bugs that caused by each method and (4) stack-trace information. Note that our utilization of stack-trace information is different from theirs in that we not only consider files that mentioned in the stack traces, but also take other closely related files into account.

It is worth noting that the two heuristics used in our approach are different from nearly all parallel approaches and thus in future it is promising to investigate the combination of our approach with these approaches to gain an even higher accuracy.

Furthermore, among all the approaches that share the same dataset with us, BLUiR [20] is reported to have the highest boost over BugLocator, and thus it is interesting to compare the boost of BLUiR with our approach. Basically, in Eclipse project, BRTracer and BLUiR achieve similar accuracy under all metrics (i.e., TNRF, MRR and MAP). In SWT project, BRTracer is noticeably inferior to BLUiR under MRR and MAP (e.g., 6.5 points lower in MRR). In AspectJ project,

BRTracer noticeably outperforms BLUiR under all metrics (e.g., 6.1 points higher in MRR). Thus, the two approaches achieve the same level of overall boost over BugLocator. Furthermore, both BRTracer and BLUiR get improvement after considering similar bug reports, and the improvement on BRTracer is mostly greater than on BLUiR. Therefore, our approach is more compatible with the usage of similar bug reports. Because the two approaches differ in performance among different projects and with/without similar bug reports, we assume that the improvement techniques used in BRTracer and BLUiR are orthogonal. Hopefully, we can achieve a higher accuracy by combining our heuristics with BLUiR. However, since the implementation of BLUiR has not been released to public yet, we leave it to our future work.

B. Bug-Report-Oriented Fault Localization

Apart from BugLocator, there exist some other lines of research on bug-report-oriented fault localization. Given a fault, bug-report-oriented fault localization aims to use textual description of the fault in the bug report obtained from a bug repository to localize the fault. Typically, such a technique may not pinpoint the buggy piece of code, but may provide developers with further information about how the bug relates to the source code. Since a developer unfamiliar with the code base may have very few clues from only a bug report, such an approach may provide the developer with more clues.

As a typical bug report contains only textual information, information retrieval (IR) plays a central role in existing approaches to bug-report-oriented fault localization. Thus, a main line of research is to investigate various models for IR. In fact, researchers have investigated almost all classic IR models [2], [3], [24], [7], [8]. Another main line of research is to utilize other information beside textual information (e.g., [9], [1]). Due to the nature of bug reports and corpora extracted for source code files, using textual information alone is difficult to be very accurate.

Recently Kim et al. [25] proposed a two-phase machine learning-based approach to predict files to be fixed. In the first phase, their model examines the bug reports and determines whether there is sufficient information in the bug reports. Then, the second phase prediction is performed only on the bug reports that the model believes to be predictable.

Different from existing research, the research reported in this paper focuses on segmentation and stack-trace analysis in bug-report-oriented fault localization. Furthermore, our research is actually generic to existing research as both segmentation and stack-trace analysis can be easily adapted to combine with other approaches besides BugLocator.

C. Feature Location

Feature location aims to locate the source code related to each functionality. Intuitively, such an approach may be used to locate all the code that is functionally related to a bug report. In fact, some researchers (e.g., Poshyvanyk et al. [2], [3] and Gay et al. [4]) did use approaches to feature location to handle bug reports as a way to evaluate those approaches. However, the nature of feature location determines that such an approach would typically find more code than necessary to fix a bug. That is to say, an approach to feature location can hardly be very accurate for handling bug reports. Below, we briefly discuss some representative research on feature location. Please refer to Dit et al. [26] for a recent comprehensive survey on feature location.

According to Dit et al. [26], approaches to feature location typically rely on the following information: static information [27], [28], dynamic information [29], and textual information [30], [4]. There are also approaches that combine two or more types of information: Eisenbarth et al. [31] and Antoniol and Guéhéneuc [32] combine static information and dynamic information; Zhao et al. [33] and Hill et al. [34] combine static information and textual information; Poshyvanyk et al. [2], [3] and Liu et al. [35] combine dynamic information with textual information; and Eaddy et al. [36] combine all the three types of information. Note that approaches using only static information may have to rely on human intervention to link features to source code units, while combining static information with textual information may naturally remove the reliance on human intervention.

D. Bug-Report Analysis

Due to the need to efficiently handle a large number of bug reports and the wealth of information embedded in bug reports, there has been a large amount of research around bug reports in recent years. As summarized in [37], a significant portion of research (e.g., [38], [39], [40], [41], [42]) focuses on detecting and utilizing duplicate bug reports. Anvik et al. [43] and Kanwal et al. [44] proposed techniques to identify suitable developers for handling bug reports. Bettenburg et al. [45] empirically identified some guidelines for bug reporters to submit quality bug reports. Rastkar et al. [46] proposed a technique to distill essential information from verbose bug reports.

However, as techniques proposed for bug-report analysis typically do not consider how bug reports are related to source code, none of those techniques can be directly used or adapted in a straightforward way for bug-report-oriented fault localization.

VII. Conclusion

When matching a bug report against source code files, larger files may be more likely to be affected by noisy matches. Furthermore, bug reports may contain descriptions with direct pointers to possible faulty files, but information retrieval may be difficult to deal with those descriptions accurately. In this paper, we have proposed two techniques (i.e., segmentation and stack-trace analysis) to deal with these two issues. First, we divide the corpus extracted from each source code file into a series of segments and use the segment having the highest similarity with the bug report to substitute the file. Second, we identify names of explicitly-described possible faulty files in bug reports and provide more favor to these files and files directly related to these files. We implemented our approach (named BRTracer) via combining both techniques on top of BugLocator and performed an empirical evaluation to compare BRTracer against BugLocator. Our empirical results demonstrate that BRTracer can significantly outperform BugLocator on all the three projects under each metric either with or without considering similar bug reports. Our empirical results further demonstrate that segmentation and stack-trace analysis

can complement each other for boosting bug-report-oriented fault localization.

ACKNOWLEDGMENT

This work is supported by the National Basic Research Program of China No. 2011CB302604, the High-Tech Research and Development Program of China under Grant No.2013AA01A605 and the National Natural Science Foundation of China under Grant No. 61202071, 61272157,61121063, 61228203, 61272089.

REFERENCES

[1] 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. ICSE*, 2012, pp. 14–24.

[2] D. Poshyvanyk, Y.-G. Guéhéneuc, A. Marcus, G. Antoniol, and V. Rajlich, "Combining probabilistic ranking and latent semantic indexing for feature identification," in *Proc. ICPC*, 2006, pp. 137–146.

[3] ——, "Feature location using probabilistic ranking of methods based on execution scenarios and information retrieval," *IEEE TSE*, vol. 33, no. 6, pp. 420–432, June 2007.

[4] G. Gay, S. Haiduc, A. Marcus, and T. Menzies, "On the use of relevance feedback in IR-based concept location," in *Proc. ICSM*, 2009, pp. 351–360.

[5] C. D. Manning, P. Raghavan, and H. Schütze, *Introduction to Information Retrieval*. Cambridge University Press, 2008.

[6] W. B. Croft, D. Metzler, and T. Strohman, *Search Engines: Information Retrieval in Practice*. Addison-Wesley, 2010.

[7] S. Lukins, N. Kraft, and L. Etzkorn, "Bug localization using latent dirichlet allocation," *IST*, vol. 52, no. 9, pp. 972–990, September 2010.

[8] S. Rao and A. Kak, "Retrieval from software libraries for bug localization: A comparative study of generic and composite text models," in *Proc. MSR*, 2011, pp. 43–52.

[9] P. Bangcharoensap, A. Ihara, Y. Kamei, and K. Matsumoto, "Locating source code to be fixed based on initial bug reports - A case study on the Eclipse project," in *Proc. IWESEP*, 2012, pp. 10–15.

[10] S. Kim, T. Zimmermann, E. Whitehead Jr., and A. Zeller, "Predicting faults from cached history," in *Proc. ICSE*, 2007, pp. 489–498.

[11] N. Fenton and N. Ohlsson, "Quantitative analysis of faults and failures in a complex software system," *IEEE TSE*, vol. 26, no. 8, pp. 797–814, 2000.

[12] S. Davies and M. Roper, "Bug localisation through diverse sources of information," in *Proc. ISSREW*. IEEE, 2013, pp. 126–131.

[13] A. Schröter, N. Bettenburg, and R. Premraj, "Do stack traces help developers fix bugs?" in *Proc. MSR*, 2010, pp. 118–121.

[14] V. Dallmeier and T. Zimmermann, "Extraction of bug localization benchmarks from history," in *Proc. ASE*, 2007, pp. 433–436.

[15] E. M. Voorhees, "Trec-8 question answering track report," in *Proc. 8th Text Retrieval Conference*, 1999, pp. 77–82.

[16] C. D. Manning and H. Schütze, *Foundations of Statistical Natural Language Processing*. MIT Press, 1999.

[17] A. Bachmann and A. Bernstein, "Software process data quality and characteristics: A historical view on open and closed source projects," in *Proc. IWPSE-Evol*, 2009, pp. 119–128.

[18] X. Ma, M. Zhou, and D. Riehle, "How commercial involvement affects open source projects: three case studies on issue reporting," *Science China Information Sciences*, vol. 56, no. 8, pp. 1–13, 2013.

[19] X. Ye, R. Bunescu, and C. Liu, "Learning to rank relevant files for bug reports using domain knowledge," in *Proc. FSE*, 2014, pp. 66–76.

[20] R. K. Saha, M. Lease, S. Khurshid, and D. E. Perry, "Improving bug localization using structured information retrieval," in *Proc. ASE*, 2013, pp. 345–355.

[21] B. Sisman and A. C. Kak, "Assisting code search with automatic query reformulation for bug localization," in *Proc. MSR*, 2013, pp. 309–318.

[22] T.-D. B. Le, S. Wang, and D. Lo, "Multi-abstraction concern localization," in *Proc. ICSM*, 2013, pp. 364–367.

[23] C. Tantithamthavorn, A. Ihara, and K. ichi Matsumoto, "Using co-change histories to improve bug localization performance," in *Proc. SNPD*, 2013, pp. 543–548.

[24] S. Deerwester, "Improving information retrieval with latent semantic indexing," in *Proc. ASIS*, 1988, pp. 36–40.

[25] D. Kim, Y. Tao, S. Kim, and A. Zeller, "Where should we fix this bug? a two-phase recommendation model," *IEEE TSE*, vol. 39, no. 11, pp. 1597–1610, 2013.

[26] B. Dit, M. Revelle, M. Gethers, and D. Poshyvanyk, "Feature location in source code: A taxonomy and survey," *Journal of Software: Evolution and Process*, vol. 25, no. 1, pp. 53–95, January 2013.

[27] K. Chen and V. Rajlich, "Case study of feature location using dependence graph," in *Proc. IWPC*, 2000, pp. 241–249.

[28] M. P. Robillard and G. C. Murphy, "Concern graphs: Finding and describing concerns using structural program dependencies," in *Proc. ICSE*, 2002, pp. 406–416.

[29] N. Wilde and M. Scully, "Software reconnaissance: Mapping program features to code," *Journal of Software Maintenance: Research and Practice*, vol. 7, no. 1, pp. 49–62, 1995.

[30] A. Marcus, A. Sergeyev, V. Rajlich, and J. Maletic, "An information retrieval approach to concept location in source code," in *Proc. WCRE*, 2004, pp. 214–223.

[31] T. Eisenbarth, R. Koschke, and D. Simon, "Locating features in source code," *IEEE TSE*, vol. 29, no. 3, pp. 210–224, March 2003.

[32] G. Antoniol and Y. Guéhéneuc, "Feature identification: A novel approach and a case study," in *Proc. ICSM*, 2005, pp. 357–366.

[33] W. Zhao, L. Zhang, Y. Liu, J. Sun, and F. Yang, "SNIAFL: Towards a static non-interactive approach to feature location," in *Proc. ICSE*, 2004, pp. 293–303.

[34] E. Hill, L. Pollock, and K. Vijay-Shanker, "Exploring the neighborhood with Dora to expedite software maintenance," in *Proc. ASE*, 2007, pp. 14–23.

[35] D. Liu, A. Marcus, D. Poshyvanyk, and V. Rajlich, "Feature location via information retrieval based filtering of a single scenario execution trace," in *Proc. ASE*, 2007, pp. 234–243.

[36] M. Eaddy, A. V. Aho, G. Antoniol, and Y. Guéhéneuc, "CERBERUS: Tracing requirements to source code using information retrieval, dynamic analysis, and program analysis," in *Proc. ICPC*, 2008, pp. 53–62.

[37] J. Zhang, X. Wang, D. Hao, B. Xie, L. Zhang, and H. Mei, "A survey on bug-report analysis," Peking University, Tech. Rep. PKU-SEI-2014-07-01, 2014. [Online]. Available: http://sei.pku.edu.cn/~zhanglu/Download/PKUSEI20140701.pdf

[38] X. Wang, L. Zhang, T. Xie, J. Anvik, and J. Sun, "An approach to detecting duplicate bug reports using natural language and execution information," in *Proc. ICSE*, 2008, pp. 461–470.

[39] N. Bettenburg, R. Premraj, T. Zimmermann, and S. Kim, "Duplicate bug reports considered harmful... really?" in *Proc. ICSM*, 2008, pp. 337–345.

[40] C. Sun, D. Lo, X. Wang, J. Jiang, and S. Khoo, "A discriminative model approach for accurate duplicate bug report retrieval," in *Proc. ICSE*, 2010, pp. 45–54.

[41] J. Zhou and H. Zhang, "Learning to rank duplicate bug reports," in *Proc. CIKM*, 2012, pp. 852–861.

[42] A. T. Nguyen, T. T. Nguyen, T. N. Nguyen, D. Lo, and C. Sun, "Duplicate bug report detection with a combination of information retrieval and topic modeling," in *Proc. ASE*, 2012, pp. 70–79.

[43] J. Anvik, L. Hiew, and G. C. Murphy, "Who should fix this bug?" in *Proc. ICSE*, 2006, pp. 361–370.

[44] J. Kanwal and O. Maqbool, "Bug prioritization to facilitate bug report triage," *Journal of Computer Science and Technology*, vol. 27, no. 2, pp. 397–412, 2012.

[45] N. Bettenburg, S. Just, A. Schr oter, C. Weiss, R. Premraj, and T. Zimmermann, "What makes a good bug report?" in *Proc. FSE*, 2008, pp. 308–318.

[46] S. Rastkar, G. C. Murphy, and G. Murray, "Summarizing software artifacts: a case study of bug reports," in *Proc. ICSE*, 2010, pp. 505–514.