6-2012

Where should the bugs be fixed? More accurate information retrieval-based bug localization based on bug reports

Jian ZHOU
Tsinghua University

Hongyu ZHANG
Tsinghua University

David LO
Singapore Management University, davidlo@smu.edu.sg

Citation

ZHOU, Jian; ZHANG, Hongyu; and LO, David. Where should the bugs be fixed? More accurate information retrieval-based bug localization based on bug reports. (2012). *ICSE 2012: 34th International Conference on Software Engineering, Zurich, June 2-9*. 14-24.
Available at: https://ink.library.smu.edu.sg/sis_research/1531

SBFL: thu thập dữ liệu thực thi, xác định kết quả các test cases, xây dựng ma trận thực thi, sau đó xác định nơi xảy ra lỗi dựa vào ma trận thực thi (Và vì thế yêu cầu phải chạy chương trình)

LSI: Xây dựng ma trận thuật ngữ (cột dọc), và tài liệu(cột ngang), sau đó áp dụng Singular Value Decomposition giúp phân tích cấu trúc tiềm ẩn của dữ liệu văn bản, cuối cùng ánh xạ tài liệu và truy vấn vào không gian ngữ nghĩa

Relevance feedback: ví dụ tìm kiếm "machine learning", nhưng một số kết quả không liên quan đến chủ đề bạn quan tâm.Bạn chọn một số bài báo phù hợp.Hệ thống sử dụng các từ khóa trong các bài báo đó để mở rộng truy vấn ban đầu.Lần tìm kiếm tiếp theo sẽ chính xác hơn.

Where Should the Bugs Be Fixed?

More Accurate Information Retrieval-Based Bug Localization Based on Bug Reports

Jian Zhou[1], Hongyu Zhang[1,*] and David Lo[2]
[1]School of Software, Tsinghua University, Beijing 100084, China
Tsinghua National Laboratory for Information Science and Technology (TNList)
zhoujian1286@yahoo.com.cn, hongyu@tsinghua.edu.cn
[2]School of Information Systems, Singapore Management University, Singapore
davidlo@smu.edu.sg

***Abstract*—For a large and evolving software system, the project team could receive a large number of bug reports. Locating the source code files that need to be changed in order to fix the bugs is a challenging task. Once a bug report is received, it is desirable to automatically point out to the files that developers should change in order to fix the bug. In this paper, we propose BugLocator, an information retrieval based method for locating the relevant files for fixing a bug. BugLocator ranks all files based on the textual similarity between the initial bug report and the source code using a revised Vector Space Model (rVSM), taking into consideration information about similar bugs that have been fixed before. We perform large-scale experiments on four open source projects to localize more than 3,000 bugs. The results show that BugLocator can effectively locate the files where the bugs should be fixed. For example, relevant buggy files for 62.60% Eclipse 3.1 bugs are ranked in the top ten among 12,863 files. Our experiments also show that BugLocator outperforms existing state-of-the-art bug localization methods.**

***Keywords*-bug localization; information retrieval; feature location; bug reports**

I. Introduction

Software quality is vital for the success of a software project. Although many software quality assurance activities (such as testing, inspection, static checking, etc) have been proposed to improve software quality, in reality software systems are often shipped with defects (bugs). For a large and evolving software system the project team could receive a large number of bug reports over a long period of time. For example, around 4414 bugs were reported for the Eclipse project in 2009.

Tác giả nêu vấn đề khó khăn trong việc sửa lỗi khi số lượng báo cáo lỗi quá lớn, hoặc mã nguồn chứa quá nhiều file

Once a bug report is received and confirmed, the project team should locate the source code files that need to be changed in order to fix the bug. However, it is often costly to manually locate the files to be changed based on the initial bug reports, especially when the numbers of files and reports are large. For a large project consisting of hundreds or even thousands of files, manual bug localization is a painstaking and time-consuming activity. As a result, the bug fix time is often prolonged, maintenance cost is increased and customer satisfaction rate is hampered.

In recent years, some researchers have applied information retrieval techniques to automatically search for relevant files based on bug reports [16, 25, 31, 32]. They treat an initial bug report as a query and rank the source code files by their relevance to the query. The developers can then examine the returned files and fix the bug. These methods are *information retrieval based bug localization* methods. Unlike spectrum-based fault localization techniques [1, 18, 19, 22, 23], information retrieval (IR) based bug localization does not require program execution information (such as passing and failing traces). They locate the bug-relevant files based on initial bug reports.

Giới thiệu IR, so với SBFL thì không cần chạy chương trình để xác định lỗi.

Many of the existing IR-based bug localization methods are proposed in the context of feature/concept location, using a small number of selected bug reports [16, 24, 31]. For example, Poshyvanyk et al. proposed a feature location method called PROMESIR, which utilizes an information-retrieval technique (Latent Semantic Indexing) and a probabilistic ranking technique [31]. They applied their method to locate 3 bugs in Eclipse and 5 bugs in Mozilla. Gay et al. proposed an approach to augment IR-based concept location via an explicit relevance feedback (RF) mechanism [16]. They applied their bug localization approach on 9 bug reports. Recently, Lukins et al. performed a study on applying LDA (Latent Dirichlet Allocation) to search for bug-related methods and files [25]. They used 322 bugs across 25 versions of three projects (Eclipse, Mozilla and Rhino) for the evaluation. In each version, only a small number of bugs were selected (less than 20 on average). Besides the problem of small-scale evaluations, the performance of the existing bug localization methods can be further improved too. For example, using Latent Dirichlet Allocation (LDA), only buggy files for 22% of Eclipse 3.1 bug reports are ranked in the top 10 [25]. More detailed discussions about the current methods and their limitations are given in the next section.

Tác giả giới thiệu một số cách tiếp cận sử dụng IR, tuy nhiên các tiếp cận trên vẫn còn đang gặp hạn chế về độ chính xác với một số phương pháp chỉ đạt 22% trong top 10 kết quả là đúng.

In this paper, we propose BugLocator, a new method that can automatically search for relevant buggy files based on initial bug reports. We propose a revised Vector Space Model (rVSM) to rank all source code files based on an initial bug report. In rVSM, we take the document length into

[*] corresponding author

978-1-4673-1067-3/12/$31.00 © 2012 IEEE

ICSE 2012, Zurich, Switzerland

Tác giả giới thiệu một phương pháp tiếp cận mới sử dụng phiên bản được sửa đổi của Vector Space Model (rVSM), kết hợp điều chỉnh thứ hạng thu được dựa vào các thông tin lỗi trước đó và đạt được cải thiện đáng kể so với các phương pháp hiện có

consideration, which could optimize the classic VSM model for bug localization. We also adjust the obtained ranks by using information of the similar bugs that have been fixed before. We have evaluated BugLocator on four open source projects (Eclipse, AspectJ, SWT and ZXing) of different sizes, with a total of more than 3,000 bugs. The evaluation results show that BugLocator is effective. For example, buggy files for 62.6% of Eclipse 3.1 bugs are ranked in top 10. On average, the percentages of bugs whose relevant files are ranked in top 1, top 5 and top 10 are above 30%, 50% and 60%, respectively, confirming the effectiveness of the proposed approach. Our experiments also show that BugLocator outperforms existing bug localization methods using Vector Space Model (VSM) [32], Latent Dirichlet Allocation (LDA) [25], Latent Semantic Indexing **(**LSI) [30, 31], and Smoothed Unigram Model (SUM) [32].

2 đóng góp quan trọng của tác giả bao gồm đề xuất 1 phương pháp tìm kiếm lỗi với độ chính xác tăng đáng kể và thực hiện đánh giá quy mô lớn

The contributions of our work are as follows:

- We propose BugLocator, a new bug localization method that can perform better than the existing methods. In BugLocator, We design a new VSM method that can effectively retrieve relevant buggy files given a query bug report. Our method also utilizes information about similar bugs that have been fixed before to improve the ranking performance.
- We perform a large-scale evaluation of the bug localization techniques. We have run BugLocator on more than 3,000 bugs in total, which is much larger than the scale of experiments conducted in prior studies.

We believe our method can help project teams locate files where the bugs should be fixed. Automating bug localization work can help reduce maintenance cost and improve customer satisfaction.

The organization of the paper is as follows. In Section II, we describe the background of this work. In Section III, we describe the proposed BugLocator approach. Section IV describes our experimental design, and Section V shows and discusses the experimental results. Section VI gives the threats to validity. We discuss the related work in Section VII and conclude the paper in Section VIII.

II. Background

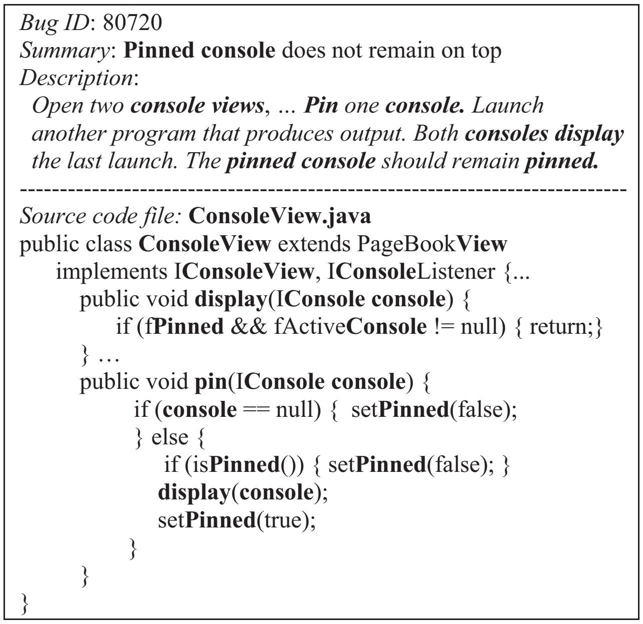
A. Bug Localization Example

Tác giả nêu ra một ví dụ về bug report khi mà file báo cáo bug chứa các từ vựng có sự liên quan cao với file code, được chỉ rõ trong hình 1

In this section, we present an example to illustrate information retrieval based bug localization approach. Figure 1 shows a real bug report[1] (ID: 80720) for Eclipse 3.1. Once this report is received, the developer needs to locate relevant files among more than ten thousands Eclipse source files in order to fix this bug. We find that the bug report (including bug summary and description) contains many words such as *pin(pinned)*, *console*, *view*, *display*, etc. Therefore, this bug is related to features about console view. In Eclipse 3.1, there is a source code file called *ConsoleView.java*, which also contains many occurrences of the similar words. Figure 1 shows a good match between the bug report and the source code.

We can treat the bug report and the source code files as text documents, and compute the textual similarity between them. For a corpus of files, we can rank the files based on each file's textual similarity to the bug report. Developers can then investigate the files one by one from the beginning of the ranked list until relevant buggy files are found. In this way, files relevant to the bug report can be quickly located. Clearly, the goal of bug localization is to rank the buggy files as high as possible in the list.

Tác giả nêu ra phương pháp tìm kiếm các code lỗi dựa vào xếp hạng độ tương đồng 2 văn bản source code và bug report



Bug ID: 80720
Summary: **Pinned console** does not remain on top
Description:
*Open two **console views**, ... **Pin** one **console.** Launch another program that produces output. Both **consoles display** the last launch. The **pinned console** should remain **pinned.***

Source code file: **ConsoleView.java**

```
public class ConsoleView extends PageBookView
    implements IConsoleView, IConsoleListener {...
      public void display(IConsole console) {
          if (fPinned && fActiveConsole != null) { return;}
      } …
      public void pin(IConsole console) {
            if (console == null) {  setPinned(false);
            } else {
               if (isPinned()) { setPinned(false); }
               display(console);
               setPinned(true);
            }
      }
}
```

Figure 1. A bug report and its relevant source code file

B. General Bug Localization Process

Phần này tác giả nêu ra hướng tiếp cận phổ biến của IR

Before presenting our approach, we describe a common bug localization process, which consists of four steps: corpus creation, indexing, query construction, and retrieval & ranking.

Tác giả nêu ra bước đầu tiên của quá trính là coupus creation - xử lý và lấy các từ vựng được định nghĩa trong file code

Corpus creation: This step performs lexical analysis for each source code file and creates a vector of lexical tokens. Some tokens, such as keywords (e.g., int, double, char, etc), separators, operators are common to all programs and are removed. English "stop words" (e.g., 'a', 'the', etc.) are also removed. Many variables defined in a program are actually a concatenation of words. For example, the variable *TypeDeclaration* contains two words: "type" and "declaration". The variable *isCommitable* is composed of two words: "is" and "Commitable". These composite tokens are split into individual tokens. Many tokens have the same root form. For example, "delegating", "delegate" and "delegation" share the same root "delegat". The Porter Stemming algorithm[2] is applied to reduce a word to its root.

Indexing: After the corpus is created, all the files in the corpus are indexed. By using these indexes, one can locate files containing the words in a given query and then rank these files by their relevance.

Sau bước xử lý sẽ là bước lập chỉ mục giúp ta có thể định vị các tệp chứa các từ trong một truy vấn nhất định và xếp hạng các tệp dựa vào độ liên quan

[1] https://bugs.eclipse.org/bugs/show_bug.cgi?format=multiple&id=80720

[2] http://tartarus.org/martin/PorterStemmer/

Tiền xử lý câu truy vấn

Query Construction: Bug localization considers a bug report as a query, and uses it to search for relevant files in the indexed source code corpus. It extracts tokens from the bug title and description, removes stop words, stems each word, and forms the query.

Cuối cùng là truy vấn và xếp hạng truy vấn, cách tính điểm có thể có nhiều cách tiếp cận khác nhau

Retrieval and Ranking: Retrieval and ranking of relevant buggy files is based on the textual similarity between the query and each of the files in the corpus. Various approaches can be used to compute a relevance score for each file in the corpus given an input bug report.

C. *Information Retrieval Models Used in Exisiting Bug Localization Methods*

Đã có nhiều phương pháp đề ra trước bài báo này, tác giả sẽ nêu ra một số phương pháp mà tác giả thấy là quan trọng

Many bug localization approaches have been proposed. These approaches mainly differ in the retrieval and ranking of the results. There are many retrieval and ranking models that have been used in prior studies on IR-based bug localization. Due to space constraint, we just briefly describe some important ones here:

SUM: Smoothed Unigram Model (SUM) is a statistical model that fits a single multinomial distribution to the frequencies of words in each file in the corpus [27]. The unigram model (UM) derived directly from the word frequency counts may have some problems, especially when confronted with words that have not explicitly been seen before - the probabilities of that unseen words are zero. SUM smoothes the probability distributions by assigning non-zero probabilities to the unseen words [17, 36]. SUM was used for bug localization by Rao and Kak [32] and was found to be the best performing model.

Trong các phương pháp được nêu ra, VSM tốt hơn các phương pháp khác ngoại trừ SUM

LDA: Latent Dirichlet Allocation (LDA) is a generative probabilistic model for collections of discrete data such as text corpora [11]. It is a Bayesian model, which extracts latent topics from a collection of documents. Each topic is a collection of tokens with attached probabilities. Each document is represented by a probabilistic mixture of topics. It was used for Lukins et al. [25] for bug localization.

LSI: Latent Semantic Indexing (LSI) [12], also called latent semantic analysis (LSA), is an indexing and retrieval method that can identify the relationship between the terms and concepts contained in an unstructured collection of text by using mathematical techniques such as Singular Value Decomposition (SVD). This method was used by Poshyvanyk et al. for bug localization [30, 31].

VSM: In Vector Space Model (VSM), each document is expressed as a vector of token weights typically computed as a product of token frequency and inverse document frequency of each token [26]. Cosine similarity is widely used to determine how close the two vectors are. Rao and Kak [32] evaluated the performance of VSM model in bug localization and found that it is worse than SUM but better than other models (including LDA and LSI).

The existing methods have the following common limitations:

Các phương pháp hiện có tồn tại hạn chế như độ chính xác thấp và được thử nghiệm trên quy mô nhỏ

- **Low accuracy**: The performance of existing bug localization methods can be further improved. For example, using LDA, relevant files of only 22% Eclipse 3.1 bugs are ranked in the top 10 [25].
- **Small-scale experiments**: Many of the existing static bug localization methods only used a small number of selected bug reports in their evaluation.

III. The Proposed Approach

A. *Analysis of Bug Localization Problem*

Tác giả đề xuất một số góc nhìn giúp cải thiện kết quả bài toán

To improve bug localization performance, we leverage the following observations:

Source code files: A project's source code repository contains source code files. As illustrated in Figure 1, source code files may contain words that are similar to those occurring in the bug reports. Therefore, analyzing source code files can help determine the location where the bug has impact on, i.e., the buggy files.

Source code files có thể chứa các thông tin quan trọng.

Similar bugs: Once a new bug report is received, we can examine similar bugs that were reported and fixed before. The information on locations where past similar bugs were fixed could help us locate the relevant files for the new bug.

Nếu bug report đang xử lý có vẻ giống với lỗi đã được xử lý trước đó thì vị trí sửa lỗi trước đó có thể có thông tin giúp ích.

Software size: When two source files have similar scores, we need to determine which one should be ranked higher. From our experiences in software defect prediction [37] and from other people's work on quantitative analysis of fault distributions [14, 29], we know that statistically, larger files are more likely to contain bugs. Therefore for bug localization we need to assign higher scores to larger files.

Các file có kích thước lớn có khả năng cao là file lỗi (nên được đánh giá điểm cao hơn)

The source code file information has been used by existing bug localization methods [16, 25, 31, 32]. However, to our best knowledge, the information about similar bugs and software sizes has not been well utilized. During the design of our approach, we take these information into consideration.

Ngoại trừ góc nhìn đầu tiên đã được sử dụng, 2 cách tiếp cận dưới vẫn chưa được sử dụng và tác giả đã đưa 2 hướng đó vào mô hình

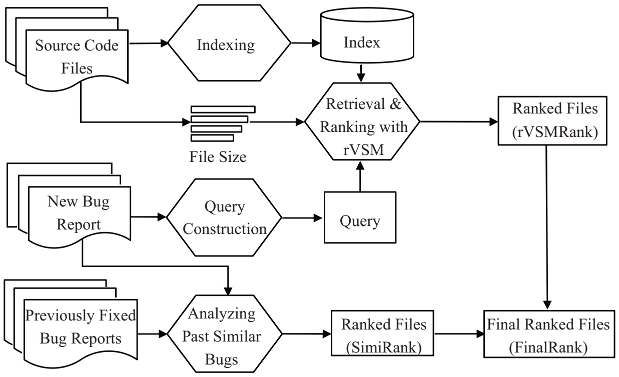


Figure 2. The overall structure of BugLocator

B. *The Overall Structure of BugLocator*

Figure 2 shows the overall structure of the proposed bug localization approach, BugLocator. When a new bug report is received, we treat it as a query and apply a revised Vector Space Model (rVSM) to search the source code repository. A rank of relevant files is returned from the query on source code files. For the new bug report, we also collect the similar bugs that have been fixed before, and rank the relevant files by analyzing past similar bugs and their fixes. Finally, we combine the ranks obtained from the query on source code files and from the analysis of past similar bugs, and return the users the combined ranks. The users can then examine the returned files in descending order to locate the bug. We describe the detailed procedures in the following subsections.

C. Ranking Based on Source Code Files

We consider source code files as a text corpus, and the initial bug report as a query. We can then apply information retrieval techniques to create a model for searching source code files based on the bug report. The similarity between each file and the bug report is computed. The files are then ranked by the similarity values and returned as output.

We propose a revised Vector Space Model (rVSM) to index and rank the source code files. In a classic VSM, the relevance score between a document *d* and a query *q* is computed as the cosine similarity between their corresponding vector representations:

$$Similarity(q,d) = \cos(q,d) = \frac{\overrightarrow{V_q} \bullet \overrightarrow{V_d}}{\left|\overrightarrow{V_q}\right|\left|\overrightarrow{V_d}\right|} \qquad (1)$$

, where $\overrightarrow{V_d}$ and $\overrightarrow{V_q}$ are a vector of term weights for the document *d* and query *q*, respectively. $\overrightarrow{V_q} \bullet \overrightarrow{V_d}$ represents the inner product of the two vectors. The term weight *w* is computed based on the *term frequency (tf)* and the *inverse document frequency (idf)*. The basic idea is that the weight of a term in a document is increasing with its occurrence frequency in this specific document and decreasing with its occurrence frequency in other documents. In classic VSM, *tf* and *idf* are defined as follows:

$$tf(t,d) = \frac{f_{td}}{\#terms}, idf(t) = \log(\frac{\#docs}{n_t}) \qquad (2)$$

, where f_{td} refers to the number of occurrences of a term *t* in document *d*, n_t refers to the number of documents that contain the term *t*, *#terms* represents the total number of terms in document *d*, and *#docs* represents the total number of documents in the corpus. Over the years, many variants of *tf(t,d)* have been proposed to improve the performance of the VSM model [26]. These include logarithm, augmented, and Boolean variants of the classic VSM. It is observed that *the logarithm variant* can lead to better performance [8, 13]:

$$tf(t,d) = \log(f_{td}) + 1 \qquad (3)$$

In rVSM, we use Equation (3) to define *tf*. Thus in Equation (1), each term weight *w* in the document vector $\overrightarrow{V_d}$ and its norm $\left|\overrightarrow{V_d}\right|$ are calculated as follows:

$$\begin{aligned} w_{t\in d} &= tf_{td} \times idf_t = (\log f_{td} + 1) \times \log(\frac{\#docs}{n_t}) \\ \left|\overrightarrow{V_d}\right| &= \sqrt{\sum_{t\in d} ((\log f_{td} + 1) \times \log(\frac{\#docs}{n_t}))^2} \end{aligned} \qquad (4)$$

In the similar way, we obtain the vector of term weights for the query $\overrightarrow{V_q}$ and its norm $\left|\overrightarrow{V_q}\right|$.

Classical VSM favours small documents during ranking. Long documents are often poorly represented because they have poor similarity values [15]. According to previous studies [14, 29, 37], larger source code files tend to have higher probability of containing a bug. Therefore we should rank larger files higher in the case of bug localization. We thus define a function *g* (Equation 5) to model the document length in rVSM:

$$g(\#terms) = \frac{1}{1 + e^{-N(\#terms)}} \qquad (5)$$

Equation (5) is a logistic function (i.e., an inverse logit function) that ensures that larger documents are given higher scores during ranking. We use Equation (5) to compute the length value for each source file according to the number of terms the file contains.

In Equation (5), we use the normalized value of #terms as the input to the exponential function e^{-x}. The normalization function is defined as follows:

Suppose that *X* is a set of data, x_{max} and x_{min} are the maximum and minimum data in *X*, the normalization value of any *x* in *X* is:

$$N(x) = \frac{x - x_{\min}}{x_{\max} - x_{\min}} \qquad (6)$$

Combining the above analysis, we thus propose a new scoring algorithm for rVSM as follows:

$$\begin{aligned} &rVSMScore(q,d) = g(\#term) \times \cos(q,d) \\ &= \frac{1}{1 + e^{-N(\#terms)}} \times \frac{1}{\sqrt{\sum_{t\in q} ((\log f_{tq} + 1) \times \log(\frac{\#docs}{n_t}))^2}} \\ &\times \frac{1}{\sqrt{\sum_{t\in d} ((\log f_{td} + 1) \times \log(\frac{\#docs}{n_t}))^2}} \\ &\times \sum_{t\in q\cap d} (\log f_{tq} + 1) \times (\log f_{td} + 1) \times \log\left(\frac{\#docs}{n_t}\right)^2 \end{aligned} \qquad (7)$$

Given a bug report, we use Equation (7) to determine the relevance scores (*rVSMScore*) between each source code file and the bug report. A ranked list (*rVSMRank*) can be obtained according to the scores (the first returned result has the highest score).

D. Ranking Based on Similar Bugs

For a new bug report, we also examine similar bugs that have been fixed before in order to adjust the rankings of the relevant files. The assumption here is that similar bugs tend to fix similar files. We propose a method for ranking relevant files based on similar bugs as follows:

We first construct a three-layer heterogeneous graph as shown in Figure 3. The top layer (layer 1) contains one node representing a newly reported bug *B*. The second layer contains nodes representing previously fixed bugs *S* that are similar to *B*. In our approach, we do not enforce a similarity threshold. A link between *B* and a bug in layer 2 indicates that there is a non-zero similarity value between their bug reports. The third layer contains nodes representing all source code files *F*. If a bug in layer 2 is fixed in a file in layer 3, a link between them is established, indicating that the bug has impact on the file.

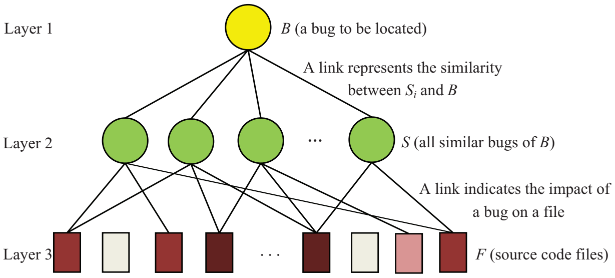

Layer 1
B (a bug to be located)
A link represents the similarity between S_i and *B*
Layer 2
...
S (all similar bugs of *B*)
A link indicates the impact of a bug on a file
Layer 3
. . .
F (source code files)

Figure 3. Heterogeneous Bug-File graph

The weight of each node in layer 2 (S_i) represents the degree of similarity between S_i and the newly reported bug *B*. This similarity is computed by Equation (1). The weight of each node in layer 3 (F_j) represents the degree of relevance between a source code file and the bug *B*, which is computed as follows:

$$SimiScore = \sum_{\substack{\text{All } S_i \text{ that} \\ \text{connect to } F_j}} (Similarity(B, S_i) / n_i) \qquad (8)$$

, where S_i is a node in layer 2 that connects to F_j, n_i is the total number of connections to layer 3 S_i has (i.e., the number of files that are modified to fix the bug S_i).

After computing the *SimiScore* for each file using Equation (8), we then rank all files based on the *SimiScore* values, and obtain a ranking of relevant files *SimiRank*.

E. Combining Ranks

Having computed the scores obtained from querying source code files (*rVSMScore*) and from similar bug analysis (*SimiScore*), we then combine these two scores for each file as follows:

$$\begin{aligned} FinalScore = {} & (1-\alpha) \times N(rVSMScore) \\ & + \alpha \times N(SimiScore) \end{aligned} \qquad (9)$$

, where α is a weighting factor and $0 \leqslant \alpha \leqslant 1$.The *FinalScore* is a weighted sum of *rVSMScore* and *SimiScore*. The source code files ranked by *FinalScore* in descending order are returned to users (*FinalRank*). Files that are ranked higher are the more relevant ones, i.e., more likely to contain the newly reported bug *B*.

Before we combine *rVSMScore* and *SimiScore*, we normalize them to the range of 0 to 1, using the normalization function defined in Equation (6).

The parameter α adjusts the weights of the two rankings. The value of α can be set empirically, our experience shows when α is between 0.2 and 0.3, the proposed method performs the best.

IV. Experimental Setup

A. Subject Systems

To evaluate the effectiveness of BugLocator, we use four open source projects as shown in Table I. All projects have complete bug database and change history, and have different numbers of bugs and source code files. We choose Eclipse[3] in our evaluation because it is a well-known large-scale open source system and it is widely used in empirical software engineering research. The AspectJ project is a part of the iBUGs public dataset provided by the University of Saarland[4] [9, 10]. It is also the subject used for evaluating various IR models for bug localization [32]. Both Eclipse and AspectJ use the Bugzilla bug tracking system and the CVS/SVN version control system. We also investigate the SWT[5] component of Eclipse, to evaluate the bug performance at the subproject level. To further evaluate the generality of our approach, we choose an Android project ZXing[6], which is maintained by Google's bug tracking system and version control system.

B. Data Collection

For each subject system, we collect its initial bug reports from the bug tracking system (such as BugZilla). To evaluate the bug localization performance, we only collect the bug reports of fixed bugs.

To establish the links between bug reports and source code files, we adopt the traditional heuristics proposed by Bachmann and Bernstein [5]:

1) Scan through the change logs for bug IDs in a given format (e.g. "issue 681", "bug 239" and so on).

2) Exclude all false-positive bug numbers (e.g. "r420", "2009-05-07 10:47:39 -0400" and so on).

3) Check if there are other potential bug number formats or false positive number formats, add the new formats and scan the change logs iteratively.

4) Check if potential bug numbers exist in the bug-tracking database with their status marked as fixed.

Based on these heuristics we mine the source code repository (such as CVS and SVN) for links between source code files and bug reports.

TABLE I. The Studied Projects

Project	Description	Study Period	#Fixed Bugs	#Source Files
Eclipse (v3.1)	An open development platform for Java	Oct 2004 - Mar 2011	3075	12863
SWT (v3.1)	An open source widget toolkit for Java	Oct 2004 - Apr 2010	98	484
AspectJ	An aspect-oriented extension to the Java programming language	Jul 2002 - Oct 2006	286	6485
ZXing	A barcode image processing library for Android applications	Mar 2010- Sep 2010	20	391

[3] http://www.eclipse.org

[4] http:// www.st.cs.uni-saarland.de/ibugs/

[5] http://www.eclipse.org/swt/

[6] http://code.google.com/p/zxing/

C. Research Questions

Our experiments are designed to address the following research questions:

***RQ1**: How many bugs can be successfully located by BugLocator?*

To answer this question, we run BugLocator on the four subject systems as described in Section IV.A. For each bug report, we first obtain the relevant files that have been modified to fix the bug using the method described in Section IV.B. We then check the ranks of these files in the query results returned by BugLocator. If the files are ranked in top 1, top 5 or top 10, we consider the report has been effectively localized. We perform the experiment for all bug reports and calculate the percentage of bugs that have been successfully located. We also compute the Mean Average Precision (MAP) and Mean Reciprocal Rank (MRR) measures (described in Section IV.D) to further evaluate bug localization performance.

***RQ2**: Does the revised Vector Space Model (rVSM) improve the bug localization performance?*

In Section III, we propose rVSM*,* a revised vector space model (Equation 7) for retrieving relevant files from source code repository. rVSM adjusts the ranks of large files and incorporates a more effective term-frequency variant. To evaluate the effectiveness of rVSM, we perform bug localization on the subject systems using classic and revised VSM, and compare the results.

***RQ3**: Does the consideration of similar bugs improve the bug localization performance?*

In Section III, we propose to use similar bugs to adjust the ranks obtained by rVSM. To evaluate the usefulness of the proposed similar bug analysis, we perform bug localization on the four subject systems with/without the rankings learned from past similar bugs. Furthermore, according to Equation (9), the parameter α adjusts the weights of the two rankings. When $\alpha = 0$, the final rank is only dependent on the queries of source code files. When the value of α is between 0 and 1, the final rank is a combination of two ranking results. In our experiments, we also evaluate the effect of different α values.

***RQ4**: Can BugLocator outperform other bug localization methods?*

Bug localization has attracted much research interest in recent years. In our experiments, we compare BugLocator to the bug localization methods implemented using the following IR techniques:

- LDA, which was used by Lukins et al. [25] for bug localization. Following their LDA configuration, in our experiment, for AspectJ, SWT and ZXing, we set K (the number of topics) to 100, α (the hyper-parameter for the per-document topic distribution) to 0.5 (this is the default value computed by the standard formula: 50/K), and β *(*the hyper-parameter for the per-topic word distribution) to 0.1, For Eclipse, as it is a large system consisting of many files, we set K to 500, α to 0.1 and β to 0.1, which can lead to a better performance. We use JGibbLDA[7], an open source tool written in Java, to implement the LDA model.
- SUM, which was used by Rao and Kak [32] for bug localization. In their study, SUM is shown to be the best IR model for bug localization, outperforming sophisticated models like LDA and LSI.
- VSM, which was also used by Rao and Kak [32] for bug localization. In their study, VSM was the second best IR approach for bug localization.
- LSI, which was used by Poshyvanyk et al. [30, 31] for bug localization. Previous experiments [25, 32] show that the performance of SUM, VSM or LDA is better than LSI.

Following Rao and Kak [32], we use KL divergence [21] to compute the similarity measures for LDA and SUM. We use the cosine similarity measure for VSM and LSI.

D. Evaluation Metrics

To measure the effectiveness of the proposed bug localization method, we use the following metrics:

- Top *N* Rank, which is the number of bugs whose associated files are ranked in the top *N* (*N*= 1, 5, 10) of the returned results. Given a bug report, if the top *N* query results contain at least one file at which the bug should be fixed, we consider the bug located. The higher the metric value, the better the bug localization performance.
- MRR (Mean Reciprocal Rank), which is a statistic for evaluating a process that produces a list of possible responses to a query [34]. The reciprocal rank of a query is the multiplicative inverse of the rank of the first correct answer. The mean reciprocal rank is the average of the reciprocal ranks of results of a set of queries Q:

$$MRR = \frac{1}{|Q|}\sum_{i=1}^{|Q|}\frac{1}{rank_i} \qquad (10)$$

The higher the MRR value, the better the bug localization performance.

- MAP (Mean Average Precision), which provides a single-figure measure of quality of information retrieval [26], when a query may have multiple relevant documents. The Average Precision of a single query (*AvgP*) is the average of the precision values obtained for the query, which is computed as follows:

$$AvgP_i = \sum_{i=1}^{M} \frac{P(j)\times pos(j)}{\text{number of positive instances}} \qquad (11)$$

, where *j* is the rank, *M* is the number of instances retrieved, *pos(j)* indicates whether the instance in the rank *j* is relevant or not. *P(j)* is the precision at the given cut-off rank *j* and is defined as follows:

[7] http://jgibblda.sourceforge.net

$$P(j) = \frac{\text{number of positive instances in top j positions}}{j} \qquad (12)$$

Then the MAP for a set of queries is the mean of the average precision values for all queries. In bug localization, a bug may be relevant to multiple files. We use MAP to measure the average performance of BugLocator for locating all relevant files. The higher the MAP value, the better the bug localization performance.

V. Experimental Results

A. Experimental Results for Research Questions

RQ1: *How many bugs can be successfully located by BugLocator?*

Table II shows the best bug localization results achieved by BugLocator for all subject systems. For 896 Eclipse bugs (29.14%), BugLocator successfully locates the relevant buggy source code files and ranks them as the top 1 among the returned results. For 1653 Eclipse bugs (53.76%), BugLocator ranks their relevant files within the top 5 of the returned results. For 1925 Eclipse bugs (62.60%), the relevant files can be found within the top 10 results.

TABLE II. The Performance of BugLocator

System	α	Top 1	Top 5	Top 10	MRR	MAP
ZXing	0.2	8 (40%)	12 (60%)	14 (70%)	0.50	0.44
SWT	0.2	39 (39.80%)	66 (67.35%)	80 (81.63%)	0.53	0.45
AspectJ	0.3	88 (30.77%)	146 (51.05%)	170 (59.44%)	0.41	0.22
Eclipse	0.3	896 (29.14%)	1653 (53.76%)	1925 (62.60%)	0.41	0.30

For AspectJ, there are total 6,485 Java source code files and 286 bugs. For 59.44% of the bugs (i.e., 170 bugs) the first relevant file is returned in the top 10, and for 51.05% of the bugs (i.e., 146 bugs) the first relevant file is returned in the top 5. The results indicate that our approach is effective in localizing bugs in AspectJ.

AspectJ was also the subject program investigated in [32]. In [32], the authors compared various information retrieval methods for bug localization and found that SUM performs the best (in terms of the percentage of bugs being successfully located and MAP). We compare the results of BugLocator with the SUM results given in [32], and find that BugLocator outperforms SUM (Figure 4). For example, using SUM, the relevant files of 19.59% bugs are returned as the top 1, while using BugLocator we can locate 30.77% bugs in the top 1 returned file. The MAP values for SUM and BugLocator are 0.14 and 0.22, respectively. In general, BugLocator improves the performance of SUM by 10%.

For SWT, Table II shows that for 39.80% of the bugs, BugLocator ranks their relevant Java source file as top 1; for 67.35% of the bugs, BugLocator ranks their relevant files within top 5; for 81.63% of the bugs, BugLocator ranks their relevant files within top 10. These ratios are higher than those of AspectJ and Eclipse.

For ZXing, BugLocator achieves similar good performance. The percentages of bugs whose relevant files are ranked top 1, top 5, and top 10 are 40%, 60%, and 70%, respectively.

In summary, the experimental results show that BugLocator can help locate a large percentage of bugs by examining a small number of source files.

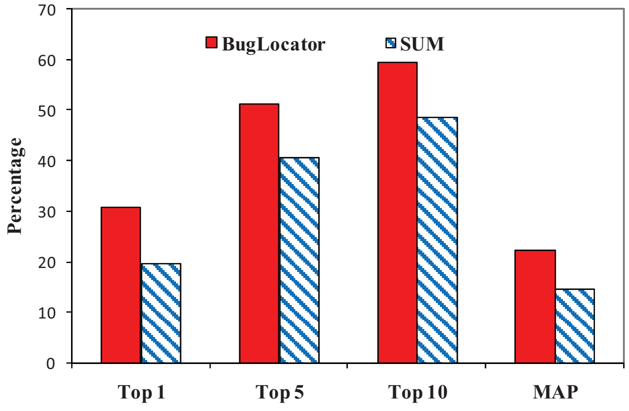


Figure 4. The comparison between the results of BugLocator and the SUM results given in [32] on AspectJ dataset

RQ2: *Does the revised Vector Space Model (rVSM) improve the bug localization performance?*

Table III shows the results of bug localization using the classic VSM and the proposed revised VSM methods. For fair comparisons, for the revised VSM we do not consider similar bugs (i.e, the weight α is set to 0). The experimental results show that the proposed VSM method outperforms the standard VSM. For example, for Eclipse 3.1, 749 bugs (24.36%) whose relevant source file are returned as top 1 using the proposed VSM method. In classic VSM, this number is only 211 (6.86%). When measuring the performance in terms of MAP and MRR, the proposed VSM method can lead to MRR value 0.35 and MAP value 0.26, which are better than the values achieved by the standard VSM (MRR=0.13 and MAP=0.09). Similar results are observed for other projects as well.

TABLE III. The Performance of Bug Localization with Classic and Revised VSM Models

System	VSM Method	Top 1	Top 5	Top 10	MRR	MAP
ZXing	Classic	4 (20%)	7 (35%)	10 (50%)	0.28	0.27
	Revised	8 (40%)	11 (55%)	14 (70%)	0.48	0.41
SWT	Classic	11 (11.22%)	32 (32.65%)	45 (45.92%)	0.23	0.20
	Revised	31 (31.63%)	64 (65.31%)	76 (77.55%)	0.47	0.40
AspectJ	Classic	36 (12.59%)	68 (23.78%)	82 (28.67%)	0.18	0.08
	Revised	65 (22.73%)	117 (40.91%)	159 (55.59%)	0.33	0.17
Eclipse	Classic	211 (6.86%)	520 (16.91%)	736 (23.93%)	0.13	0.09
	Revised	749 (24.36%)	1419 (46.15%)	1719 (55.90%)	0.35	0.26

RQ3: *Does the consideration of similar bugs improve the bug localization performance?*

Table IV below shows the experimental results of bug localization without using information from similar bugs (i.e., the weighting factor α is 0). Comparing Table II and Table IV, we can see that the information of similar bugs can indeed improve the bug localization performance. For example, for the Eclipse project, utilizing similar bugs we can locate relevant source files at top 1 for 896 bugs (29.14%), within top 10 for 1925 bugs (62.60%). The MRR and MAP values are 0.41 and 0.30, respectively. Without considering similar bugs, only 749 bugs (24.36%) have their relevant files ranked as the top 1, and 1719 bugs (55.90%) have their relevant files ranked within top 10. The MRR and MAP values are only 0.35 and 0.26, respectively. Similar results are observed for other projects as well.

TABLE IV. THE PERFORMANCE OF BUG LOCALIZATION WITHOUT USING SIMILAR BUGS

System	Top 1	Top 5	Top 10	MRR	MAP
ZXing	8 (40%)	11 (55%)	14 (70%)	0.48	0.41
SWT	31 (31.63%)	64 (65.31%)	76 (77.55%)	0.47	0.40
AspectJ	65 (22.73%)	117 (40.91%)	159 (55.59%)	0.33	0.17
Eclipse	749 (24.36%)	1419 (46.15%)	1719 (55.90%)	0.35	0.26

We also evaluate the impact of similar bug information on bug localization performance, with different α values. We find that at beginning, the bug localization performance increases when the α value increases. However, after a certain point, further increase of the α value will decrease the performance. As an example, Figure 5 below shows the bug localization performance (measured in terms of MAP and MRR) for the Eclipse project. When the α value increases from 0 to 0.3, both MAP and MRR values increases. Increasing α value further from 0.4 to 0.9 however leads to lower performance. When α is between 0.2 and 0.3, we obtain the best bug localization performance.

RQ4: *Can BugLocator outperform other bug localization methods?*

We implement bug localization methods using VSM, LDA, SUM and LSI models and perform experiments on all subject systems. We then compare the performance of BugLocator with the related methods. Figure 6 shows the percentage of bugs that can be located in top 1 and top 10 returned files. Clearly, BugLocator outperforms all other methods. For example, using BugLocator we can locate 29.14% Eclipse bugs in the first returned (top 1) files, while using VSM, LDA, SUM and LSI models, we can only locate 6.86%, 0.32%,1.72% and 4.23% Eclipse bugs in the first returned files, respectively. BugLocator also outperforms other models when the performance is measured in terms of MAP and MRR. For example, for ZXing, the MAP and MRR values are 0.44 and 0.50 respectively, which are much higher than the second best model (i.e., SUM), whose MAP and MRR values are 0.30 and 0.37, respectively. Detailed results are omitted due to space constraints. The t-tests at 95% confidence level confirm that our method statistical significantly outperforms the others.

B. Discussions of the Results

1) Why does the proposed rVSM method work?

Our experimental results described in the previous section show that the proposed rVSM performs better than the classical VSM when used for bug localization. In this section, we discuss why the proposed rVSM can achieve better performance.

The differences between rVSM and VSM are in the Equations (4) and (5). Equation (4) uses the logarithm of the original *tf* value. This is because terms with high frequency may have negative impact on information retrieval performance. It is often not the case that the term importance is proportional to its occurrence frequency. The logarithm variant of *tf* can help smooth the impact of the high frequent terms [8, 13].

Equation (5) adjusts the ranking results based on file sizes. This is based on the findings of our earlier study [37] that the larger files tend to be more defect-prone than the smaller files.

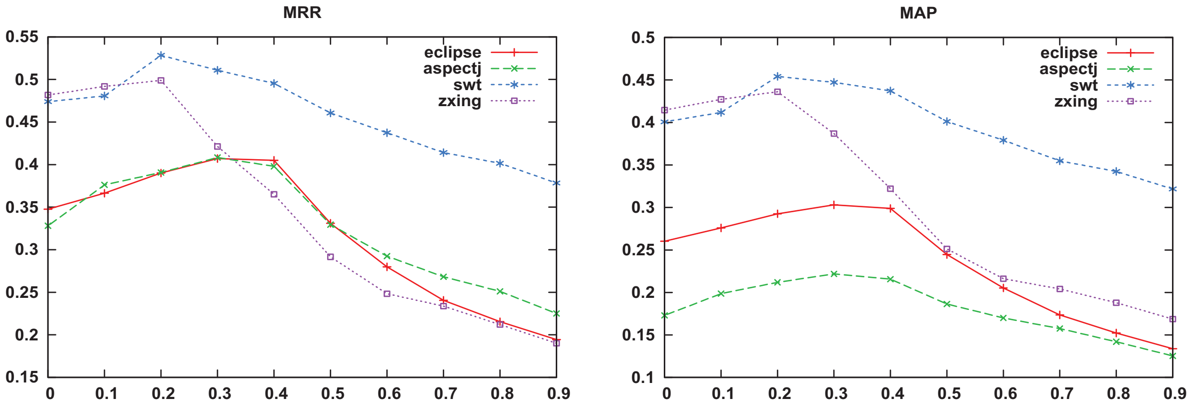


Figure 5. The impact of α on bug localization performance (MAP and MRR)

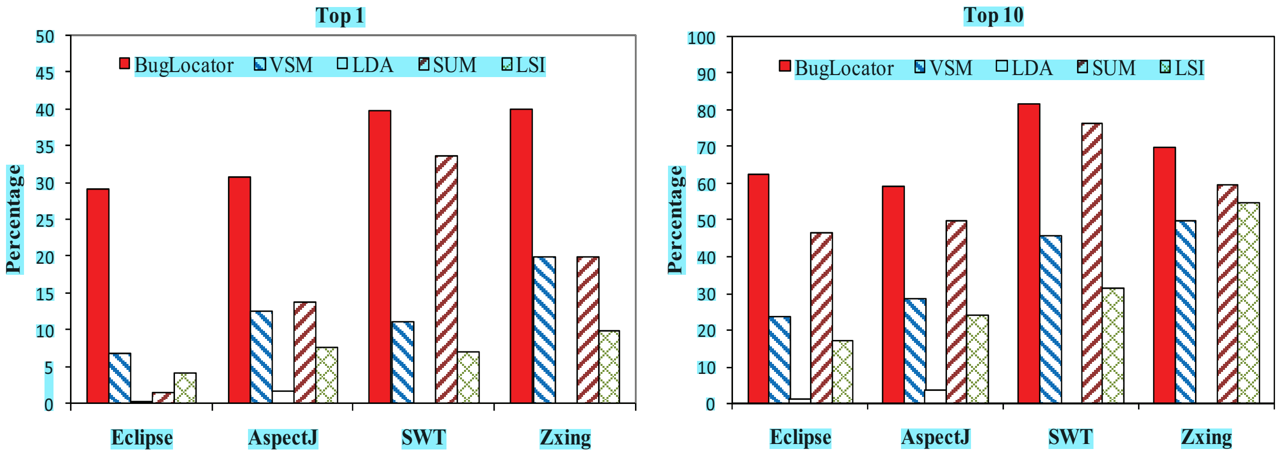


Figure 6. The comparisons between different bug localization methods

TABLE V. THE COMPARISONS OF DIFFERENT LENGTH FUNCTIONS

Length function *g*	Expression	MAP	MRR
Logistic	$f(x)=\frac{1}{1+e^{-x}}$	0.26	0.35
Exponential	$f(x)=e^{x}-0.5$	0.25	0.33
Square root	$f(x)=1-\frac{1}{\sqrt{x}}$	0.19	0.26
Linear	$f(x)=\frac{x}{2}+0.5$	0.25	0.34

In [37], we found that a small number of largest files account for a large proportion of the defects. For example, in Eclipse 3.0, 20% of the largest files are responsible for 62.29% pre-release defects and 60.62% post-release defects. Similar phenomenon is also observed by many others include Ostrand et al. [29]. They studied the “ranking ability” of LOC for two industrial systems and found that 20% of largest files contain 73% and 74% of the bugs for the two systems. In summary, the empirical studies confirm that by ranking larger files higher we can locate more bugs.

Equation (5) uses a logistic function *g* to adjust the ranking results. We also experiment with other length functions including linear, square root and exponential functions (Table V). These functions weight files of different sizes differently. Our experiment results (Table V) show that the logistic function achieves the best overall MAP and MRR values, outperforming other length functions.

2) Why can similar bugs help improve bug localization performance?

We also explore why similar bugs can improve the bug localization. We find out that for many bugs, the associated files have overlaps with the associated files of their similar bugs. For example, in Eclipse, 1207 (39.3%) bugs have at least one relevant file that is common to the files of their top 10 most similar bugs. For 602 (19.6%) bugs, all their relevant files are covered by their top 10 most similar bugs. These results suggest that similar bugs can improve the bug localization performance.

The analysis of similar bugs becomes more important when the textual similarity between bug reports and source code is low. As an example, for the Eclipse bug 89014 that is reported on March 24, 2005, it was fixed in the file *BindingComparator.java*. Using rVSM, the relevant file *BindingComparator.java* is only ranked 2527, because the textual similarity between source code and the bug report is low. However, the analysis on similar bugs found that this bug is actually similar to previous fixed bugs 83817, 79609 and 79544, which all introduced bug-fixing changes to the file *BindingComparator.java*. Therefore, BugLocator combines the scores obtained from rSVM and similar bug analysis based on Equation (9), and the final rank of the file *BindingComparator.java* becomes 7.

3) The percentage of code to be examined for bug localization

Our experimental results reported in the previous sections only evaluate the performance of bug localization in terms of the number of relevant files retrieved. In practices, developers are also interested in the actual lines of code need to be examined in order to locate a bug. This is of particular concern as the proposed rVSM model ranks larger source files higher via the length function defined in Equation (5).

We perform further experiments to evaluate how many lines of code are required to be examined in order to locate the bugs. For each bug, we count the number of files to be examined before locating the bug, and compute the lines of code for each file. The results show that BugLocator is still effective when its performance is measured in terms of lines of code to be examined. For example, by examining 1% lines of code, BugLocator can locate nearly 80% bugs in Eclipse and 60% bugs in AspectJ.

BugLocator can also locate more bugs than SUM when the same number of lines of code is examined. For Eclipse, using BugLocator we can locate more than 95% bugs by examining 10% of code, while using SUM (the best performing method described in [32]) we can only locate about 81% bugs by examining the same amount of code. For the other systems, we obtain similar results.

VI. Threats to Validity

There are potential threats to the validity of our work:

- All datasets used in our experiments are collected from open source projects. The nature of the data in open source projects may be different from those in projects developed by well-managed software organizations. We need to evaluate if our solution can be directly applied to commercial projects. We leave this as a future work.
- A limitation of our approach is that we rely on good programming practices in naming variables, methods and classes. If a developer uses non-meaningful names the performance of bug localization would be affected. However, in our experiments we notice that in most well-managed projects, developers generally follow good naming conventions.
- Bug reports provide crucial information for developers to fix the bugs. A "bad" bug report could cause a delay in bug fixing. Our approach also relies on the quality of bug reports. If a bug report does not provide enough information, or provides misleading information, the performance of BugLocator is adversely affected.

VII. Related Work

Bug fixing is an important but still costly activity in software development. Spectrum-based fault localization techniques [1, 18, 19, 22, 23] can help developers locate faults by examining a small portion of code. These techniques usually contrast the program spectra information (such as execution statistics) between passed and failed executions to compute the fault suspiciousness of individual program elements (such as statements, branches, and predicates), and rank these program elements by their fault suspiciousness. Developers may then locate faults by examining a list of program elements sorted by their suspiciousness. Examples of spectrum-based fault localization techniques include Tarantula [18, 19], Jaccard and Ochiai [1]. The spectrum-based fault localization techniques require program runtime execution traces. Our approach is based on the query of bug reports against the source code repository, which does not require the collection of the passing and failing execution traces. There are also other techniques that help developers automatically locate bugs, such as delta debugging [39] and dynamic slicing [38]. Unlike these techniques, our approach is a static approach, which does not require the execution of the programs.

In recent years, many information retrieval based bug localization methods have been proposed [25, 28, 32]. As described in the previous sections, BugLocator performs better than the related methods because of the utilization of rVSM and similar bug information. This area of work is also closely related to feature/concept location [2, 3, 24, 28], which is about identifying the parts of the source code that correspond to a specific functionality. The results can be used as starting points in change impact analysis. The problem of locating bug-related code could be also treated as a feature/concept location problem. Poshyvanyk et al. [31] presented a feature location method called PROMESIR, which combines results from both dynamic analysis and information retrieval. They applied PROMESIR to locate 8 bugs in Mozilla and Eclipse systems. Gay et al. [16] also proposed a concept location approach that augments information retrieval based concept location via an explicit relevance feedback mechanism. They evaluated their approach using 7 Eclipse bug reports. Our approach is dedicated to bug localization. We perform large-scale evaluations using more than 3000 bug reports from four different systems. Unlike the work described in [31] and [16], we do not require program execution or user interaction.

Our work is also related to research on mining software repository. The existence of large amount of data stored in bug tracking systems provides many opportunities for automated software quality analysis and improvement. Many researchers mine bug report information to solve software engineering problems such as duplicate bug detection [33, 35], automatic bug triage [4, 7], bug report quality analysis [6, 7], and defect prediction [20, 37]. Because of the large number of bugs, such problems cannot be effectively solved by manual efforts. In our approach, we utilize bug report information to automatically locate buggy files.

VIII. Conclusions

Once a new bug report comes, developers need to know which files should be modified to fix the bug. For a large software project, they may need to examine a large number of source code files in order to locate the bug, which could be a tedious and costly work. In this paper, we have proposed an IR-based method named BugLocator for locating relevant source code files based on initial bug reports. BugLocator utilizes a revised Vector Space Model (rVSM) as well as similar bug information. The evaluation results on four real-world open source projects show that BugLocator can perform bug localization effectively. The results also show that BugLocator outperforms existing methods such as those based on VSM, LDA, LSI, and SUM.

In future, we will explore if program execution information can be integrated into our approach to help further improve bug localization performance. We will also apply BugLocator to industrial projects to evaluate its effectiveness in practice.

Our tool and the experimental data are available at:

http://code.google.com/p/bugcenter

Acknowledgment

This work is supported by NSFC grant 61073006 and Tsinghua University project 2010THZ0. We thank Rongxin Wu and Aihui Zhou for helping with data collection.

References

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

[2] G. Antoniol, G. Canfora, G. Casazza, and A. Lucia, Identifying the Starting Impact Set of a Maintenance Request: A Case Study, *Proc.*

Fourth European Conf. Software Maintenance and Reeng. (CSMR '00), Zurich, Switzerland, p. 227-231, March 2000.

[3] G. Antoniol and Y. Gue´he´neuc, Feature Identification: A Novel Approach and a Case Study, *Proc. 21st IEEE Int'l Conf. Software Maintenance (ICSM '05)*, Budapest, Hungary, p.357-366, Sept 2005.

[4] J. Anvik, L. Hiew, and G. C. Murphy. Who should fix this bug? In *ICSE '06: Proceedings of the 28th international conference on Software engineering*, p. 361–370, Shanghai, China, May 2006.

[5] A. Bachmann and A. Bernstein. Software process data quality and characteristics: a historical view on open and closed source projects. *IWPSE-Evol '09 Proceedings of the joint international and annual ERCIM workshops on Principles of software evolution (IWPSE) and software evolution (Evol) workshops,* ACM, 2009.

[6] 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*, Atlanta, GA , November 2008.

[7] N. Bettenburg, R. Premraj, T. Zimmermann, and S. Kim., Duplicate bug reports considered harmful... really? In *Proceedings of the 24th IEEE International Conference on Software Maintenance*, Beijing, China, September 2008.

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

[9] V. Dallmeier, T. Zimmermann. Extraction of Bug Localization Benchmarks from History. In *Proceedings of the 22nd IEEE/ACM International Conference on Automated Software Engineering (ASE 2007)*, Atlanta, Georgia, USA, November 2007.

[10] V. Dallmeier, T. Zimmermann. Automatic Extraction of Bug Localization Benchmarks from History. *Technical Report,* Saarland University, June 2007.

[11] M. B. David, Y. Ng. Andrew, M. I. Jordan. Latent Dirichlet Allocation*, Journal of Machine Learning Research*, vol. 3, p. 993-1022, 2003.

[12] S. Deerwester,et al, Improving Information Retrieval with Latent Semantic Indexing, *Proceedings of the 51st Annual Meeting of the American Society for Information Science 25*, p.36–40, 1988.

[13] S. T. Dumais,Improving the retrieval of information from external sources, *Behavior Research Methods, Instruments, and Computers*, Psychonomic Society, p.229 - 236, 1991.

[14] N. Fenton and N. Ohlsson, *Quantitative Analysis of* Faults and Failures in a Complex Software System, *IEEE Trans. Software Eng.*, 26 (8), pp. 797-814, 2000.

[15] E. Garcia, Description, Advantages and Limitations of the Classic Vector Space Model, Oct 2006, available at: http://www.miislita.com/term-vector/term-vector-3.html

[16] G. Gay, S. Haiduc, A. Marcus and T. Menzies, On the use of relevance feedback in IR-based concept location, *Proc. the 25th IEEE International Conference on Software Maintenance*, Edmonton, Alberta, Canada, p.351-360, September 2009.

[17] I. J. Good, The population frequencies of species and the estimation of population parameters. *Biometrika,*40(3 and 4) , p.237-264, 1953

[18] J. A. Jones and M. J. Harrold. Empirical evaluation of the Tarantula automatic fault-localization technique. In *Proceedings of the 20th IEEE/ACM International Conference on Automated Software Engineering (ASE 2005)*, Long Beach, California, p. 273-282, 2005.

[19] J. A. Jones, M. J. Harrold, J. Stasko. Visualization of test information to assist fault localization. In *Proceedings of the 24th International Conference on Software Engineering (ICSE 2002)*, Orlando, Florida, USA, ACM Press, p. 467-477, May 2002.

[20] S. Kim, T. Zimmermann, E. Whitehead Jr., A. Zeller, Predicting Faults from Cached History, *Proc. ICSE'07,* Minneapolis, USA, May 2007.

[21] S. Kullback, K. P. Burnham, N. F. Laubscher, G. E. Dallal, L. Wilkinson, D. F. Morrison, M. W. Loyer, B. Eisenberg, et al. Letter to the Editor: The Kullback–Leibler distance. *The American Statistician* 41 (4), p. 340–341, 1987.

[22] B. Liblit, M. Naik, A. X. Zheng, A. Aiken, M. Jordan. Scalable statistical bug isolation. In *Proceedings of the 2005 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI 2005),* Chicago, IL, USA, p.15-26, June 2005.

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

[24] D. Liu, A. Marcus, D. Poshyvanyk, V. Rajlich, Feature Location via Information Retrieval based Filtering of a Single Scenario Execution Trace, *in Proceedings of 22nd IEEE/ACM International Conference on Automated Software Engineering (ASE 2007)*, Atlanta, Georgia, November 5-9, p. 234-243.

[25] S. Lukins, N. Kraft and L. Etzkorn. Bug localization using latent Dirichlet allocation. *Information and Software Technology* ,Volume 52, Issue 9, p. 972-990, September 2010.

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

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

[28] A.Marcus, A. Sergeyev, V.Rajlich, J. Maletic, An Information Retrieval Approach to Concept Location in Source Code. In *Proceedings of the 11th IEEE Working Conference on Reverse Engineering (WCRE 2004)*, Delft, The Netherlands, p. 214-223, November 9-12, 2004.

[29] T. Ostrand, E. Weyuker and R. Bell, Predicting the Location and Number of Faults in Large Software Systems, *IEEE Trans. Software Eng.*, 31 (4), pp. 340-355, 2005.

[30] D. Poshyvanyk, Y.-G. Gueheneuc, A. .Marcus, G. Antoniol, V.Rajlich, Combining Probabilistic Ranking and Latent Semantic Indexing for Feature Identification, *Proceedings of the 14th IEEE International Conference on Program Comprehension (ICPC 2006)*, Athens, Greece, p.137-146, June 2006.

[31] D. Poshyvanyk, Y. Guéhéneuc, A. Marcus, G. Antoniol and V. Rajlich, Feature Location using Probabilistic Ranking of Methods based on Execution Scenarios and Information Retrieval, *IEEE Transactions on Software Engineering,* p. 420-432, 33(6), 2007.

[32] S. Rao and A. Kak. Retrieval from software libraries for bug localization: a comparative study of generic and composite text models. In *Proceeding of the 8th working conference on Mining software repositories (MSR'11)*, ACM, Waikiki, Honolulu, Hawaii, p.43-52, May 2011.

[33] C. Sun, D. Lo, X. Wang, J. Jiang, and S.C. Khoo. A discriminative model approach for accurate duplicate bug report retrieval, *Proc. of the 32nd ACM/IEEE International Conference on Software Engineering (ICSE'10)*. Cape Town, South Africa, May 2010.

[34] E. M. Voorhees, *TREC-8* Question Answering Track Report, *Proceedings of the 8th Text Retrieval Conference*, p. 77–82, 1999.

[35] 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 *Proceedings of the 30th International Conference on Software Engineering (ICSE'08)*, Leipzig, Germany, May 2008.

[36] I. H. Witten and T. C. Bell, The Zero-frequency Problem: Estimating the Probabilities of Novel Events in Adaptive Text Compression, *IEEE Transcations on information Theory*, 37(4), p.1085-1094,1991.

[37] H. Zhang, An Investigation of the Relationships between Lines of Code and Defects, *Proc. the 25th IEEE International Conference on Software Maintenanc (ICSM'09)*, Edmonton, Canada, p. 274-283, September 2009.

[38] X. Zhang, H. He, N. Gupta, and R. Gupta. Experimental evaluation of using dynamic slices for fault location. In *Automated and Algorithmic Debugging (AADEBUG)*, Monterey, California, USA , p. 33–42, 2005.

[39] A. Zeller, R. Hildebrandt, Simplifying and isolating failure-inducing input, *IEEE Transactions on Software Engineering* 28 (2), p. 183–200, 2002.