Improved Bug Localization using Association Map and Information Retrieval

Abstract—**Bug localization is one of the most challending tasks undertaken by the developers during software maintenance. Existing studies mostly rely on lexical similarity between the bug reports and source code for bug localization. However, such similarity always does not exist, and these studies suffer from vocabulary mismatch issues. In this paper, we propose a bug localization technique that (1) not only uses lexical similarity between bug report and source code documents but also (2) exploits the co-occurrences between keywords from the past reports and source files from corresponding changed code. Experiments using a collection of 3000 bug reports show that our technique can locate buggy files with a Top-10 accuracy of 64.05% and a mean reciprocal rank@10 of 0.31 and a mean precision average@10 of 37%, which are highly promising. ~~Comparison with state-of-the-art IR-based bug localization techniques confirms superiority of our technique.~~**

Index Terms—**bug localization, bug report, source codes, association map**

I. Introduction

Bug localization is the process of locating the source codes that need to be changed in order to fix a given bug. Locating buggy files is time-consuming and costly ~~if it is done by manual effort,~~ especially for the large software system ~~when the number of bug of that system becomes large~~. Therefore, effective methods for locating bugs automatically from bug reports are highly desirable. In automatic bug localization technique~~, it~~ takes a subject system as an input and produces a list of entities such as classes, methods etc. against a search query. ~~For example,~~ information retrieval based techniques rank the list of entities by predicted relevance and return a ranked list of entities or source codes which may contain the bug. The bug localization techniques are also affected by the fact of designing effective query. If a query contains inadequate information, then the retrieval results will not be relevant at all. ~~Moreover the performance of existing bug localization approach did not reach to an accepted level and so far those studies showed good results for a small set of bugs~~. Therefore, in this paper, we apply a bug localization technique ~~on large dataset~~ that exploits an association link established from bug report repository to the corresponding source code base through commit logs.

In existing studies, information retrieval techniques [26] [17] [11] [12] [8] are applied to automatically search for relevant files based on bug reports. In one case of Text Retrieval (TR) based approaches, Zhou et al. [26] propose BugLocator using revised Vector Space Model (rVSM), which requires the information extracted from bug reports and source codes. One of the issue association with TR-based technique is treating source codes as flat text that lacks structure. But exploiting source code structure can be a useful way to improve bug localization accuracy. 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). However, large files are more likely to contain noise as usually only a small part of a large file is relevant to the bug report. So, by treating files as single units or favoring large files, the techniques are more likely to be affected by the noise in large files. So, Saha et al. [17] present BLUiR, which retrieve structural information from code constructs. However, bug reports often contain stack-trace information, which may provide direct clues for possible faulty files. Most existing approaches directly treat the bug descriptions as plain texts and do not explicitly consider stack-trace information. Here, Moreno et al. [11] combine both stack trace similarity and textual similarity to retrieve potential code elements. To deal with two issues associated with source code structure and stack trace information, Wong et al. [23] proposed a technique that use segmentation i.e., divides each source code file into segments and use stack-trace analysis, which significantly improve the performance of BugLocator.

LDA topic-based approaches [12] [8] assume that the textual contents of a bug report and it's corresponding buggy source files share some technical aspects of the system. Therefore, they develop topic model that represents those technical aspects as topic. However, existing bug localization approaches applied on small-scale data for evaluation so far. 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]. But, now it is an important research question to know how effective these approaches are for locating bugs in large-scale (i.e., big data). In the field of query processing Sisman and Kak [19] proposed a method that examines the files retrieved for the initial query supplied by a user and then selects from these files only those additional terms that are in close proximity to the terms in the initial query. Their [19] experimental evaluation on two large software projects using more than 4,000 queries showed that the proposed approach leads to significant improvements for bug localization and outperforms the well-known QR methods in the literature. There are two common issues associated with existing studies. First, most of the bug localization techniques exploit only lexical similarity measure for retrieving relevant files from the code base. So, it is expected that the search query constructed from new bug report should contain keywords similar to code constructs in code base. This issue demands developers or users previous expertise on the given project, which can not be guaranteed in real world. Second, closely related to the first issue there is another well known problem called vocabulary mismatch. In order to convey the same concept on both search query (i.e., new bug report) and source code files the developers tend to use different vocabularies. This issue questions the applicability of exploiting lexical similarity measure.

So, in order to resolve these two issues, in our work we propose a bug localization approach that combines lexical similaority with word co-occurrence measue. Our proposed technique exploits the association of keywords extracted from fixed bug report with their changed source code location. Here, the main idea is to capture information from a collection of bug reports and exploit them for locating relevant source code location. So our approach not (1) only relies on the lexical similarity measure between big reports and source code files for bug localization, and (2) but also addresses the vocabulary mismatch problem by using a keyword-source map constructed from fixed bug information. However, we named our proposed tool as BLuAMIR (Bug Localization using Association Map and Information Retrieval).

We compare the performance of our proposed tool with a state of the art bug localization techniques . So, contributions of our paper include:

- A novel bug localization technique that exploits the association relationship between bug report keywords with their buggy source files.
- Comprehensive evaluation of the technique using about 3000 bugs from Eclipse bug repository and validation against state-of-the-art.

II. An Example Use Case Scenario

Consider a new bug (ID 37026) is submitted to the Eclipse-UI-Platform bug repository system. The title of this bug is *[Working Sets] IWorkingSet.getImage should be getImageDescriptor*. We preprocess this title as well as the description of this new bug to form a query. Now we apply both VSM and our proposed techniques on this query in order to locate possible buggy files. The recommended ranked results are presented in Table I. From version repository, we have found that so far 14 number of files have been modified in order to fix this bug. From Table I, we can see that our proposed tool BLuAMIR correctly identifies 5 buggy files where VSM approach locates only 2 buggy files. The ranking of BLuAMIR is 1,2,3,6,7 where the rank from VSM is 1 and 4. Note that BLuAMIR retrieves total 5 buggy files including the 2 located by VSM technique. However, VSM and Co-Occ scores are also provided for each recommended buggy file. Note that Same file is retrieved in rank 1 for both techniques. The buggy file which is retrieved in rank 4 by VSM, retrieved as rank 3 by BLuAMIR. The other 3 buggy files recommended by BLuAMIR in Top-10 having rank of 12, 27 and 24 in the VSM approach retrieval.

We try to reason why BLuAMIR can locate more buggy files than VSM. In BLuAMIR, the Co-Occ score comes from the association map, which is created from keywords extracted from previously fixed bug report into their fixed buggy files. In VSM, we compare term presented in the bug reports and source files. But we know, there exist a vocabulary missmatch problem between bug reports and source corpus. But, in computing Co-Occ rank we dont directly compare terms between bug reports and source files. So vocabulary missmatch problem is resolved here. Moreover, in the working example, this association map aids locating more buggy files than VSM technique. IF we go deeper, we can find that due to vocabulary missmatch problem, VSM failed to retrieve buggy files where our proposed tool utilize an association map that successfully retrieve previous relationship between bug report keywords and their source buggy files.

III. Research Question Formulation

Our proposed tool-BLuAMIR are designed to answer the following research questions.

- RQ1: How many bugs can be successfully located by BLuAMIR?
- RQ2: Does our proposed approach-BLuAMIR resolve the vocabulary missmatch problem?
- RQ3: How does BLuAMIR eliminate vocabulary missmatch problem?
- RQ4: In BLuAMIR comparable with the state-of-the-art techniques in identifying buggy files?

IV. Proposed System Diagram

Our proposed approach combine lexical similarity and co-occurence similarity measure. Zhou et al. [26] proposed BugLocator based on two different similarity scores- one is rVSM score and the other one is Simi score. In BLuAMIR, we follow two different approach for computing source code ranks. In Approach I we retrieve relevant ranked source files based on two different scores - i) VSM and ii) Word Co-occurence and in Approach II based on three different scores- i) rVSM, ii) Simi and iii) Word Co-occurence. However, we have divided our approach into two different sections or parts- 1) calculate rVSM and Simi scores or VSM score and co-occurence measure and 2) combine all three or two scores in order to localize recommended buggy source files for a given newly reported bug. As we have combined two existing scores with our proposed word co-occirence score, we will discuss the system diagram of association map database in Part I and then we represent rest of our system diagram in Part II.

A. Part I: Mapping Bug Source Code Links

In Part I, we construct an association map database - between keywords extracted from previously fixed bug reports collection and source code links. The system diagram for Part I is given in Fig. 1(a).

This mapping construction involves two steps. At first we create association map using information contained in bug reports collection and commit logs. We collect information

TABLE I
A WORKING EXAMPLE OF BLUAMIR

Query #ID	VSM score	Retrieved Files (VSM Approach)	GoldSet	VSM score	Co-Occ-Score	ScoreAll	Retrieved Files (BLuAMIR)	Goldset
#37026	1.00	WorkingSetLabelProvider.java	✓	0.60	0.25	0.85	WorkingSetLabelProvider.java	✓
	0.80	ImageFactory.java	✗	0.35	0.35	0.70	WorkingSet.java	✓
	0.76	WorkingSetMenuContributionItem.java	✗	0.45	0.22	0.67	IWorkingSet.java	✓
	0.75	IWorkingSet.java	✓	0.38	0.25	0.63	WorkingSetTypePage.java	✗
	0.70	ProjectImageRegistry.java	✗	0.30	0.33	0.63	CommandImageService.java	✗
	0.68	IWorkingSetManagerTest.java	✗	0.31	0.29	0.60	WorkbenchImages.java	✓
	0.67	IWorkbenchPartDescriptor.java	✗	0.33	0.27	0.60	WorkingSetAdapterFactory.java	✓
	0.65	MockWorkingSetPage.java	✗	0.30	0.27	0.57	EditorIconTest.java	✗
	0.65	MissingImageDescriptor.java	✗	0.30	0.27	0.57	PerspectiveDescriptor.java	✗
	0.64	WorkingSetTypePage.java	✗	0.24	0.33	0.57	IAction.java	✗

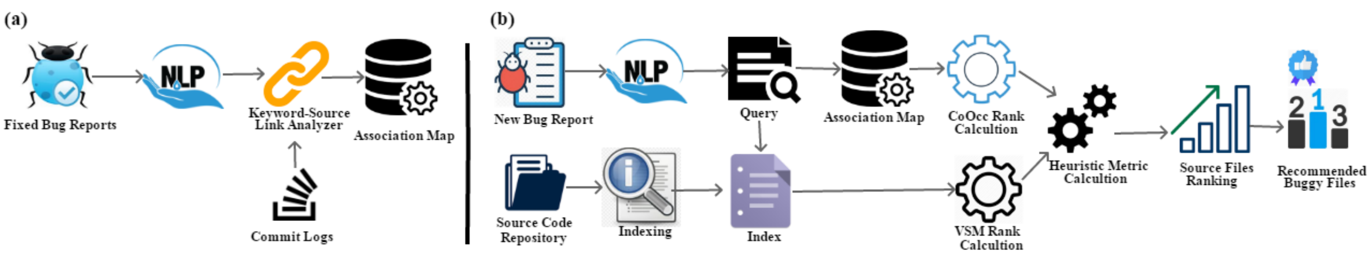

Fig. 1. Proposed System Diagram: (a) Mapping Bug Source Code Links and (b) Bug Localization Using VSM, and Co-occurence Ranks

from title and description fields of each bug report. There are also developers comment section in a bug report, which is highly informative. But large documents also tend to reduce performance by taking too long to process. So, we reluctant to include developers comments of each bug report. However, we extract keywords from each bug report after preprocessing them such as stop word removal. In version repository, we have commit logs, where the developers commit for several changes in a software project. For example, when a bug report is fixed, the developer who fixed this bug also creates a commit log containing which type of change he had to made to resolve the bug associated with the location of the source code files where the change has made. So, if we analyze commit logs, we can retrieve source code files information which have been changed in order to fix a given bug. Here the idea is, we first extract keywords from bug report and corresponding buggy source code files information for the same bug report from commit log. Then we construct an association map database containing this connected relationship information. The links between keywords and source code links can be described as: each or several keywords can be linked to one or more source code files and each source code file can be linked to one or several keywords. This way we connect keywords and source code files to form an association map database named keyword-source code links.

B. Part II: Bug Localization Using either VSM or rVSM and Simi, and Co-occurence Ranks

The system diagram for this part has illustrated in Fig. 1(b). In BLuAMIR, we calculate similarity score in two ways - one is based on the combination of VSM and Co-occurence scores (i.e., Approach I) and other one is rVSM, Simi and Co-occurence scores (i.e., Approach II). However, for simplicity we represent the former approach (i.e., Approach I) in our proposed schematic diagram in Fig. 1(b). In this approach, we calculate VSM and Co-occurence ranks in order to recommend buggy source codes for a given newly reported bug. On the other hand, in our second approach, we work with three different ranks i.e., rVSM, Simi and Co-occurence. We replicate an existing technique proposed by Zhou et al. [26] for computing rVSM and Similarity scores. Basically, both VSM and rVSM are TF-IDF based score, which is measured between query and source code files. On the other hand, Simi score refers to that fact that if a bug is similar to another bug, then they both tend to relate to same sources. However, we describe both scores in Section VI. We have constructed an association map databases from keywords collected from bug report into its corresponding source code information in Section V. For the candidate keyword tokens for the initial query, we exploit association map database (i.e., keyword-source code links) and retrieve the relevant source code files. We use some heuristic functions in order to combine three ranks and recommend buggy relevant source files.

V. PART I: ASSOCIATION MAP DATABASE CONSTRUCTION

In this part, we construct an association map databases - between keywords and corresponding source code links extracted from bug reports and commit messages respectively. This section can be further divided into several parts: keyword extraction from bug reports, source code information extraction from commit logs, keyword- source code linking. We also create an algorithm for creating association map database between bug report keywords into their buggy source files, which is given in Algorithm 1. However, several steps of map database construction are discussed in the followings:

Algorithm 1 Construction of Association Map Database Between Bug Reports and Source Files

1: **procedure** BUG REPORTS (BRC) ▷ BRC: a collection of bug reports
2: $MAP_{KS} \leftarrow \{\}$ ▷ an association map
3: $MAP_{kb} \leftarrow$ createKeywordsToBugMaps(MAP_{adj}) ▷ collecting unique keywords from keywords to bug map
4: $KB \leftarrow$ collectKeywords(MAP_{kb})
▷ Linking keywords from a bug report into its change set files
5: **for** keywords $KB_i \in KB$ **do**
6: $BUG_{id} \leftarrow$ retrieveBugIds (K_i)
7: **for** each bug id $BUG_{id_j} \in BUG_{id}$ **do**
8: $SF_{links} \leftarrow$ getLinkedSourceFiles(BUG_{id}) ▷ maps all source code files to its keywords
9: $MAP[KB_i].links \leftarrow MAP[KB_i].links + SF_{links}$
10: **end for**
11: **end for**
▷ collecting all keyword-source files links
12: $MAP_{KS} \leftarrow MAP[KB]$
13: **return** MAP_{KS}
14: **end procedure**

Keyword Extraction from Bug Reports: We collect title, description and developers comments from each fixed bug report in a collection of bug reports. We perform standard natural language pre-processing on the corpus such as stop word removal, splitting and stemming. The purpose of removing stop words is that they contain very little semantic for a sentence. The process of stemming step extracts the root of each of the word. We use this stop word list [1] during stop word removal and Porter Stemming stemmer [2] for stemming. Line 3 in Algorithm 1 describe this step.

Source Code Link Extraction from Commit Logs: We go through all commit messages and try to find those commit message that contain keywords related to bug fix such as fix, fixed and so on. Each of these commit messages presents other information such as the ID of bug report for which it was created and the links of the corresponding changed source code files. We then construct a linking relationship between each fixed bug report ID into their changed source code files, which is mentioned in line 4 of Algorithm 1.

Keyword-Source Code Linking: At this point, in one side we have pre-processed keywords associated with each bug report and on the other side we have a relationship information between bug report ID and buggy source code links. We construct a bipartite graph between keywords collected from a bug report to its buggy source code locations. Here, one or more keywords can be linked to one or more buggy source code files links and a source code file can be linked to one or more keywords, which is constructed from line 5 to line 11 in Algorithm 1.

[1] 1https://www.ranks.nl/stopwords

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

VI. Part II: Localizing Buggy Source Code Files

In this part, we combine existing two lexical similarity based bug localization approaches with our proposed keyword-source co-occurence relation. One is with VSM score and other one is with rVSM and Simi ranks which are proposed in BugLocator [26]. So, there are two sub parts of this section.

A. Approach I: Combinition of VSM and Co-occurence Rank:

For locating a new bug we compute similarity scores for all source code files for a given project. However, we need to focus on some concepts which are required to understand our proposed system. They are described as follows:

VSM Score Calculation: In this technique, each source code file is ranked based on source code file scores. Source code file contains words those can be also occurred in the bug reports. This is considered as a hint to locate buggy files. The basic idea of a VSM (Vector Space Model) or TF-IDF model 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 [26]. In our proposed approach we have used both classical Vector Space Model (VSM) and revised Vector Space Model (rVSM) proposed by Zhou et al. [26] in order to index and rank source code files. In classic VSM, *tf* and *idf* are defined as follows:

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

Here *tf* is the term frequency of each unique term t in a document d and f_{td} is the number of times term t appears in document d. So the equation of classical VSM model is as follows

$$VSMScore(q,d) = cos(q,d) = \frac{1}{\sqrt{\sum_{t \epsilon q}((\frac{f_{tq}}{\#terms}) \times log(\frac{\#docs}{n_t}))^2}} \times \frac{1}{\sqrt{\sum_{t \epsilon d}((\frac{f_{td}}{\#terms}) \times log(\frac{\#docs}{n_t}))^2}} \times \sum_{t \epsilon q \bigcap d} (\frac{f_{tq}}{\#terms}) \times (\frac{f_{td}}{\#terms}) \times log(\frac{\#docs}{n_t})^2 \qquad (2)$$

This *VSM* score is calculated for each query bug report q against every document d in the corpus. However, in the above equation *#terms* refers to the total number of terms in a corpus, n_t is the number of documents where term t occurs.

Co-occurence Scores Calculation A query typically contains several keywords or words. For each keyword, we look for relevant source files in the keyword-source files association map. We assume these files are relevant because we created the map between the content of bug reports and their buggy source files for previously fixed bug reports. When we analysis these links for all keywords in a query, a relevant file can be found from the association relationship more than once.

Therefore, we then normalize the frequency of source files using standard TFIDF normalization technique. Finally we recommend first Top-K files with their CoOccScores. The equation for computing co-occurence score is given belows:

$$CoOccScore = \sum_{AllS_i thatconnecttoW_j} (Link(Wj, S_i)) \quad (3)$$

Here, the link $Link(Wj, S_i)$ between keyword and source file is 1 if they are connected in the association map and 0 otherwise.

Approach I: Final Score Calculation We compute VSM score using Appache Lucene library. Then we combine that score with our CoOccScore using equation 4.

$$FinalScoreApproach1 = (1 - \alpha) \times N(VSMScore) + \alpha \times N(CoOccScore) \quad (4)$$

Here, the weighting factor α varies from 0.1 to 0.5, for which we discuss results in the experiment section.

B. Approach II: Combination of rVSM, Simi Rank and Co-occurence Rank:

Ranking based on Revised Vector Space Mode: The main difference between classic VSM and revised VSM is that in case of revised version logarithm variant is used in computing term frequency. The equation [26] for calculating term frequency is:

$$tf(t, d) = log(f_{td}) + 1 \quad (5)$$

So the new equation of revised VSM model is as follows:

$$rVSMScore(q, d) = g(\#term) \times cos(q, d)$$
$$\frac{1}{1 + e^{-N(\#terms))}} \times \frac{1}{\sqrt{\sum_{t\epsilon q}((logf_{tq} + 1) \times log(\frac{\#docs}{n_t}))^2}} \times \frac{1}{\sqrt{\sum_{t\epsilon d}((logf_{td} + 1) \times log(\frac{\#docs}{n_t}))^2}} \times \sum_{t\epsilon q \bigcap d} (logf_{tq} + 1) \times (logf_{td} + 1) \times log(\frac{\#docs}{n_t})^2 \quad (6)$$

This *rVSM* score is calculated for each query bug report q against every document d in the corpus. However, in the above equation *#terms* refers to the total number of terms in a corpus, n_t is the number of documents where term t occurs.

Ranking based on similar bug information The assumption of this ranking is similar bugs of a given bug tend to modify similar source code files. Here, we construct a 3-layer architecture as described in [26]. In the top layer (layer 1) there is a bug B which represents a newly reported bug. All previously fixed bug reports which have non-negative similarity with bug B are presented in second layer. In third layer all source code files are shown. In order to resolve each bug in second layer some files in the corpus were modified or changed, which are indicated by a link between layer 2 and layer 3. Foe all source code files in layer 3, similarity score

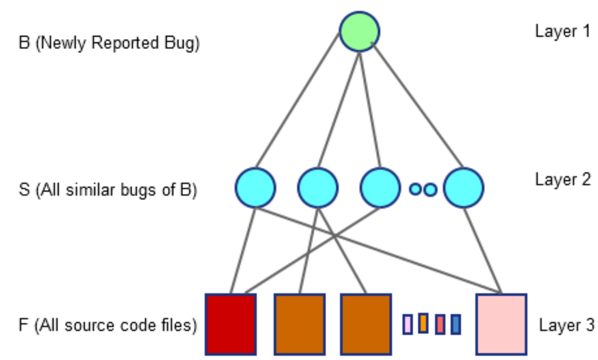

Fig. 2. Bug and its Similar Bug Relationship with Source Code Files:

is computed, which can be referred to as degree of similarity. The score can be defined as:

$$SimiScore = \sum_{AllS_i thatconnecttoF_j} (Similarity(B, S_i)/n_i) \quad (7)$$

Here, similarity between newly reported bug B and previously fixed bug S_i is calculated based on cosine similarity measure and n_i is the total number of link S_i has with source code files in layer 3.

Approach II: Final Score Calculation For each query, we compute the rVSM score against all source codes in the database using equation 6 and we also calculate Simi score using equation 7. Then we calculate co-occurence scores for the query using equation 3. We finally combine the three ranks and for that we use three weighting factor α, β and γ. The final equation is given in equation 8.

$$FinalScoreApproach1 = \gamma \times N(rVSMScore) + \beta \times N(SimiScore) + \alpha \times N(CoOccScore) \quad (8)$$

We work with different values of α, which are presented in the experiment section. We use value of 0.2 for β, varying α from 0.1 to 0.5 and thus, γ from 0.7 to 0.3. So that they end up into 1.

VII. Experiment and Discussion

In this section, at first we discuss detail of our data set, then we describe the evaluation metrics and finally we present our experimental results.

A. Experimental Dataset

We work with three different dataset - Eclipse, SWT aand Zxing. We work with Eclipse data set which is a popular IDE for Java. We downloaed a git based Eclipse project from git repository [3]. We work with Eclipse Platform UI project. On the other hand, currently Eclipse Platform UI project contains

[3]https://git.eclipse.org/c/platform/eclipse.platform.ui.git/

more than 10K number of bugs where we only work with the bugs which are fixed. We create quires from each bugs considering their title and short summary. All bug reports are collected from [4]. In order to obtain the links between previously fixed bugs and source code files, we analyze git project commit message. We ran through all commit messages and track Bug IDs associated with examined source code files. In order to evaluate our proposed tool we have also used two more dataset that Zhou et al. [26] used to evaluate BugLocator. This dataset contains 118 bug reports in total from two popular open source projects– SWT, and ZXing along with the information of fixed files for those bugs. The detail of our dataset is presented in II. SWT is a component of Eclipse [5]. Zxing is an android based project maintained by google [6].

TABLE II
DESCRIPTION OF DATA SETS

Project Name	Description	Study Period	#Fixed Bugs	#Source Files
Eclipse Platform Ant	Popular IDE for Java		3855	11732
SWT (V 3.1)	An open source widget toolkit for Java	Oct 2004 - Apr 2010	98	484
Zxing	A barcode image processing library for Android Application	Mar 2010 - Sep 2010	20	391

B. Evaluation Metrics

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

Ton N-Rank (Hit@N): It represents the number of bug, for which their associated files are returned in a ranked list. Here, *N* may be 1, 5 or 10. We assume that if at least one associated file is presented in the resulted ranked list, then the given bug is located. The higher the metric value, the better the bug localization performance

MRR(Mean Reciprocal Rank) The reciprocal rank of a query is the multiplicative inverse of the rank of the first correct answer. So mean reciprocal rank is the average of the reciprocal ranks of results of a set of queries Q

$$MRR = \frac{1}{|Q|}\sum_{i=1}^{|Q|}\frac{1}{rank_i} \tag{9}$$

where $ranki$ is the position of the first buggy file in the returned ranked files for the first query in Q.

MAP(Mean Average Precision) Mean Average Precision is the most commonly used IR metric to evaluate ranking approaches. It considers the ranks of all buggy files into consideration. So, MAP emphasizes all of the buggy files instead of only the first one. MAP for a set of queries is the mean of the average precision scores for each query. The average precision of a single query is computed as:

$$AP = \sum_{k=1}^{M}\frac{P(k)\times pos(k)}{number\,of\,positive\,instances} \tag{10}$$

where k is a rank in the returned ranked files, M is the number of ranked files and $pos(k)$ indicates whether the kth file is a buggy file or not. $P(k)$ is the precision at a given top k files and is computed as follows:

$$P(k) = \frac{\#buggy\,Files}{k} \tag{11}$$

Wilcoxon signed-rank test The Wilcoxon signed-rank test is a non-parametric statistical hypothesis test used to compare two related samples, matched samples, or repeated measurements on a single sample to assess whether their population mean ranks differ. We perform this test with the help of [7].

Cross Validation We divide our query data into k number of sets. Typically k is 10, but we work with k =5 and k=10. Each set contains a training set and tesing set. Training data is used to create mapping between keywords extracted from bug reports and source code files. 10-fold-cross validation data is presented in table IV and table III.

C. Experimental Results

During experiment, we evaluate our proposed approach in different ways. To create mapping between bug report keywords and source files, we consider three different options - (1) including only title or summary of a bug report in creating corpus, (2) in addition with title we also include description field of a bug report and (3) full content of a bug report could be an option. Neither option 1 nor 3 provides better result and option 2 optimized the performance. We explain this in a way that providing only title of a big report conveys very little information. On the other hand, including full content of a bug report also create too much information that contains huge noise data and also takes longer time during mapping them into source code files. Therefore, title and description of a bug report optimized those two options. However, considering title and description did not get rid of noise and therefore we discard all keywords that happen to exist in 25% or more documents in the corpus.

However, we divide our experimental result into two parts based on experimented dataset. In part I, we work on Eclipse dataset, where we compare the performance of BLuAMIR with BugLocator [26] and VSM based bug localization techniques. In this part, we replicate BugLocator proposed by Zhou et al. [26]. We compare the performance of our replicated BugLocator with Aproach I and Approach II of our proposed tool BLuAMIR. On the other hand, in part II we experiment with SWT and Zxing dataset as in [26]. Here we collect the results reported in [26] and then compare the performance with our Approach I of BLuAMIR. We also answer our research questions at the end of this section. In the next two subsections we will discuss our experimental results.

[4]https://bugs.eclipse.org/
[5]http://www.eclipse.org/swt/
[6]http://code.google.com/p/zxing/
[7]https://www.socscistatistics.com/tests/signedranks/Default.aspx

D. Performance Comparison of Approach I and Approach II on Eclipse Dataset

We compare the performance of our proposed bug localization approach with two existing techniques - 1) BugLocaotor [26] which is based on rVSM and Simi scores and 2) VSM which is based on vector space model. In this section we work on Eclipse dataset and apply Approach I and Approach II of BLuAMIR on that dataset.

TABLE III
PERFORMANCE OF PROPOSED TECHNIQUE (VSM+CO-OCCERENCE) RANKS

# Test Case	#Methodology	Top 1 %	Top 5 %	Top 10 %	MRR@10	MAP@10
1	VSM	23.67	46.59	56.97	0.33	0.32
	VSM + Co-Score	28.40	51.77	60.06	0.38	0.36
2	VSM	24.62	48.05	57.96	0.34	0.34
	VSM + Co-Score	27.63	53.15	61.26	0.38	0.37
3	VSM	20.78	40.96	51.20	0.30	0.29
	VSM + Co-Score	23.12	46.85	59.76	0.34	0.32
4	VSM	22.52	42.94	53.75	0.32	0.31
	VSM + Co-Score	23.42	49.25	61.56	0.35	0.33
5	VSM	27.33	52.25	63.36	0.38	0.35
	VSM + Co-Score	29.73	53.15	62.16	0.39	0.36
6	VSM	25.00	50.60	60.84	0.36	0.34
	VSM + Co-Score	26.13	51.65	62.76	0.37	0.35
7	VSM	29.82	54.52	66.27	0.40	0.38
	VSM + Co-Score	35.43	60.96	72.07	0.46	0.44
8	VSM	27.79	51.36	60.73	0.38	0.36
	VSM + Co-Score	36.64	58.86	67.87	0.46	0.43
9	VSM	29.13	52.55	64.86	0.39	0.36
	VSM + Co-Score	29.13	61.86	69.97	0.42	0.40
10	VSM	19.03	39.58	51.34	0.28	0.26
	VSM + Co-Score	24.62	51.05	63.06	0.36	0.34
Average	VSM	24.98%	47.94%	58.73%	0.35	0.33
	VSM + Co-Score	28.43%	53.86%	64.05%	0.39	0.37

VSM vs Our proposed Tool We combine VSM score and co-occurence scores in order to produce ranked result. We also compare the performance of baseline VSM technique and our proposed combined approach. The comparison is presented in table III. We compute Top-1, Top-5, Top-10 performance and MRR and MAP for both approaches. In all cases our proposed approach outperforms VSM-based bug localization approach. The Top-10 performance of our tool is 64.05% whereas it is 58.73% for VSM.

We also compute Wilcoxon signed-rank test both for MRR and MAP. For MRR the Z -value is -2.8031. The p -value is 0.00512. The result is significant at p¡=0.05. The W-value is 0. The critical value of W for N = 10 at p¡=0.05 is 8. Therefore, the result is significant at p¡=0.05. For MAP - the Z -value is -2.8031. The p -value is 0.00512. The result is significant at p¡=0.05. The W-value is 0. The critical value of W for N = 10 at p¡=0.05 is 8. Therefore, the result is significant at p¡=0.05.

BugLocator VS Our Proposed Tool: We combine rVSM and simi ranks with our co-occurence rank. Here, co-occurence rank is computed based on keyword-source code mapping database.In table IV we, compare the performance of our proposed approach in terms of top 1, 5, 10 rank, MRR and MAP. We can see that our proposed approach outperforms in all cases. For example, our Top-10 performance 52.94% has an improvement than BugLocator (36.40%).

We also compute Wilcoxon signed-rank test both for MRR and MAP. For MRR the Z -value is -2.8031. The p -value is 0.00512. The result is significant at p¡=0.05. The W-value is 0. The critical value of W for N = 10 at p¡=0.05 is 8. Therefore, the result is significant at p¡=0.05. For MAP - the Z -value is -2.8031. The p -value is 0.00512. The result is significant at p¡=0.05. The W-value is 0. The critical value of W for N = 10 at p¡=0.05 is 8. Therefore, the result is significant at p¡=0.05.

TABLE IV
PERFORMANCE OF BUGLOACTOR AND PROPOSED TECHNIQUE (RVSM+SIMI+CO-OCCERENCE)

# Test Case	Methodology	Top 1 %	Top 5 %	Top 10 %	MRR	MAP
1	BugLocator	8.88	24.56	34.32	0.16	0.15
	rVSM+Simi+ Co-Score	16.27	39.35	49.41	0.26	0.25
2	BugLocator	9.9	24.62	34.53	0.17	0.16
	rVSM+Simi+ Co-Score	18.32	41.44	53.45	0.28	0.27
3	BugLocator	7.50	21.32	30.63	0.14	0.13
	rVSM+Simi+ Co-Score	18.62	42.34	52.25	0.28	0.27
4	BugLocator	8.1	21.02	29.13	0.14	0.13
	rVSM+Simi+ Co-Score	14.71	42.04	54.35	0.26	0.25
5	BugLocator	9.6	30.63	42.94	0.18	0.18
	rVSM+Simi+ Co-Score	22.22	42.34	57.66	0.31	0.30
6	BugLocator	10.21	31.53	43.54	0.19	0.19
	rVSM+Simi+ Co-Score	25.53	52.25	60.66	0.36	0.34
7	BugLocator	9.61	30.33	40.84	0.18	0.17
	rVSM+Simi+ Co-Score	25.22	52.25	64.56	0.36	0.34
8	BugLocator	8.4	26.13	39.94	0.19	0.16
	rVSM+Simi+ Co-Score	24.92	51.95	62.46	0.36	0.34
9	BugLocator	11.11	28.83	40.24	0.19	0.18
	rVSM+Simi+ Co-Score	21.92	48.65	62.46	0.33	0.32
10	BugLocator	6.6	20.72	27.93	0.12	0.12
	rVSM+Simi+ Co-Score	16.21	40.54	55.55	0.27	0.26
Average	BugLocator	8.99%	25.87%	36.40%	0.16	0.16
	rVSM+Simi+ Co-Score	20.39%	45.32%	57.28%	0.31	0.29

E. Comparison with State-of-art Technique

We compare the performance of BLuAMIR with BugLocator for the the same dataset for SWT and Zxing presented in Table V. Here, the results from buglocator is directly copied from their paper [26]. We also collect the same bug reports and the same source code repository for both of them. So the results can be compared. For SWT, we can see our tool BLuAMIR performs better for Top-1, Top-5, MRR and MAP. However, for Top-10 BLuAmir is comparable with Buglocator. On the other hand, for Zxing our tool BLuAMIR outperforms for top-5, top-10 and MAP.

TABLE V
PERFORMANCE COMPARISON BETWEEN BUGLOCATOR AND BLUAMIR

# System	#Localization Approach	Top 1 %	Top 5 %	Top 10 %	MRR@10	MAP@10
SWT	BugLocator	39.80	67.35	81.63	0.53	0.45
	BLuAMIR	44.90	73.45	80.61	0.57	0.54
Zxing	BugLocator	40.00	60.00	70.00	0.50	0.44
	BLuAMIR	30.00	70.00	75.00	0.48	0.46

F. Weighting Factor Analysis

For both of our proposed approaches we have used some weighting functions, which are described as follows:

Weighting Function for VSM+Co-ocuurence Ranking (Approach I): For our Approach I, we also compute performance TopK accuracy, MRR and MAP for different weighting function such as *ALPHA* is 0.2, 0.3, 0.4. The results are presented in Table VI. Here, it shows, more α produces better

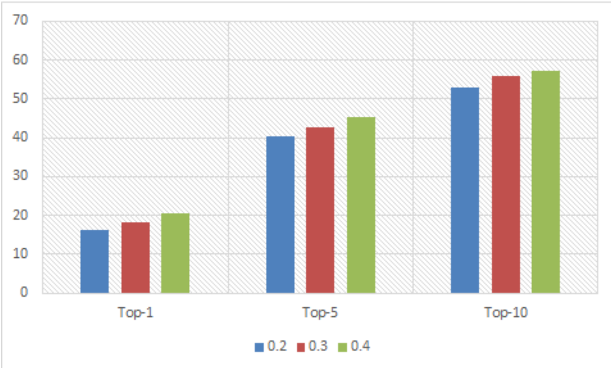

Fig. 3. The impact of α on bug localization performance (Top-1, Top-5, Top-10)
for proposed approach1.

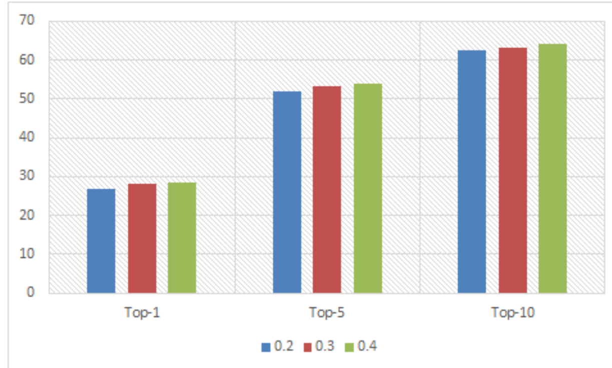

Fig. 4. The impact of α on bug localization performance (Top-1, Top-5, Top-10)
for proposed approach2.

performance. That means if we increase the co-occurence scores with higher weighting function, the better performance is resulted in this proposed approach. We also illustrate the impact of α for Top-1, Top-5 and Top-10 retrieval on Eclipse dataset for approach I. (VSM+Co-ocuurence Ranking) in Figure 4.

TABLE VI
PERFORMANCE OF (VSM+CO-OCCERENCE) FOR DIFFERENT WEIGHTING FACTORS

α	**Top 1 %**	**Top 5 %**	**Top 10 %**	**MRR**	**MAP**
0.2	26.72	51.94	62.43	0.37	0.36
0.3	28.06	53.35	63.24	0.39	0.37
0.4	28.43	53.86	64.05	0.39	0.37
Average	27.74%	53.05%	63.24%	0.38	0.37

Weighting Function for rVSM+Simi+Co-ocuurence Ranking (Approach II): We compute performance TopK accuracy, MRR and MAP for different weighting function such as α is 0.2, 0.3, 0.4, β is 0.2 nad γ is 0.6, 0.5, 0.4 respectively. The results are presented in Table VII. Here, it shows, α produces better performance. That means if we increase the co-occurence scores with higher weighting function, the better performance is resulted. This also prove our co-occurence rank is effective in producing better results. We also represent the impact of α for Top-1, Top-5 and Top-10 retrieval on Eclipse dataset for approach 1 (rVSM+Simi+Co-ocuurence Ranking) in figure 3.

TABLE VII
PERFORMANCE OF (RVSM+SIMI+CO-OCCERENCE) FOR DIFFERENT WEIGHTING FACTORS

α	β	γ	**Top 1 %**	**Top 5 %**	**Top 10 %**	**MRR**	**MAP**
0.2	0.2	0.6	16.13	40.40	52.99	0.26	0.25
0.3	0.2	0.5	18.26	42.83	55.73	0.29	0.27
0.4	0.2	0.4	20.40	45.32	57.28	0.31	0.29
Average			18.26%	42.85%	55.33%	0.29	0.27

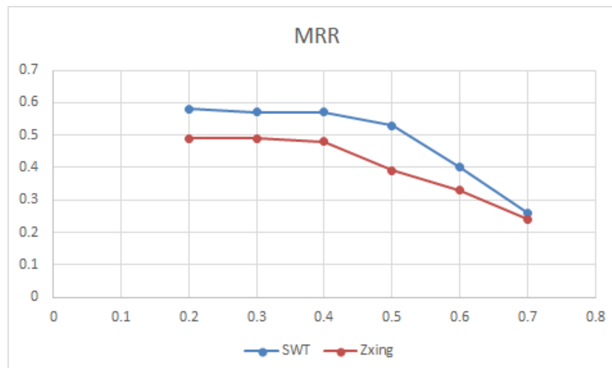

Fig. 5. The impact of α on bug localization performance (MRR)

G. *Impact of varying the value of alpha on BLuAMIR in terms of MAP and MRR*

We also evaluate the impact of co-occurence score on bug localization performance, with different $\aleph$ values in terms of MAP and MRR for SWT and Zxing. At the beginning, the bug localization performance increases when the α value increases. However, after a certain point, further increase of the b value will decrease the performance. For example, Figure 5 and 6 show the bug localization performance (measured in terms of

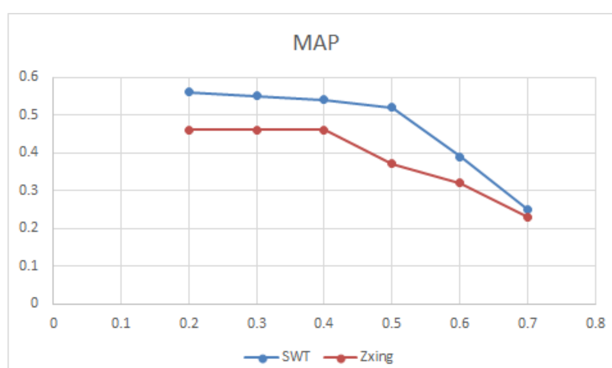

Fig. 6. The impact of α on bug localization performance (MAP)

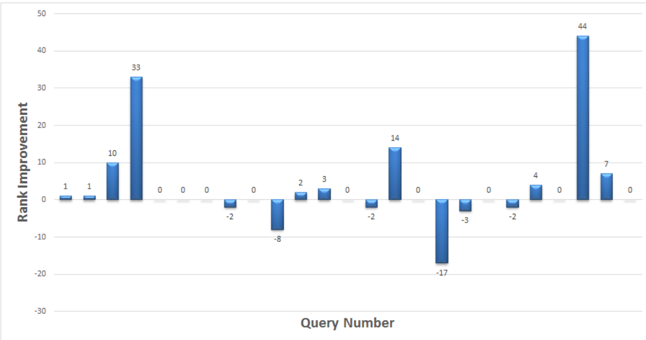

Fig. 7. Quey wise comparison between VSM and BLuAMIR for 30 Eclipse Query

MRR and MAP) for the SWT and Zxing projects. When the $\aleph$ value increases from 0.1 to 0.4, both MRR and MAP values increases. Increasing α value further from 0.4 to 0.7 however leads to lower performance. Note that we obtain the best bug localization performance when α is between 0.3 and 0.4.

H. Answering Research Questions

Answering RQ1: To answer our *RQ1*, we can go to Table III, 64.05% bugs are successfully located in Top-10 for Eclipse dataset. Our proposed tool BLuAMIR also outperforms on Zxing dataset for Top-10 retrieval, which is 75.00%. However, on the subject system SWT, the Top-10 result of BLuAMIR is comparable, which is 80.61%.

Answering RQ2: We combine co-occurence rank with VSM score in Approach1 and rVSM and Simi scores in Approach2. We apply both of these approaches on Eclipse dataset. We also replicate baseline techniques for those results, which are provided in Table IV and Table III. Here, we can see that both cases BLuAMIR outperforms in terms of Top-1, Top-5, Top-10, MRR and MAP. As co-occurence score is based on the association map between keywords and source files, thus no direct matching of vocabulary is required. Adding co-occurence score increases the performance in both cases also indicate that the association map is helping in locating buggy files. Therefore, vocabulary miss-match problem is resolved here.

Answering RQ3: We investigate how vocabulary miss-match problem is eliminated in BLuAMIR. We perform a query wise ranking comparison for Approach1 for 30 query from Eclipse system, which is given in Figure 7. Here, X-axis represents query number and Y-axis represents the difference of best rank retried by VSM annd our proposed BLuAMIR. Among 25 query, 10 cases BLuAMIR performs better than VSM, 6 cases VSM retrieves better ranked results than BLuAMIR and 9 cases they both do the same raking retrieval. As most cases BLuMIR provides better ranked results,

***Find some queries having vocabulary miss-match problem

Answering RQ4: We compare the performance of BLuAMIR with state-of-the-art bug localization technique, BugLocator proposed by Zhou et al. [26] on two dataset i.e., SWT and Zxing. These results can be found in Table V. For SWT, we can see our tool BLuAMIR performs better for Top-1, Top-5, MRR and MAP. However, for Top-10 BLuAmir is comparable with Buglocator. On the other hand, for Zxing our tool BLuAMIR outperforms for top-5, top-10 and MAP. So we can say that our proposed tool BLuAMIR outperforms most cases and comparable for a few cases with state-of-the-art bug localization technique.

VIII. Related Work

There are many bug localization approaches proposed so far. They can be broadly categorized into two types - dynamic and static techniques. Generally, dynamic approaches can localize a bug much more precisely than static approaches. 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. Some of the well known dynamic approaches are spectrum-based fault localization, e.g., [1, 6, 7, 16], model-based fault localization, e.g., [3, 10], dynamic slicing [25], delta debugging [24].

Static approaches, on the other hand, do not require any program test cases or execution traces. In most cases, they need only program source code and bug reports. They are also computationally efficient. The static approaches usually can be categorized into two groups: program analysis based approaches and IR-based approaches. FindBugs is a program analysis based approach that locates a bug based on some predefined bug patterns [5]. Therefore, FindBug does not even need a bug report. However, it often detects too many false positives and misses many real bugs [20]. IR-based approaches use information retrieval techniques (such as, TFIDF, LSA, LDA, etc.) to calculate the similarity between a bug report and a source code file. There are three traditionally-dominant IR paradigms TF.IDF [18], the "probabilistic approach" known as BM25 [15], or more recent language modeling [13]. Another empirical study [2] show that all three approaches perform comparably when well-tuned. However, Rao and Kak [14] investigates many standard information retrieval techniques for bug localization and find that simpler techniques, e.g., TFIDF and SUM, perform the best.

In contrast with shallow "bag-of-words" models, latent semantic indexing (LSI) induces latent concepts. While a probabilistic variant of LSI has been devised [4], its probability model was found to be deficient. Lukins et al. [9] use Latent Dirichlet Allocation (LDA), which is a well-known topic modeling approach, to localize bug [9]. However, LSI is rarely used in practice today due to errors in induced concepts introducing more harm than good [4] and LDA is not be able to predict the appropriate topic because it followed a generative topic model in a probabilistic way [8].

Sisman and Kak [19] propose a history-aware IR-based bug localization solution to achieve a better result. Zhou et al. [26]

propose BugLocator, which leverages similarities among bug reports and uses refined vector space model to perform bug localization. Saha et al. [17] build BLUiR that consider the structure of bug reports and source code files and employ structured retrieval to achieve a better result. Moreno et al. [11] uses a text retrieval based technique and stack trace analysis to perform bug localization. To locate buggy files, they combines the textual similarity between a bug report and a code unit and the structural similarity between the stack trace and the code unit. Different from the existing IR-based bug localization approaches, Wang and Lo [21] propose AmaLgam, a new method for locating relevant buggy files that put together version history, similar report, and structure, to achieve better performance. Later [22] also propose AmaLgam+, which is a method for locating relevant buggy files that puts together fives sources of information i.e., version history, similar reports, structure, stack traces, and reporter information. In our proposed technique BLuAMIR, we use version history, similar report and association relationship.

IX. Threats To Validity

This section discusses the validity and generalizability of our findings. In particular, we discuss Construct Validity, Internal Validity, and External Validity.

Internal Validity: We used three artifacts of a software repository: bug Reports, source codes and version logs, which are generally well understood. Our evaluation uses three dataset - two of them collected from the same benchmark dataset of bug reports and source code shared by Zhou et al. [26], and other is collected from an open source project. Bug reports provide crucial information for developers to fix the bugs. A "bad" bug report could cause a delay in bug fixing. Our proposed 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 BLuAMIR is adversely affected.

External Validity: 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. Then we will perform statistical tests to show that the improvement of our approach is statistically significant.

Construct Validity In our experiment, we use three evaluation metrics, i.e., Top N rank, MAP and MRR, and one statistical test, i.e., Wilcoxon signed-rank test. These metrics have been widely used before to evaluate previous approaches [17, 26] and are well- known IR metrics. Thus, we argue that our research has strong construct validity.

Reliability: In our experiment section, we performed numerous experiments using various combinations of weighting functions to find the optimum parameters and the best accuracy of bug localization. The optimized α, β, and γ values are based on our experiments and are only for our proposed tool BLuAMIR. To automatically optimize control parameters for target projects, in the future we will expand our proposed approach using machine learning methods or generic algorithms.

X. Conclusion and Future Work

During software evolution of a system, a large number of bug reports are submitted. For a large software project, developers must may need to examine a large number of source code files in order to locate the buggy files responsible for a bug, which is a tedious and expensive work. In this paper, we propose BLuAMIR, a new method for locating relevant buggy files that combines historical data, similar report, and keyword-source association map to achieve a higher accuracy. We perform a large-scale experiments on four projects, namely Eclipse, SWT and ZXing to localize more than 3,000 bugs. Our experiment of those dataset show that our technique can locate buggy files with a Top-10 accuracy of 64.05% and a mean reciprocal rank@10 of 0.31 and a mean precision average@10 of 37%, which are highly promising. We also compare our technique with state-of-the-art IR-based bug localization technique i.e., BugLocator. This also confirms superiority of our technique.

In the future, we will explore if several other bug related information such as bug report structure, source code structure, stack traces, reporter information can be integrated into our approach in order to improve bug localization performance. We would also like to reduce the threats to external validity further by applying our approach on more bug reports collected from other software systems.

References

[1] R. Abreu, P. Zoeteweij, R. Golsteijn, and A. J. C. van Gemund. A Practical Evaluation of Spectrum-based Fault Localization. *J. Syst. Softw.*, 82(11):1780–1792, November 2009.

[2] H. Fang, T. Tao, and C. Zhai. A Formal Study of Information Retrieval Heuristics. In *Proceedings of the 27th Annual International ACM SIGIR Conference on Research and Development in Information Retrieval*, SIGIR '04, pages 49–56.

[3] A. Feldman and A. van Gemund. A Two-step Hierarchical Algorithm for Model-based Diagnosis. In *Proceedings of the 21st National Conference on Artificial Intelligence - Volume 1*, AAAI'06, pages 827–833, 2006.

[4] Thomas Hofmann. Probabilistic Latent Semantic Indexing. *SIGIR Forum*, 51(2): 211–218.

[5] D. Hovemeyer and W. Pugh. Finding Bugs is Easy. *SIGPLAN Not.*, 39(12):92–106.

[6] 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 '05, pages 273–282, 2005.

[7] Lucia, D. Lo, Lingxiao Jiang, and A. Budi. Comprehensive Evaluation of Association measures for Fault Localization. In *2010 IEEE International Conference on Software Maintenance*, pages 1–10, 2010.

[8] S.K. Lukins, N.A. Kraft, and L.H. Etzkorn. Source Code Retrieval for Bug Localization Using Latent Dirichlet Allocation. In *Reverse Engineering, 2008. WCRE '08. 15th Working Conference on*, pages 155–164, Oct 2008.

[9] Stacy K. Lukins, Nicholas A. Kraft, and Letha H. Etzkorn. Bug localization using latent dirichlet allocation. *Inf. Softw. Technol.*, 52(9):972–990, September 2010.

[10] W. Mayer and M. Stumptner. Model-based Debugging – State of the Art And Future Challenges. *Electron. Notes Theor. Comput. Sci.*, 174(4):61–82.

[11] L. Moreno, J.J. Treadway, A. Marcus, and Wuwei Shen. On the Use of Stack Traces to Improve Text Retrieval-Based Bug Localization. In *Software Maintenance and Evolution (ICSME), 2014 IEEE International Conference on*, pages 151–160, Sept 2014.

[12] Anh Tuan Nguyen, Tung Thanh Nguyen, J. Al-Kofahi, Hung Viet Nguyen, and T.N. Nguyen. A Topic-Based Approach for Narrowing the Search Space of Buggy Files from a Bug Report. In *Automated Software Engineering (ASE), 2011 26th IEEE/ACM International Conference on*, pages 263–272, Nov 2011.

[13] Jay M. Ponte and W. Bruce Croft. A Language Modeling Approach to Information Retrieval. In *Proceedings of the 21st Annual International ACM SIGIR Conference on Research and Development in Information Retrieval*, SIGIR '98, pages 275–281.

[14] Shivani Rao and Avinash Kak. Retrieval from Software Libraries for Bug Localization: A Comparative Study of Generic and Composite Text Models. In *Proceedings of the 8th Working Conference on Mining Software Repositories*, MSR '11, pages 43–52.

[15] S. E. Robertson, S. Walker, and M. Beaulieu. Experimentation As a Way of Life: Okapi at Trec. *Inf. Process. Manage.*, 36(1):95–108.

[16] D. Saha, M. G. Nanda, P. Dhoolia, V. K. Nandivada, V. Sinha, and S. Chandra. Fault Localization for Data-centric Programs. In *Proceedings of the 19th ACM SIGSOFT Symposium and the 13th European Conference on Foundations of Software Engineering*, ESEC/FSE '11, pages 157–167, 2011.

[17] R.K. Saha, M. Lease, S. Khurshid, and D.E. Perry. Improving Bug Localization using Structured Information Retrieval. In *Automated Software Engineering (ASE), 2013 IEEE/ACM 28th International Conference on*, pages 345–355, Nov 2013.

[18] G. Salton and C. Buckley. Term-weighting Approaches in Automatic Text Retrieval. *Inf. Process. Manage.*, 24(5):513–523.

[19] B. Sisman and A.C. Kak. Assisting Code Search with zAutomatic Query Reformulation for Bug Localization. In *Mining Software Repositories (MSR), 2013 10th IEEE Working Conference on*, pages 309–318, May 2013.

[20] H. Tang, S. Tan, and X. Cheng. A Survey on Sentiment Detection of Reviews. *Expert Systems with Applications*, 36(7), 2009.

[21] Shaowei Wang and David Lo. Version History, Similar Report, and Structure: Putting Them Together for Improved Bug Localization. In *Proceedings of the 22Nd International Conference on Program Comprehension*, ICPC 2014, pages 53–63, 2014.

[22] Shaowei Wang and David Lo. Amalgam+: Composing Rich Information Sources for Accurate Bug Localization. *Journal of Software: Evolution and Process*, 28: 921–942, 2016.

[23] Chu-Pan Wong, Yingfei Xiong, HongYu Zhang, Dan Hao, Lu Zhang, and Hong Mei. Boosting Bug-Report-Oriented Fault Localization with Segmentation and Stack-Trace Analysis. In *Software Maintenance and Evolution (ICSME), 2014 IEEE International Conference on*, pages 181–190, Sept 2014.

[24] A. Zeller and R. Hildebrandt. Simplifying and Isolating Failure-Inducing Input. *IEEE Trans. Softw. Eng.*, 28(2):183–200.

[25] X. Zhang, H. He, N. Gupta, and R. Gupta. Experimental Evaluation of Using Dynamic Slices for Fault Location. In *Proceedings of the Sixth International Symposium on Automated Analysis-driven Debugging*, AADEBUG'05, pages 33–42, 2005.

[26] Jian Zhou, Hongyu Zhang, and D. Lo. Where Should the Bugs be Fixed? More Accurate Information Retrieval-Based Bug Localization Based on Bug Reports. In *Software Engineering (ICSE), 2012 34th International Conference on*, pages 14–24, June 2012.