Improved Bug Localization using Association Mapping and Information Retrieval

***Abstract*—Bug localization is one of the most challenging tasks undertaken by the developers during software maintenance. Most of the existing studies rely on lexical similarity between the bug reports and source code for bug localization. Unfortunately, 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 a given 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 0.37 and a mean precision average@10 of 39%, which are highly promising. Comparison with the state-of-the-art techniques and their variants report that our technique can improve 12% in MAP@10 and 1.94% in MRR@10 over the state-of-the-art.**

***Index Terms*—bug localization, bug report, source code, information retrieval**

I. Introduction

Bug localization is the process of locating the source code that needs to be changed in order to fix a given bug. Locating buggy files is time-consuming and costly in terms of efforts [22]. This is even more challenging especially for the large software systems. Thus, effective methods for locating bugs automatically from bug reports are highly desirable. Traditional bug localization techniques accept a bug report and a subject system as inputs and produce a list of entities (e.g., classes, methods) for the bug report. Information retrieval based techniques consider relevance between bug report and source code in terms of their lexical similarity, ~~and return a ranked list of source code entities which may contain the bug~~. Hence, these techniques are affected by the quality and content of a submitted query. If a query contains inadequate information, then the retrieval results might not be relevant at all. Thus, simply relying on lexical similarity might not be helpful. In this work, we additionally consider an association between bug report ~~repository~~ and the corresponding codebase through commit logs for the bug localization.

Zhou et al. [26] proposed BugLocator based on rVSM that uses the lexical similarity between bug report texts and source code. Saha et al. [18] capture the structures of both bug reports and source documents, and then apply structured information retrieval for bug localization. Moreno et al. [13] consider structural similarity between stack traces in bug report and the project document structures for bug localization. However, all of these studies share two common limitations ~~as follows.~~

First, most of the bug localization techniques simply rely on only lexical similarity for retrieving relevant documents from the code base. Hence, the search query constructed from a given bug report should contain keywords similar to code tern[illegible]his also warrants for expertise on given project, which can always not be guaranteed ~~in real world~~.

In our work we propose a bug localization approach namely BLUAMIR that not only considers lexical similarity but also captures implicit association between keywords and the source code. We consider the implicit association between keywords extracted from previously fixed bug reports and their corresponding changed code. The idea ~~behind~~ is to capture the ~~implici~~t relationship without direct vocabulary match. Thus, our proposed approach addresses the vocabulary mismatch problem using a keyword-source association constructed from fixed bug information. Our proposed technique also overcomes the issue of locating large files. Construction of association mapping between large source documents and an query does not require any direct vocabulary matching. Thus, our approach is not affected by the noise from large source documents and can rank the documents better than the approaches based on lexical similarity only. ~~We compare the performance of our proposed tool with a state of the art bug localization techniques [26].~~

We evaluate our technique in several different dimensions using three widely used performance metrics and 3973 bug reports (i.e., queries) from three open source subject systems. First, we evaluate in terms of the performance metrics, contrast with the replicated baseline, and BLuAMIR localizes bugs with in average 9% higher accuracy (i.e., Hit@10), 12% higher precision (i.e., MAP@10) and and 11 higher result ranks (i.e., MRR@10) than the replicated baseline (Section). Second, we compare our technique with a state of the art approach [26] and our technique can improve 12% in MAP@10 ~~and 1.94% in MRR@10~~ over the state-of-the-art (Section).

So, contributions of our paper include:

- A novel bug localization technique that not only considers lexical similarity but also exploits the association relationship between bug report keywords and their corresponding buggy source code.
- Comprehensive evaluation of the technique using total of 3973 bugs from Eclipse, SWT and Zxing bug repository and validation against state-of-the-art bug localization technique [26].

II. Motivating Example

Let us consider a bug report (ID 37026) on the Eclipse-UI-Platform. Table II shows the title and description of the encountered bug. We preprocess the title as well as the

TABLE I
A WORKING EXAMPLE OF BLUAMIR

Rank	Retrieved Files (VSM Approach)	Score_VSM	Ground Truth	Retrieved Files (BLuAMIR)	Score_VSM	Score_Assoc	Score_Total	Ground Truth
1	WorkingSetLabelProvider.java	1.00	✓	WorkingSetLabelProvider.java	0.60	0.25	0.85	✓
2	ImageFactory.java	0.80	✗	WorkingSet.java	0.35	0.35	0.70	✓
3	WorkingSetMenuContributionItem.java	0.76	✗	IWorkingSet.java	0.45	0.22	0.67	✓
4	IWorkingSet.java	0.75	✓	WorkingSetTypePage.java	0.38	0.25	0.63	✗
5	ProjectImageRegistry.java	0.70	✗	CommandImageService.java	0.30	0.33	0.63	✗

TABLE II
A BUG REPORT (#37026, ECLIPSE.UI.PLATFORM)

Field	Content
Title	[Working Sets] IWorkingSet.getImage should be getImageDescriptor
Description	build 20030422 IWorkingSet.getImage returns an ImageDescriptor and should therefore be named getImageDescriptor. This is new API in 2.1. Should consider deprecating getImage and creating getImageDescriptor.

description of this bug report, and form a query. Now we apply both Vector Space Model (VSM) and our approach on this query, and then locate possible buggy files. ~~The retrieved ranked results are presented in Table I.~~

~~From the commit log, it is found that gold set contains 14 number of source code for this bug.~~ In Table I, we can see that our proposed tool BLuAMIR correctly identifies 3 buggy files where VSM approach locates only 2 buggy files in Top-5. ~~The ranking of BLuAMIR is 1,2,3 where the rank from VSM is 1 and 4. VSM and Association scores are also provided for each recommended buggy file. Note that Same file is retrieved in rank 1 for both techniques~~. The buggy file, IWorkingSet.java was in the fourth position by VSM, but BLUAMIR returned it at the second position which is promising.

We also investigate why BLuAMIR can locate more buggy files than VSM. ~~We know that~~ VSM score is based on overlap of terms presented both in the bug reports and source files. But, in computing association 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. Our proposed approach makes use of an association map that successfully captures previous relationship between bug report keywords and their source buggy files. In BLuAMIR, we collect keywords from a current bug report and then locate their association with source files from the association map. This association map is created from keywords extracted from previously fixed bug report into their fixed buggy files. The idea is, if a current bug shares same keywords (i.e., concepts) with a previous fixed bug, there is possibility their corresponding codebase would be similar. This is how our proposed approach overcome vocabulary missmatch problem.

III. BLUAMIR: PROPOSED METHOD FOR BUG LOCALIZATION

Figure 1 shows the schematic diagram of our proposed approach. First, we (a) create an association mapping where the inherent association between past bug reports and their changed documents are leveraged. Then we (b) retrieve relevant ranked source code files based on two different scores - VSM and association. We discuss different parts of our proposed approach, BLuAMIR in the following sections.

A. Association Mapping Database Construction

We construct an association mapping database - between keywords extracted from previously fixed bug reports and source code links. This mapping construction involves two steps. First we create association map using information contained in bug reports and commit logs. We collect title and description of each bug report. We extract keywords from each bug report after standard natural language preprocessing (i.e., punctuation removal, stop word removal). From version repository, we retrieve source code files which have been changed in order to fix a given bug. It should be noted each commit establishes a direct relationship between its changed documents and the corresponding bug report. Thus, the keywords from this bug report enjoy an implicit relationships with the changed source documents. We construct an association map database by leveraging this connected relationship. Mapping each keyword can be linked to one or more source code files according to commit logs. Similarly, each source code file can be linked to one or more keywords. Algorithm 1 shows pseudo-code for the construction of our mapping database. We divide this into three steps step as follows.

Keyword Extraction from Bug Reports: We collect title and description from each fixed bug report from a collection of previously fixed bug reports.. We perform standard natural language pre-processing on the corpus such as stop word removal, splitting and stemming. Stop words contain very little semantic for a sentence. Stemming 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 identify those commits that contain keywords related to bug fix such as fix, fixed and so on. In GitHub, we note that any commit that either solves a software bug or implements a feature request generally

[1] 1https://www.ranks.nl/stopwords
[2] 2http://tartarus.org/martin/PorterStemmer/

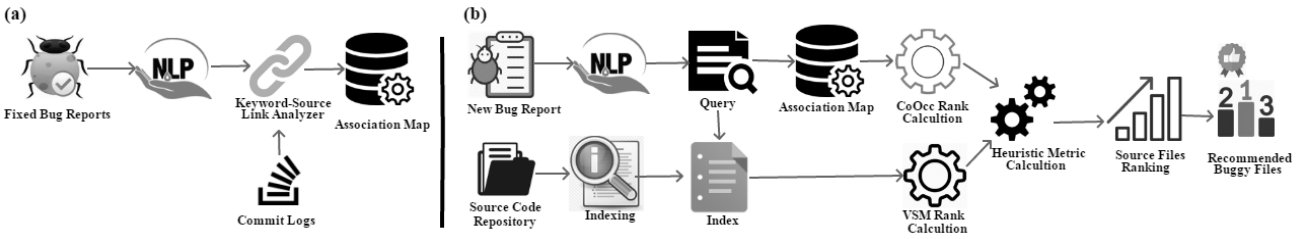

Fig. 1. Proposed Schematic Diagram: (a)Construction of association map between bug report keywords and source documents and (b) Bug Localization Using VSM and association score

Algorithm 1 Construction of Association Mapping Database

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

mentions the corresponding Issue ID in the very title of the commit. We identify such commits from the commit history of each of the selected projects using suitable regular expressions, and select them for our experiments [3]. Then, we collect the changeset (i.e., list of changed files) for each of those commit operations, and develop solution set (i.e., goldset) for the corresponding change tasks. Thus, for experiments, we collect not only the actual change requests from the reputed subject systems but also their solutions which were applied in practice by the developers [1]. We use several utility commands such as git, clone and log on GitHub for collecting those information

Keyword-Source Code Linking: At this point, we have pre-processed keywords from each bug report. We also have a implied relationship between bug ID and buggy source code links. We construct the bipartite graph between keywords collected from a bug report and buggy source document links. Here, one or more keywords can be linked to a single buggy source code files links. A source code file can be linked to one or more keywords, which is constructed from lines 4-12 in Algorithm 1.

B. Bug Localization Using VSM and Association Scores

The system diagram for this part has illustrated in Figure 1(b). In BLuAMIR, we perform bug localization in two different ways - Approach I is based on the combination of VSM and association scores (i.e., Figure 1 (b)) and Approach II is based on rVSM, Simi and association scores (i.e., Figure 1 (c)). We have constructed an association map databases from keywords collected from bug report into its corresponding source code information in Section III-A. 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.

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:

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

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

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

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.

Association 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 association. The equation for computing association score is given belows:

$$AssociationScore = \sum_{All S_i that connect to W_j} (Link(Wj, S_i)) \quad (3)$$

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.

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

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

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

C. Bug Localization using rVSM, Simi and Association Scores:

In our second approach, we work with three different ranks i.e., rVSM, Simi and Co-occurence given in Figure 1. We replicate an existing technique proposed by Zhou et al. [26] for computing rVSM and Similarity scores. If a new bug is similar to a given previously located bug, then there is a possibility that the source code files located for the past bug can provide useful information in finding buggy files for that new bug.

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:

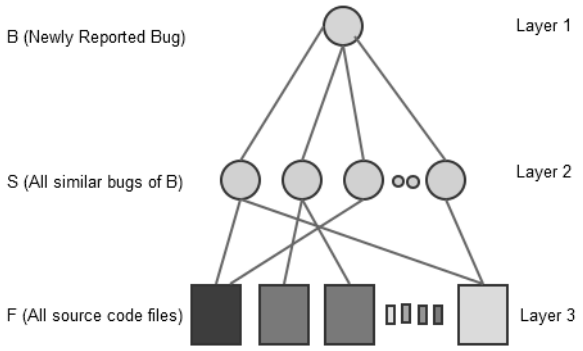

Fig. 2. Bug and its Similar Bug Relationship with Source Code Files:

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

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

$$SimiScore = \sum_{All S_i that connect to F_j} (Similarity(B, S_i)/n_i) \quad (7)$$

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.

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 association 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.

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

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.

IV. Experiment and Discussion

In this section, at first we discuss detail of our data set, then we describe the evaluation metrics, research questions 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 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 III. SWT is a component of Eclipse [5]. Zxing is an android based project maintained by google [6].

B. Evaluation Metrices

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

[3]https://git.eclipse.org/c/platform/eclipse.platform.ui.git/
[4]https://bugs.eclipse.org/
[5]http://www.eclipse.org/swt/
[6]http://code.google.com/p/zxing/

TABLE III
Description of Data Sets

Project Name	Description	Study Period	#Fixed Bugs	#Source Files
Eclipse Platform Ant	Popular IDE for Java	Nov 2001 - April 2010	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

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

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

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:

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

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:

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

C. Research Questions

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 and how?
- RQ3: Does our proposed approach-BLuAMIR eliminate large files problem and how?
- RQ4: In BLuAMIR comparable with the state-of-the-art techniques in identifying buggy files?

D. 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.

We work on Eclipse dataset, where we compare the performance of BLuAMIR with BugLocator [26] and VSM based bug localization techniques. For that, 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, we also 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 in the following subsections.

E. Answering RQ1

To answer RQ1, 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 IV
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 IV. 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. So. to answer our *RQ1*, we can go to Table IV, 64.05% bugs are successfully located in Top-10 for Eclipse dataset for Approach I of BLuAMIR.

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 V
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

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 V 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%). This also answers our RQ1.

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.

F. Answering RQ4

To answer RQ4, we compare the performance of BLuAMIR with BugLocator for the the same dataset for SWT and Zxing

presented in Table VI. 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. 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.

TABLE VI
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

G. Answering RQ2

To answer RQ2, we investigate several weighting functions for both of our proposed approaches, 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 VII. Here, it shows, more α produces better performance. That means if we increase the co-occurence scores with higher weighting function, the better performance is resulted in this proposed approach. Adding co-occurence score increases the performance in this case also indicate that the association mapping is helping in locating buggy files. Therefore, vocabulary miss-match problem is resolved here (i.e., answering RQ2). 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 VII
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 VIII. 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. Adding co-occurence score increases the performance also indicate that the association map is helping in locating buggy files. Therefore, vocabulary miss-match problem is resolved here (i.e., answering RQ2). 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.

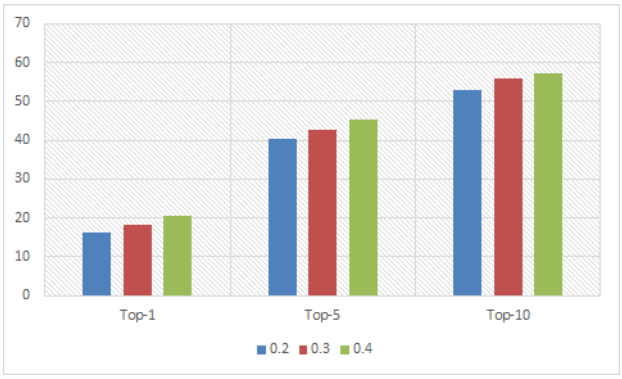

Fig. 3. The impact of α on bug localization performance (Top-1, Top-5, Top-10) for proposed approach1.

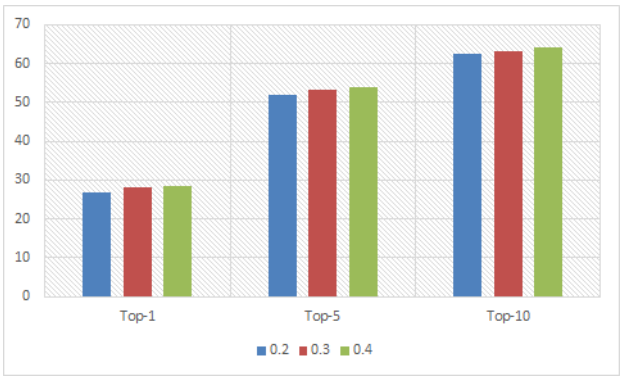

Fig. 4. The impact of α on bug localization performance (Top-1, Top-5, Top-10) for proposed approach2.

TABLE VIII
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

We also evaluate the impact of co-occurence score on bug localization performance, with different α 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

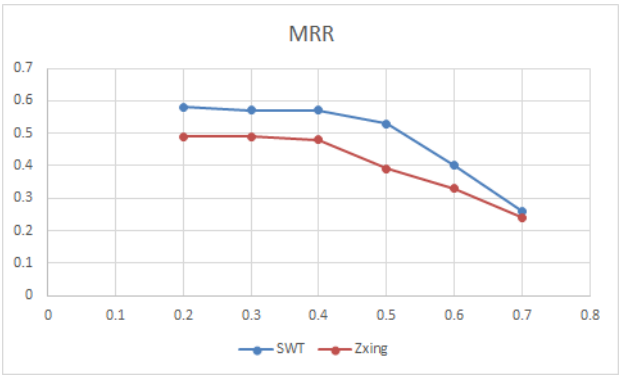

Fig. 5. The impact of α on bug localization performance (MRR)

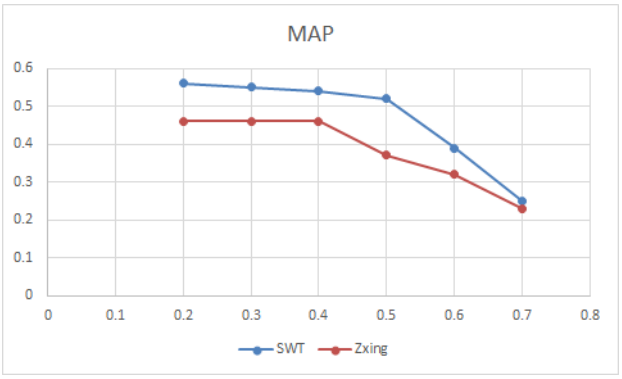

Fig. 6. The impact of α on bug localization performance (MAP)

of MRR and MAP) for the SWT and Zxing projects. When the α 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. As co-occurence score is based on the association map between keywords and source files, thus no direct matching of vocabulary is required. Therefore, the results obtained from the impact of α on bug localization performance (i.e., MAP and MRR) also suggest the vocabulary missmatch problem is resolved here.

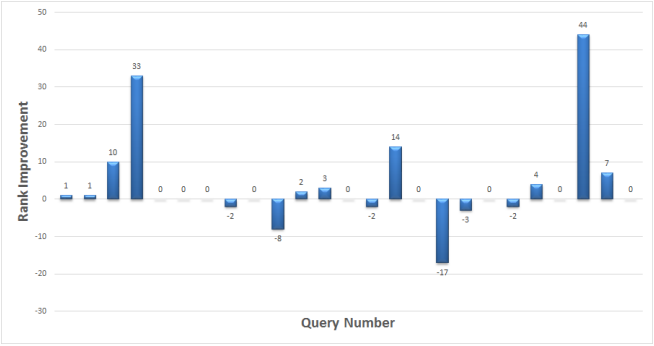

Fig. 7. Quey wise comparison between VSM and BLuAMIR for 30 Eclipse Query

Answering RQ3: We investigate how large files 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, we closely investigate the ranked results for several bugs.

case #1 Consider bug # 138283. The title of this bug is *[KeyBindings] Action set with accelerator causes conflict*. We have found 1 source code file (i.e., org.eclipse.ui.internal.WorkbenchPage.java) in rank #4 from the obtained results collected from BLuAMIR. No gold set files are resulted from VSM technique in Top-10. However, this source code file is obtained as rank # 20 in the ranked results collected by applying VSM approach. So, we go deeper into ranking score level. The VSM score for this file is 0.4935, which is pretty low than other files. Because of the length of this file is too large (i.e., contains more than 14K words) making the VSM score too low. On the other hand, in BLuAMIR, this file has Co-Occ score 0f 1.00, which make total score of 0.69, put this on rank #4. From this case study, it is clear that association mapping between fixed bug report keywords into their corresponding source files can overcome the noise associated with large files. Even if the query and recommended buggy files do not share same keyword but previously shared same concept can aid locating that query.

V. 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., [2, 8, 9, 17], model-based fault localization, e.g., [5, 12], 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 [7]. Therefore, FindBug does not even need a bug report. However, it often detects too many false positives and misses many real bugs [21]. 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 [19], the "probabilistic approach" known as BM25 [16], or more recent language modeling [14]. Another empirical study [4] show that all three approaches perform comparably when well-tuned. However, Rao and Kak [15] 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 [6], its probability model was found to be deficient. Lukins et al. [11] use Latent Dirichlet Allocation (LDA), which is a well-known topic modeling approach, to localize bug [11]. However, LSI is rarely used in practice today due to errors in induced concepts introducing more harm than good [6] and LDA is not be able to predict the appropriate topic because it followed a generative topic model in a probabilistic way [10].

Sisman and Kak [20] 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. [18] 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. [13] 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 [22] propose AmaLgam, a new method for locating relevant buggy files that put together version history, similar report, and structure, to achieve better performance. Later [23] 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.

VI. 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 [18, 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.

VII. 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]

[2] 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.

[3] A. Bachmann and A. Bernstein. Software Process Data Quality and Characteristics: A Historical View on Open and Closed Source Projects. In *Proc. IWPSE*, IWPSE-Evol '09, pages 119–128.

[4] 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.

[5] 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.

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

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

[8] 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.

[9] 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.

[10] 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.

[11] 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.

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

[13] 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.

[14] 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.

[15] 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.

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

[17] 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.

[18] 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.

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

[20] 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.

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

[22] 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.

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

[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.