

# 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 association between keywords from the previously resolved reports and their corresponding changed source documents. Experiments using a collection of 3973 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 12.6% in Top-5 accuracy over the state-of-the-art.

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

## I. INTRODUCTION

Bug localization is a process of locating the source code that needs to be changed in order to fix a given bug. Locating buggy files is not only time-consuming but also costly in terms of efforts [26]. 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 localize software bugs by considering relevance between a bug report and the source code in terms of their lexical similarity. Hence, these techniques are likely to be affected by the quality and content of a submitted query (i.e., bug report). If a query contains inadequate information, then the retrieved 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 reports and their corresponding changed source documents extracted from commit logs for the bug localization.

Zhou et al. [30] first propose BugLocator based on revised Vector Space Model (rVSM) that uses the lexical similarity between bug report texts and source code documents. Saha et al. [22] capture the structures of both bug reports and source documents, and then apply structured information retrieval for bug localization. Moreno et al. [17] consider structural similarity between stack traces in the bug report and the project document structures for bug localization. However, all of these studies share one common limitation.

Most of these IR-based bug localization techniques simply rely on only lexical similarity for retrieving relevant documents from the codebase. Hence, the search queries constructed from a given bug report should contain keywords similar to the code terms. Unfortunately, bug reports could be of low quality and could miss the appropriate keywords. Besides, an appropriate query also warrants for domain expertise on the project from a developer, which can always not be guaranteed.

To solve the problem with lexical similarity, Maletic and Marcus [13], Marcus and Maletic [15] extract semantic meaning of the documentation and source code by utilizing Latent Semantic Analysis (LSA). They use this information to identify traceability links based on similarity measures. To automatically recover traceability links from system documentation to program source code, Marcus and Maletic [14] proposed a method using Latent Semantic Indexing (LSI), and their experiments suggest that this technique is promising than VSM, and other probabilistic IR methods. LSI requires the use of a dimensionality reduction parameter that must be tuned for each document collection [9]. However, the results returned by LSI can be difficult to interpret, as they are expressed using a numeric spatial representation.

In this paper, we propose a bug localization approach namely BLuAMIR that not only considers lexical similarity between keywords and the source code but also captures implicit association between them from the past bug fixing history. In particular, we consider the implicit association between keywords extracted from previously fixed bug reports and their corresponding changed code. The idea is to capture their relationship without a direct vocabulary match. Thus, our proposed approach caters for the vocabulary mismatch problem using a keyword-source association constructed from fixed bug information. Our proposed technique also overcomes the issue with large files. Lexical similarity based approaches often suffer from the noise in the large documents during relevance estimation. 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 more effectively.

We evaluate our technique in three 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, and contrast with three replicated baseline, Latent Semantic Indexing (LSI) [14], Vector Space Model(VSM) and BugLocator [30]. BLuAMIR localizes bugs with in average 36.98%

higher accuracy (i.e., Hit@10), 63% higher precision (i.e., MAP@10), and 63.81% higher result ranks (i.e., MRR@10) than replicated baseline LSI (Section IV-G). BLuAMIR also localizes bugs with in average 9% higher accuracy (i.e., Hit@10), 12% higher precision (i.e., MAP@10) and 11% higher result ranks (i.e., MRR@10) than the replicated baseline VSM (Section IV-F). BLuAMIR localizes bugs with in average 57% higher accuracy (i.e., Hit@10), 81% higher precision (i.e., MAP@10) and 94% higher result ranks (i.e., MRR@10) than replicated BugLocator [30] (section IV-F). Second, we compare our technique with a state of the art approach [30] and our technique can improve 12% in MAP@10 and 12.6% in Top-5 accuracy over the state-of-the-art (Section IV-G). Third, in terms of query-wise improvement, BLuAMIR improves results rank of 41.05% and degrades 19.37% on Eclipse dataset than baseline VSM (Section IV-H).

Thus, this paper makes the following contributions:

- 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 two state-of-the-art approaches including BugLocator [30] and LSI [14].

The rest of the paper is organized as follows. Section II discusses a motivating example of our proposed approach, and Section III presents proposed bug localization method for BLuAMIR, and Section IV focuses on the conducted experiments and experimental results, and Section V identifies the possible threats to validity, and Section VI discusses the existing studies related to our research, and finally, Section VII concludes the paper with future plan.

## 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 bug report. We preprocess both fields, and construct a query. Then we apply both Vector Space Model (VSM) and our approach on this query, and locate possible buggy files. In Table I, we can see that basic VSM approach locates only 2 buggy files in Top-5 where, our proposed tool BLuAMIR correctly identifies 3 buggy files. The ranking from basic VSM is 1 and 4 where the rank from BLuAMIR is 1,2,3. However, 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. VSM score is based on overlap of terms found both in the bug reports and source files. On the contrary, association score do not directly compare terms between bug reports and source files. So vocabulary mismatch 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 mismatch problem, VSM failed to retrieve some 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 (i.e., query) and then retrieve their associated source files from the constructed association map. This association map is created from keywords extracted from previously fixed bug report and their fixed buggy codebase. So, the idea is, if a current bug shares same keywords (i.e., concepts) with a previous fixed bug, there is possibility they might share some codebase. This is how our proposed approach overcome vocabulary mismatch 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 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<sup>1</sup> during stop word removal and Porter Stemming stemmer<sup>2</sup> for stemming. First we create a mapping relationship between bug IDs and

<sup>1</sup><https://www.ranks.nl/stopwords>

<sup>2</sup><http://tartarus.org/martin/PorterStemmer/>

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	✗

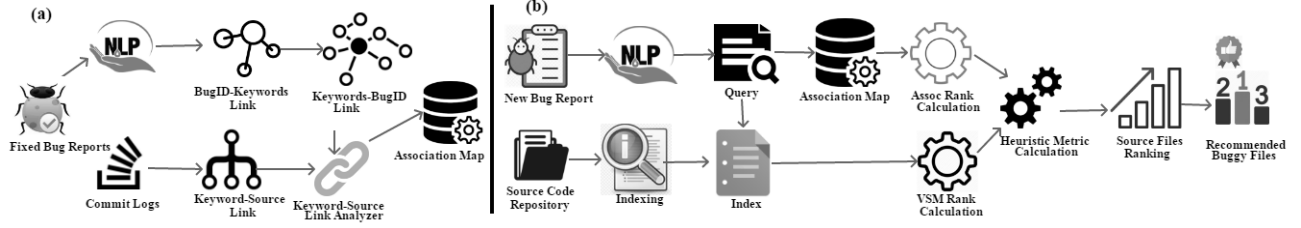


Fig. 1: Proposed Schematic Diagram: (a) Construction of association map between bug report keywords and source documents and (b) Bug Localization Using VSM and association score

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.

keywords contained in those bug reports, as described in Algorithm 1 line 3. Second we also construct another connected relationship between keywords and bug IDs, presented in line 4 of Algorithm 1.

**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 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 [2]. 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 [5]. We use several utility commands such as git, clone and log on GitHub for collecting those information. This step is done line 5 of Algorithm 1.

**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 (i.e., discussed bellows) between keywords collected from a bug report and buggy source document links. Here, one or more keywords can be

**Algorithm 1** Construction of Association Mapping Database

```

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

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 6-15 in Algorithm 1.

#### A Bipartite Graph Example:

In mathematics, a bipartite graph is a graph in which the vertices can be put into two separate groups so that the only edges are between those two groups, and there are no edges

TABLE III: A Bug Report (#322401, eclipse.UI.platform)

Field	Content
Title	[LinkedResources] Linked Resources properties page should have a Remove button
Description	I20100810-0800 Project properties > Resource > Linked Resources: Especially for invalid locations, a Remove button would be handy to remove one or multiple links.

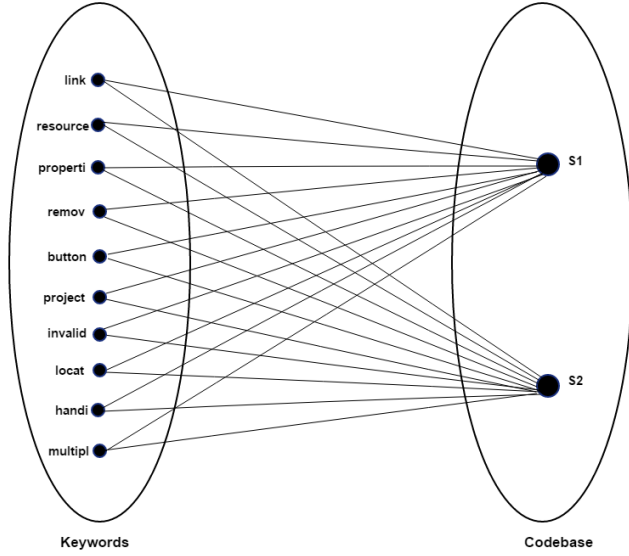


Fig. 2: A Bipartite Graph Constructed for Bug #46669.

between vertices within the same group. In our work, two groups correspond to keywords and codebase. Hence, neither two keywords nor two source files can be connected. There exist only one type of valid link, which can be established between a keyword and its corresponding sources. We construct a bipartite graph for a bug report (ID 322401) on the Eclipse UI Platform. Table III shows the *title* and *description* of the bug report. The bipartite graph is presented in Figure 2. Note, that ten unique keywords are extracted from the bug report (i.e., from title and description) and are connected through edges with their corresponding change codes (i.e.,  $S1$  represents *IDEWorkbenchMessages.java* and  $S2$  represents *LinkResourceEditor.java*).

### B. Bug Localization Using VSM and Association Scores

The system diagram for this part has illustrated in Figure 1(b), which is based on the combination of VSM and association scores (i.e., Figure 1(b)). We have constructed an association map database 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 two ranks and recommend buggy relevant source files. Algorithm 2 shows pseudo-code for our bug localization approach.

### Algorithm 2 Bug Localization using Association Mapping Database and Information Retrieval

```

1: procedure BUGLOCALIZATION( $Q, SC_{repo}, MAP_{KS}$ )
    ▷  $Q$ : a bug report query
    ▷  $SC_{repo}$ : a source code repository
    ▷  $MAP_{KS}$ : association map created from Algorithm 1
2:    $RankedR \leftarrow \{\}$  ▷ List of recommended buggy files
3:    $Index \leftarrow \text{createIndex}(SC_{repo})$ 
    ▷ Create Index from source code corpus
4:    $Score_{Assoc} \leftarrow \text{calculateAssociationScore}(Q, MAP_{KS})$ 
    ▷ Calculate Association score
5:    $Score_{VSM} \leftarrow \text{calculateVSMscore}(Q, Index)$ 
    ▷ Calculate VSM score
6:    $SC_{Assoc} \leftarrow \text{collectSCCodeWithAssocScore}(Score_{Assoc})$ 
    ▷ Collect Source Codes for which association score
    is computed
7:   for SourceCode  $SC_i \in SC_{Assoc}$  do
8:      $Rank_{Assoc_i} \leftarrow \text{retrieveAssocScore}(Score_{Assoc})$ 
    ▷ retrieve association score for each source file
9:      $Rank_{VSM_i} \leftarrow \text{retrieveVSMscore}(Score_{VSM})$ 
    ▷ retrieve VSM score for the same source file
10:     $CombineS[SC_i] \leftarrow Rank_{Assoc_i} + Rank_{VSM_i}$ 
    ▷ Combine both scores utilizing a heuristic metric
11:  end for
12:   $RankedR \leftarrow \text{retrieveTopRSF}(CombineS[SC_i])$ 
    ▷ Sort combine score and retrieve top ranked buggy
    files
13:  return  $RankedR$ 
14: end procedure

```

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:

**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:

$$AssociationScore = \sum_{All S_i \text{ that connect to } W_j} (Link(W_j, S_i)) \quad (1)$$

Here, the link  $Link(W_j, S_i)$  between keyword and source file is 1 if they are connected in the association map and 0 otherwise. This step is done in line 4 of Algorithm 2.

**Vector Space Model:** The vector space model (VSM)

is a classical method for constructing vector representations for documents [23]. It encodes a document collection by a term-by-document matrix. It represents one type of text unit (documents) by its association with the other type of text unit (terms). Here, the association is calculated by explicit evidence based on term occurrences in the documents. The similarity between documents is computed by the cosine or inner product between corresponding vectors.

**VSM Score Calculation:** 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) is that the similarity between documents is computed by the cosine or inner product between terms collected from a query and a source document. The weight of a term in a document increases with its occurrence frequency in this specific document and decreases with its occurrence frequency in other documents. In classic VSM,  $tf$  (i.e., term frequency) and  $idf$  (i.e., inverse document frequency) are defined as follows:

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

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

$$VSM Score(q, d) = \cos(q, d) = \frac{1}{\sqrt{\sum_{t \in q} ((\frac{f_{tq}}{\#terms}) \times \log(\frac{\#docs}{n_t}))^2}} \times \frac{1}{\sqrt{\sum_{t \in d} ((\frac{f_{td}}{\#terms}) \times \log(\frac{\#docs}{n_t}))^2}} \times \sum_{t \in q \cap d} (\frac{f_{tq}}{\#terms}) \times (\frac{f_{td}}{\#terms}) \times \log(\frac{\#docs}{n_t})^2 \quad (3)$$

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. This score is calculated in line 5 of Algorithm 2,

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

$$FinalScore = (1 - \alpha) \times N(VSM Score) + \alpha \times N(AssociationScore) \quad (4)$$

Here, the weighting factor  $\alpha$  varies from 0.1 to 0.5, for which we discuss results in the experiment section IV-H.

We combine both scores using equation 4 in Algorithm 2 line 7-11. Finally, top ranked recommended buggy files are retrieved, which is done line 12 of Algorithm 2.

## 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 and Zxing. We work with Eclipse data set which is a popular IDE for Java. We downloaded a git based Eclipse project from git repository<sup>3</sup>. 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 queries from each bugs considering their title and short summary. All bug reports are collected from<sup>4</sup>. 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. [30] 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 IV. SWT is a component of Eclipse<sup>5</sup>. Zxing is an android based project maintained by google<sup>6</sup>.

TABLE IV: 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
Eclipse (v3.1)	An open development platform for Java	Oct 2004 - Mar 2011	3071	11831
SWT (V 3.1)	An open source widget toolkit for Java	Oct 2004 - Apr 2010	98	484
AspectJ	Aspect-oriented extension to Java	July 2002 - Oct 2006	244	3519
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

<sup>3</sup><https://git.eclipse.org/c/platform/eclipse.platform.ui.git/>

<sup>4</sup><https://bugs.eclipse.org/>

<sup>5</sup><http://www.eclipse.org/swt/>

<sup>6</sup><http://code.google.com/p/zxing/>

correct answer. So mean reciprocal rank is the average of the reciprocal ranks of results of a set of queries  $Q$

$$MRR = \frac{1}{|Q|} \sum_{i=1}^{|Q|} \frac{1}{rank_i} \quad (5)$$

where  $rank_i$  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:

$$AP = \sum_{k=1}^M \frac{P(k) \times pos(k)}{\text{number of positive instances}} \quad (6)$$

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

$$P(k) = \frac{\#buggy\ Files}{k} \quad (7)$$

### 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 mismatch 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 replicated BugLocator [30] and VSM based bug localization techniques. The detail of this comparison is presented in the following subsection. On the other hand, we also experiment with SWT and Zxing dataset as in [30]. Here we collect the results reported in [30] and then compare the performance with BLuAMIR. We also answer our research questions in the following subsections.

### E. Comparison Between Replicated BugLocator and BLuAMIR

First, we replicate an existing technique (i.e., BugLocator) proposed by Zhou et al. [30]. Zhou et al. [30] uses revised Vector Space Model (rVSM) and Simi scores in order to index and rank source code files. Second, we combine our proposed association score with these two scores ranks (i.e., rVSM, Simi). Finally, we perform comparison between replicated BugLocator with this combined scores in terms of Top-1, Top-5, Top-10, MRR and MAP. Ranking based on rVSM and Simi and our combined approach are depicted as follows.

**Ranking based on Revised Vector Space Model:** 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 [30] for calculating term frequency is:

$$tf(t, d) = \log(f_{td}) + 1 \quad (8)$$

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

$$\begin{aligned} rVSM Score(q, d) &= g(\#term) \times \cos(q, d) \\ &= \frac{1}{1 + e^{-N(\#terms)}} \times \frac{1}{\sqrt{\sum_{t \in q} ((\log f_{tq} + 1) \times \log(\frac{\#docs}{n_t}))^2}} \times \\ &\quad \frac{1}{\sqrt{\sum_{t \in d} ((\log f_{td} + 1) \times \log(\frac{\#docs}{n_t}))^2}} \times \\ &\quad \sum_{t \in q \cap d} (\log f_{tq} + 1) \times (\log f_{td} + 1) \times \log(\frac{\#docs}{n_t})^2 \quad (9) \end{aligned}$$

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 is 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.. Here, we construct a 3-layer architecture as described in [30]. 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. For all source code files in layer 3, similarity score

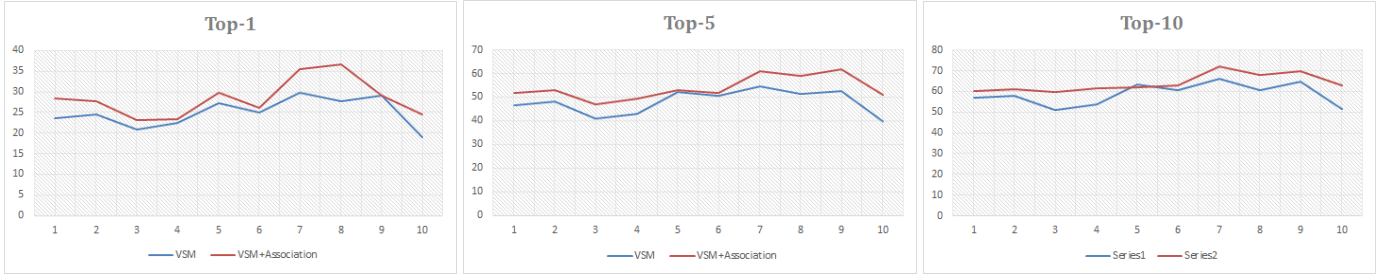


Fig. 3: Performance Comparison between VSM and VSM+Association for Top-1, Top-5 and Top-10 Retrieval

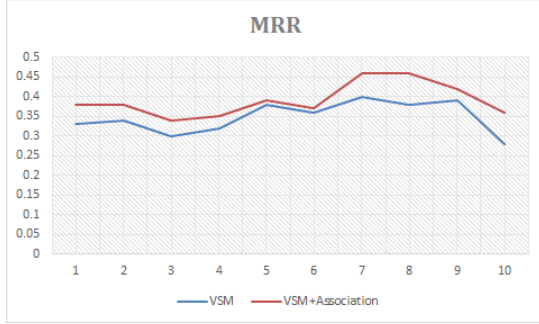


Fig. 4: MRR Comparison between VSM and VSM+Association

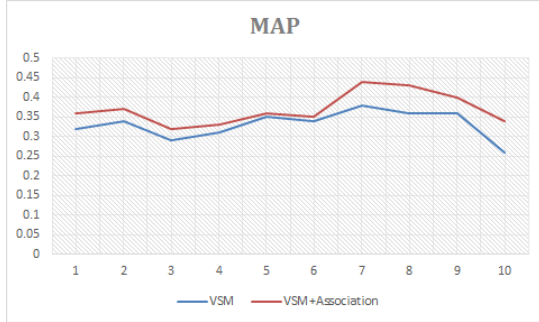


Fig. 5: MAP Comparison between VSM and VSM+Association

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

$$SimiScore = \sum_{All S_i \text{ that connect to } F_j} (Similarity(B, S_i) / n_i) \quad (10)$$

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 9 and we also calculate Simi score using equation 10. Then we calculate association scores for the query using equation 1. We finally combine the three ranks and for that

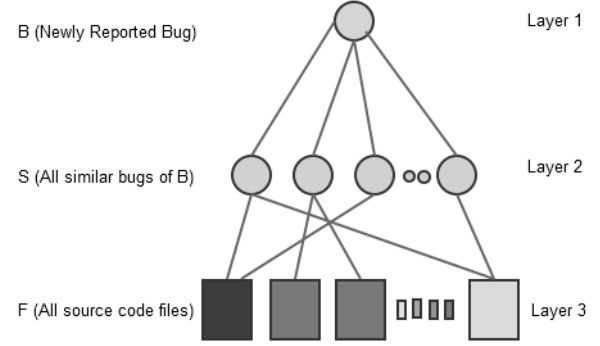


Fig. 6: Bug and its Similar Bug Relationship with Source Code Files:

we use three weighting factor  $\alpha$ ,  $\beta$  and  $\gamma$ . The final equation is given in equation 11.

$$CombinedScore = \gamma \times N(rVSMscore) + \beta \times N(SimiScore) + \alpha \times N(AssociationScore) \quad (11)$$

We work with different values of  $\alpha$ , which are presented in the experiment section IV-H. We use value of 0.2 for  $\beta$ , varying  $\alpha$  from 0.1 to 0.5 and thus,  $\gamma$  from 0.7 to 0.3. So that they end up into 1.

#### F. Answering RQ1

To answer RQ1, we compare the performance of our proposed bug localization approach with two existing techniques - 1) VSM which is based on vector space model and 2) replicated BugLocaotor [30] which is based on rVSM and Simi scores.

**VSM vs Our proposed Tool** We combine VSM score and association 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 V. We compute Top-1, Top-5, Top-10 performance (presented in Fig 3) and MRR and MAP (shown in Fig 4 and Fig 5 respectively) 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%



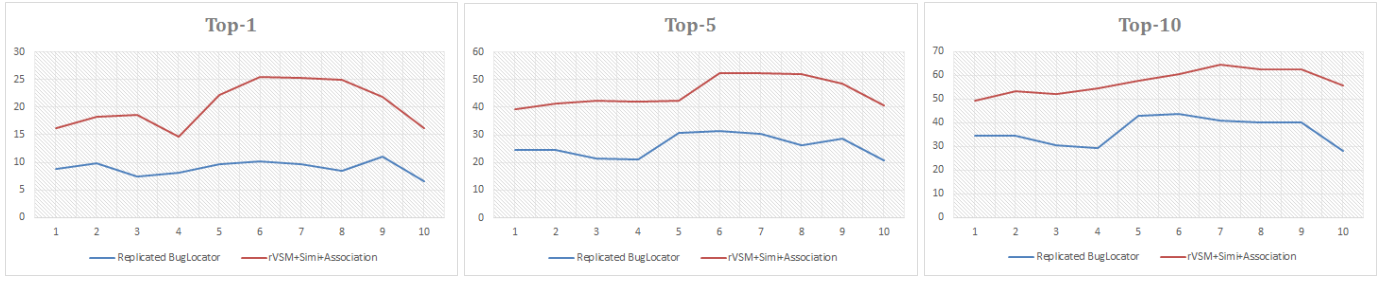


Fig. 7: Performance Comparison between Replicated BugLocator and rVSM+Simi+Association for Top-1, Top-5 and Top-10 Retrieval

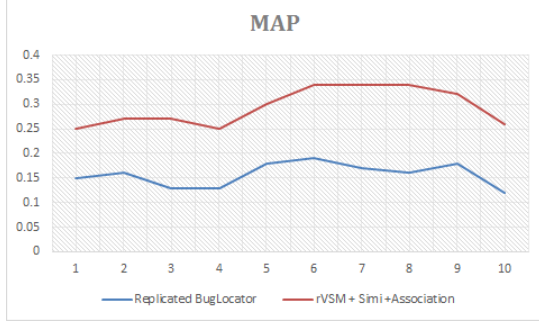


Fig. 8: MRR Comparison between Replicated BugLocator and rVSM+Simi+Association

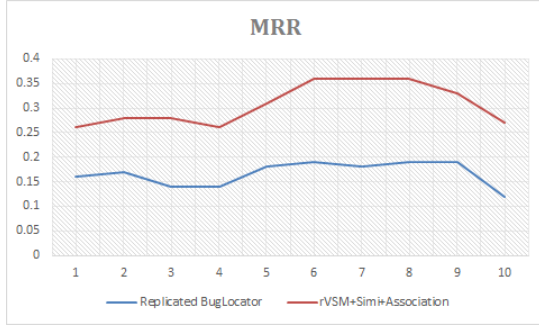


Fig. 9: MAP Comparison between Replicated BugLocator and rVSM+Simi+Association

whereas it is 58.73% for VSM. So, to answer our *RQ1*, we can go to Table V, 64.05% bugs are successfully located in Top-10 for Eclipse dataset for Approach I of BLuAMIR.

TABLE V: Performance of proposed technique (VSM+Association) Ranks

#Bugs	#Methodology	Top 1 %	Top 5 %	Top 10 %	MRR@10	MAP@10
3855	Basic VSM	24.98%	47.94%	58.73%	0.35	0.33
	VSM + Association	28.43%	53.86%	64.05%	0.39	0.37

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  $p_i=0.05$ . The W-value is 0. The critical value of W for N = 10 at  $p_i=0.05$  is 8. Therefore, the result is significant at  $p_i=0.05$ . For MAP - the Z -value is

-2.8031. The p -value is 0.00512. The result is significant at  $p_i=0.05$ . The W-value is 0. The critical value of W for N = 10 at  $p_i=0.05$  is 8. Therefore, the result is significant at  $p_i=0.05$ .

**Replicated BugLocator VS Our Proposed Tool:** We combine rVSM and simi ranks with our association rank. Here, association rank is computed based on keyword-source code mapping database. We compare the performance of our proposed approach in terms of top 1, 5, 10 rank (depicted in Fig 7) and MRR and MAP (shown in Fig 8 and Fig 9 respectively). 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*.

TABLE VI: Performance of Buglocator and proposed technique (rVSM+Simi+Association)

#Bugs	Methodology	Top 1 %	Top 5 %	Top 10 %	MRR	MAP
3855	Replicated BugLocator	8.99%	25.87%	36.40%	0.16	0.16
	rVSM+Simi+ Association	20.39%	45.32%	57.28%	0.31	0.29

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  $p_i=0.05$ . The W-value is 0. The critical value of W for N = 10 at  $p_i=0.05$  is 8. Therefore, the result is significant at  $p_i=0.05$ . For MAP - the Z -value is -2.8031. The p -value is 0.00512. The result is significant at  $p_i=0.05$ . The W-value is 0. The critical value of W for N = 10 at  $p_i=0.05$  is 8. Therefore, the result is significant at  $p_i=0.05$ .

#### G. Answering *RQ4*

To answer *RQ4*, we compare the performance of BLuAMIR with two state-of-the-art techniques - LSI [14] and BugLocator [30] for the the same dataset (i.e., SWT and Zxing depicted in Table VII). We replicated the LSI technique for bug localization, which is proposed by Marcus and Maletic [14] for traceability links. For Buglocator [30], we copied the results directly from their paper. We also collect the same bug reports and the same source code repository for both of them. So the results can be compared. Our tool BLuAMIR outperforms LSI [14] for both SWT and Zxing dataset in all five performance metrics. 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 VII: Performance Comparison between LSI, BugLocator and BLuAMIR

# System	#Localization Approach	Top 1 %	Top 5 %	Top 10 %	MRR@10	MAP@10
Eclipse	LSI					
	BugScout	14.00	24.00	31.00		
	BugLocator	24.36	46.15	55.90	0.35	0.26
	BLUiR	30.96	53.20	62.86	0.42	0.32
	BLuAMIR	27.42	52.88	62.55	0.38	0.27
AspectJ	LSI					
	BugScout	11.00	26.00	35.00		
	BugLocator	22.73	40.91	55.59	0.33	0.17
	BLUiR	32.17	51.05	60.49	0.41	0.24
	BLuAMIR	32.79	56.56	68.03	0.43	0.23
SWT	LSI	13.26	30.61	53.06	0.15	0.14
	BugLocator	31.63	65.31	77.55	0.47	0.40
	BLUiR	55.10	76.53	87.76	0.65	0.56
	BLuAMIR	48.95	72.92	82.29	0.59	0.51
	LSI	15.00	35.00	45.00	0.23	0.23
Zxing	BugLocator	40.00	55.00	70.00	0.48	0.41
	BLUiR	40.00	65.00	70.00	0.49	0.38
	BLuAMIR	50.00	80.00	85.00	0.64	0.58

#### H. Answering RQ2

To answer RQ2, we investigate several weighting functions for our proposed approach, which are described as follows:

**Weighting Function for VSM+Association Ranking:** We 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 VIII. Here, it shows, more  $\alpha$  produces better performance. That means if we increase the association scores with higher weighting function, the better performance is resulted in this proposed approach. Adding association 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  $\alpha$  for Top-1, Top-5 and Top-10 retrieval on Eclipse dataset for approach I. (VSM+Co-occurrence Ranking) in Figure 10.

TABLE VIII: Performance of (VSM+Association) for different weighting factors

$\alpha$	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

We also evaluate the impact of association score on bug localization performance, with different  $\alpha$  values in terms of MAP and MRR for SWT and Zxing. At the beginning, the bug localization performance increases when the  $\alpha$  value increases. However, after a certain point, further increase of the  $\alpha$  value will decrease the performance. For example, Figure 11 and 12 show the bug localization performance (measured in terms of MRR and MAP) for the SWT and Zxing projects. When the  $\alpha$  value increases from 0.1 to 0.4, both MRR and MAP

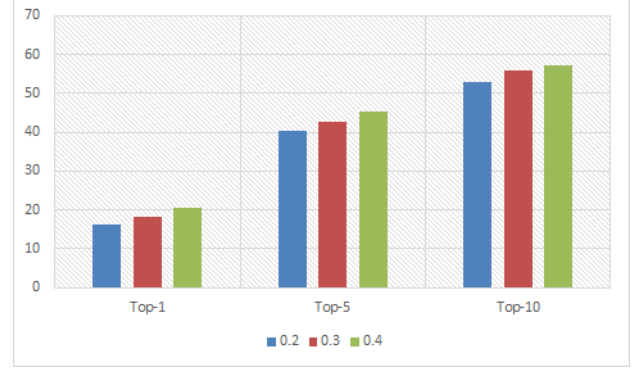


Fig. 10: The impact of  $\alpha$  on bug localization performance (Top-1, Top-5, Top-10)

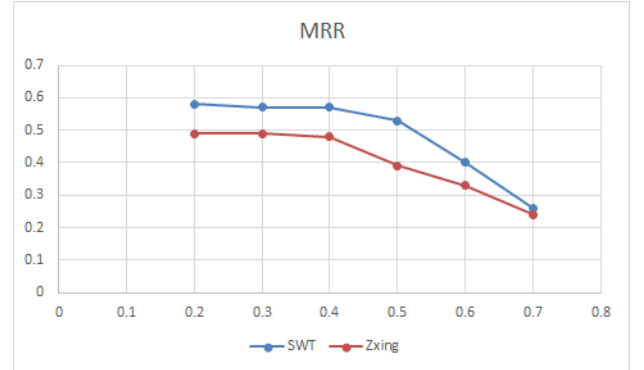


Fig. 11: The impact of  $\alpha$  on bug localization performance (MRR)

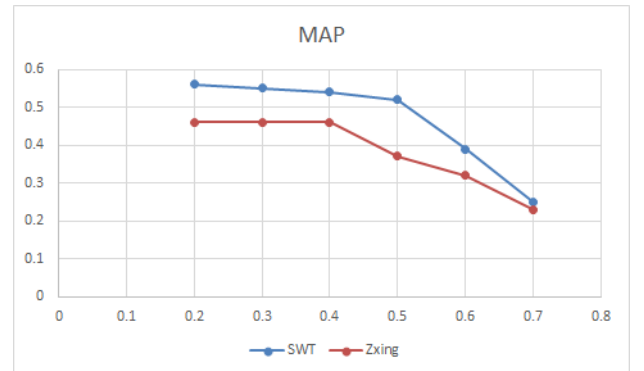


Fig. 12: The impact of  $\alpha$  on bug localization performance (MAP)

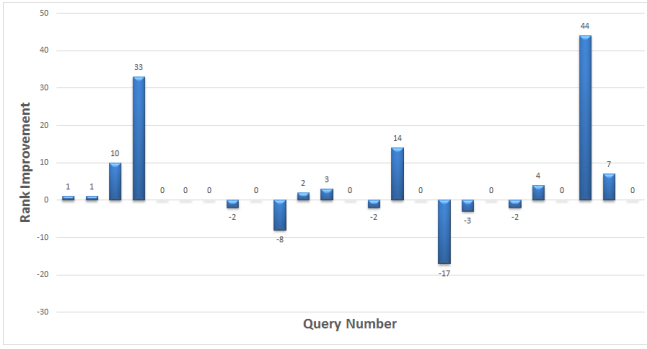


Fig. 13: Query wise comparison between baseline VSM and BLuAMIR for 30 Eclipse Query

values increases. Increasing  $\alpha$  value further from 0.4 to 0.7 however leads to lower performance. Note that we obtain the best bug localization performance when  $\alpha$  is between 0.3 and 0.4. As association 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  $\alpha$  on bug localization performance (i.e., MAP and MRR) also suggest the vocabulary mismatch problem is resolved here.

**Answering RQ3:** We investigate how large files problem is eliminated in BLuAMIR. First we perform a query-wise ranking comparison between baseline basic VSM and BLuAMIR on Eclipse dataset. The results are shown in Table IX. BLuAMIR improves 41.05% with having a mean of 13 and worsen 19.37% with a mean of 38.10 over baseline VSM.

Therefore, it is proven that BLuAMIR is showing improvement over baseline VSM and hence, we select a collection of Eclipse queries for a depth analysis. Second, we perform a query wise ranking comparison for BLuAMIR on 30 queries from Eclipse system, which is given in Figure 13. Here, X-axis represents query number and Y-axis represents the difference of best rank retrieved by VSM and 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 ranking retrieval. As most cases BLuAMIR 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 association score of 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. 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. [30], 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 [22, 30] 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  $\alpha$ ,  $\beta$ , and  $\gamma$  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.

## VI. 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

TABLE IX: Query-wise Rank Comparison on Eclipse Dataset

Technique	Total	#Improved	Mean	Min.	Max.	#Worsened	Mean	Min.	Max.	# Preserved
BLuAMIR	3072	1261 (41.05%)	13	1	88	595 (19.37%)	38.10	1	917	1216 (39.58%)

program elements sorted by their suspiciousness. Some of the well known dynamic approaches are spectrum-based fault localization, e.g., [1, 8, 10, 21], model-based fault localization, e.g., [4, 16], dynamic slicing [29], delta debugging [28].

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 [25]. 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 [23], the “probabilistic approach” known as BM25 [20], or more recent language modeling [18]. Another empirical study [3] show that all three approaches perform comparably when well-tuned. However, Rao and Kak [19] 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. [12] use Latent Dirichlet Allocation (LDA), which is a well-known topic modeling approach, to localize bug [12]. 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 [11].

Sisman and Kak [24] propose a history-aware IR-based bug localization solution to achieve a better result. Zhou et al. [30] propose BugLocator, which leverages similarities among bug reports and uses refined vector space model to perform bug localization. Saha et al. [22] 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. [17] 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 [26] propose AmaLgam, a new method for locating relevant buggy files that put together version history, similar report, and structure, to achieve better performance. Later [27] 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.

## 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] R. Abreu, P. Zoetewij, 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] 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.
- [3] 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.
- [4] 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.
- [5] S. Haiduc, G. Bavota, R. Oliveto, A. Marcus, and A. De Lucia. Evaluating the Specificity of Text Retrieval Queries to Support Software Engineering Tasks. In *Proc. ICSE*, pages 1273–1276, 2012.
- [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] A. Kontostathis. Essential Dimensions of Latent Semantic Indexing (lsi). In *2007 40th Annual Hawaii International Conference on System Sciences (HICSS'07)*, 2007.
- [10] 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.
- [11] S.K. Lukins, N.A. Kraft, and L.H. Etzkorn. Source Code Retrieval for Bug Localization Using Latent Dirichlet Allocation. In *Reverse Engineering, 2008. WCRC '08. 15th Working Conference on*, pages 155–164, Oct 2008.

- [12] 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.
- [13] J. I. Maletic and A. Marcus. Supporting Program Comprehension Using Semantic and Structural Information. In *Proceedings of the 23rd International Conference on Software Engineering. ICSE 2001*, pages 103–112, 2001.
- [14] A. Marcus and J. I. Maletic. Recovering Documentation-to-Source-Code Traceability Links Using Latent Semantic Indexing. In *Proceedings of the 25th International Conference on Software Engineering, ICSE '03*, pages 125–135.
- [15] A. Marcus and J. I. Maletic. Identification of High-level Concept Clones in Source Code. In *Proceedings 16th Annual International Conference on Automated Software Engineering (ASE 2001)*, pages 107–114, Nov 2001.
- [16] W. Mayer and M. Stumptner. Model-based Debugging – State of the Art And Future Challenges. *Electron. Notes Theor. Comput. Sci.*, 174(4):61–82.
- [17] 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.
- [18] 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.
- [19] 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.
- [20] S. E. Robertson, S. Walker, and M. Beaulieu. Experimentation As a Way of Life: Okapi at Trec. *Inf. Process. Manage.*, 36(1):95–108.
- [21] 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.
- [22] 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.
- [23] G. Salton and C. Buckley. Term-weighting Approaches in Automatic Text Retrieval. *Inf. Process. Manage.*, 24(5):513–523.
- [24] 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.
- [25] H. Tang, S. Tan, and X. Cheng. A Survey on Sentiment Detection of Reviews. *Expert Systems with Applications*, 36(7), 2009.
- [26] 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.
- [27] Shaowei Wang and David Lo. Amalgam+: Composing Rich Information Sources for Accurate Bug Localization. *Journal of Software: Evolution and Process*, 28: 921–942, 2016.
- [28] A. Zeller and R. Hildebrandt. Simplifying and Isolating Failure-Inducing Input. *IEEE Trans. Softw. Eng.*, 28(2):183–200.
- [29] 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.
- [30] 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.