# 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 bug reports and their corresponding changed source documents. Experiments using a collection of 3,431 bug reports show that on average our technique can locate buggy files with a Top-10 accuracy of 74.06%, a mean reciprocal rank@10 of 0.52 and a mean average precision@10 of 41% which are highly promising. Comparison with the state-of-the-art techniques and their variants report that our technique can improve upon them by 32.26% in MAP@10 and 26.74% in Top-5 accuracy.**

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

## I. Introduction

Bug localization is a process of locating such source code that needs to be changed in order to fix a given bug. Manually locating buggy files is not only time-consuming but also prohibitively costly in terms of development efforts [35]. This is even more challenging for the large software systems. Thus, effective, automated approaches are highly warranted for localizing the software bugs. Traditional Information Retrieval (IR) based bug localization techniques [28, 39] accept a bug report and a subject system as inputs and then return a list of buggy entities (e.g., classes, methods) against the bug report. They localize the bugs by simply relying on the *lexical similarity* between a bug report and the source code. Hence, they are likely to be affected by the quality and content of a submitted query (i.e., bug report). That is, if a query does not contain adequate information, then the retrieved results might not be relevant at all. As existing findings [24, 34] suggest, bug reports could be of low quality and could miss the appropriate keywords. Thus, lexical similarity alone might not be sufficient enough to solve the bug localization problem.

In order to address the limitations with lexical similarity, several existing studies [16, 18, 19] derive underlying semantics of a text document by employing Latent Semantic Analysis (LSA). Marcus et al. and colleagues adopt this technology in the context of concept location [18, 19], program comprehension [16] and traceability link recovery problems [17], and reported higher performance than traditional Vector Space Model (VSM) and probabilistic models. Unfortunately, their approach suffers from a major limitation. Latent Semantic Indexing (LSI) requires the use of a dimensionality reduction parameter that must be tuned for each document collection [11]. The results returned by LSI can also be difficult to interpret, as they are expressed using a numeric spatial representation. Other related studies [13, 22] adopt Latent Dirichlet Allocation (LDA) for bug localization. However, they are also subject to their hyper-parameters and could even be outperformed by simpler models (e.g., rVSM [39]).

In this paper, we propose a bug localization approach namely BLuAMIR that not only considers *lexical similarity* between a bug report (the query) and the source code but also captures *implicit association* between them from the bug fixing history. First, we determine the lexical similarity between each source document and the query using Vector Space Model (VSM) [39]. Second, we construct association maps between keywords of previously fixed bug reports and their corresponding changed documents using a bipartite graph. Third, we prioritize such source documents that are associated with the keywords (from query at hand) in these maps. Then, we rank the source documents based on their *lexical* and *association scores*. Thus, our approach caters for the vocabulary mismatch between a bug report (the query) and the source code with implicit association. That is, unlike traditional IR-based approaches [28, 39], it could return the buggy documents even if the query does not lexically match with the source code documents. Our approach also does not require the dimensionality reduction since we use a finite graph rather than a large sparse term-document matrix [17, 18].

We evaluate our technique in three different aspects using three widely used performance metrics and 3,431 bug reports (i.e., queries) from four open source subject systems. First, we evaluate in terms of the performance metrics, and contrast with two replicated baselines – Latent Semantic Indexing (LSI) [17] and basic Vector Space Model (VSM) [31]. BLuAMIR localizes bugs with 9%–37% higher accuracy (i.e., Hit@10), 12%–63% higher precision (i.e., MAP), and 11%–64% higher reciprocal ranks (i.e., MRR) than these baselines (Sections IV-F, IV-E). Second, we compare our technique with three state of the art approaches – BugScout [22], BugLocator [39] and BLUiR [28] (Section IV-F). Our technique can localize bugs with 6%–54% higher accuracy (i.e., Hit@5), 4%–32% higher precision (i.e., MAP) and 8%–27% higher reciprocal ranks (i.e., MRR) than these state-of-the-art approaches. Third, in terms of query-wise improvement, BLuAMIR improves result ranks of 41% more and degrades 19% less queries than baseline VSM (Section IV-G) with Eclipse system.

Thus, this paper makes the following contributions:

- A novel technique that not only considers the lexical

TABLE I
A WORKING EXAMPLE OF BLUAMIR

| | VSM: Lexical Similarity Only | | | BLuAMIR: Lexical Similarity + Implicit Association | | | | |
|---|---|---|---|---|---|---|---|---|
| **Rank** | **Retrieved Documents** | **$Score_{VSM}$** | **Ground Truth** | **Retrieved Documents** | $Score_{VSM}$ | $Score_{Assoc}$ | $Score_{Total}$ | **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<br>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. |

similarity between a bug report and the source code but also exploits their implicit associations through bug-fixing history for bug localization.
- Comprehensive evaluation of the technique using three widely used performance metrics and a total of 3,431 bug reports from four subject systems –Eclipse, SWT, AspectJ and ZXing.
- Comparison with not only two baselines [17, 31] but also three state-of-the-art approaches – BugScout [22], BugLocator [39] and BLUiR [28] with statistical tests.
- Experimental meta data and our used dataset for replication and third party reuse.

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 missmatch 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 missmatch problem.

## III. BLUAMIR: PROPOSED APPROACH FOR BUG LOCALIZATION

Figure 1 shows the schematic diagram of our proposed approach. First, we (a) construct an association map between the bug reports and their corresponding changed source documents with the help of bug-fixing history. Then we (b) retrieve buggy source code documents for a given query (bug report) by leveraging not only their lexical similarity but also their implicit associations derived from the association map above. We discuss different parts of our proposed approach – BLuAMIR – in the following sections.

### A. Construction of Association Map

We construct an association map between keywords from previously fixed bug reports and their corresponding changed source documents. The map construction involves three steps. We not only show different steps of our map construction in Fig. 1-(a) but also provide the relevant pseudo-code in Algorithm 1. We discuss each of these steps as follows:

**Keyword Extraction from Bug Reports:** First, we collect *title* and *description* of each bug report extracted from a subject system. Then we perform standard natural language preprocessing, and remove punctuation marks, stop words and small words from them. 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. Finally,

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

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

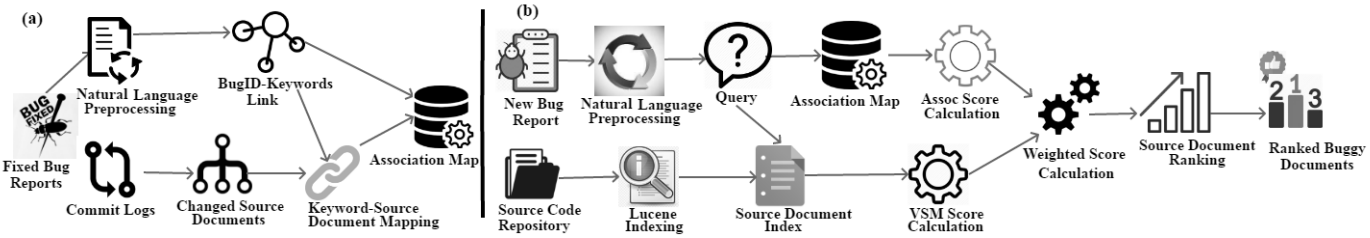


Fig. 1. Proposed Schematic Diagram: (a) Construction of association map between bug report keywords and source documents and (b) Bug Localization Using VSM and association score

**Algorithm 1** Construction of Association Map

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

we select a list of remaining keywords from each bug report. So, we create a mapping relationship between bug IDs and keywords contained in those bug reports, as described in Algorithm 1 line 5. We also construct another connected relationship between keywords and bug IDs, presented in line 6 of Algorithm 1.

**Source Code Link Extraction from Commit Logs:** Second, we consult with version control history of each system, and identify the bug-fixing commits using appropriate regular expressions [3, 36]. 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 [7]. We use several utility commands such as git, clone and log on GitHub for collecting those information. This step is done line 7 of Algotrithm 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. It should be noted each bug-fixing commit establishes an inherent relationship between its changed documents and the target bug report. Thus, the keywords from this bug report enjoy an implicit relationships with the changed source code documents. Third, we leverage such implicit relationships and construct the bipartite graph (i.e., discussed bellows) by explicitly connecting keywords from each bug report and the corresponding changed source documents. 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 8-15 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 between vertices within the same group.

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<br>Project properties > Resource > Linked Resources: Especially for invalid locations, a Remove button would be handy to remove one or multiple links. |

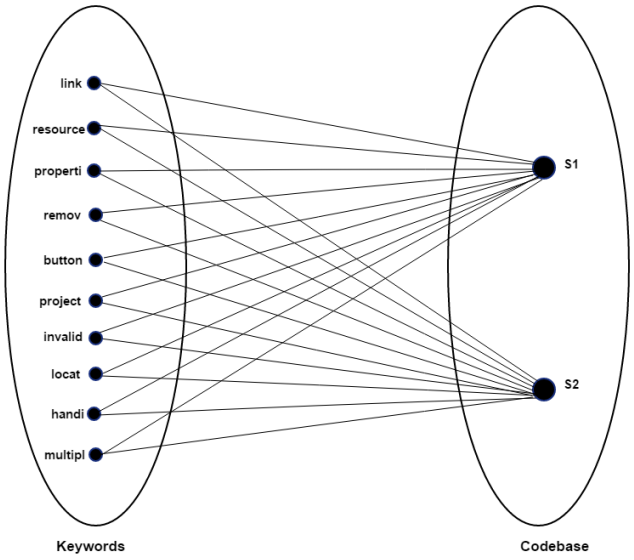


Fig. 2. A Bipartite Graph Constructed for Bug #46669.

In our work, two groups correspond to keywords and codebase. Hence, nether 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 *LinkedResourceEditor.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 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 two ranks and recommend buggy relevant source files. Algorithm 2 shows pseudo-code for our bug localization approach.

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

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

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

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:

$$AssociationScore = \sum_{AllS_ithatconnecttoW_j} (Link(Wj, S_i)) \quad (1)$$

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. 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 [29]. 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 consid-

ered 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 $f_{\text{td}}$ is the number of times term $t$ appears in document $d$. So the equation of classical VSM model is as follows

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

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, $n_{\text{t}}$ 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 Appache Lucene library. Then we combine that score with our association using equation 4.

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

Here, the weighting factor $\alpha$ varies from 0.2 to 0.4, for which we discuss results in the experiment section IV-G.

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 four different dataset - Eclipse, AspectJ, SWT aand Zxing. In order to evaluate our proposed tool we have used the same four dataset that Zhou et al. [39] and Saha et al. [28] used to evaluate BugLocator and BLUiR respectively. This dataset contains 3479 bug reports in total from four popular open source projects–Eclipse, SWT, AspectJ and ZXing along with the information of fixed files for those bugs. The detail of our dataset is presented in IV. Eclipse[3] is a well-known large-scale open source system and it is widely used in empirical software engineering research. SWT[4] is a component of Eclipse. The AspectJ[5] project is a part of the iBUGs public dataset provided by the University of Saarland. Zxing[6] is an android based project maintained by google. We create quires from each bugs considering their title and short summary (i.e., description). 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. Then we construct the association map between keywords extracted from bug reports and their source code links. During creating the map we have noticed some bug reports do not contain their fixed source files in codebase. So, we discarded those bug reports which yield total 3431 bug reports as depicted in table IV.

TABLE IV
Description of Data Sets

| Project Name | Description | Study Period | #Fixed Bugs | #Source Files |
|---|---|---|---|---|
| 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 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.

**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 $Q$

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

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

[3]https://bugs.eclipse.org/
[4]http://www.eclipse.org/swt/
[5]http://www.st.cs.uni-saarland.de/ibugs/
[6]http://code.google.com/p/zxing/

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:

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

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: (a) How does our proposed approach-BLuAMIR perform in Bug Localization? (b) How does our proposed association score affect its performance?
- 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: Is 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.

First, we compare the performance of BLuAMIR with replicated basic VSM based bug localization technique for Eclipse dataset. We also compare the performance between replicated LSI Marcus and Maletic [17] and BLuAMIR for AspectJ, SWT and ZXing datasets. The detail of this comparison is presented in the following subsection. Second, we compare the performance of BLuAMIR with three state of the arts techniques, BugScout [22], BugLocator [39] and BLUiR [28] for the same dataset as in [39]. Here we collect the results reported by the authors and then compare the performance with 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 re-implemented existing techniques - 1) classic VSM which is based on vector space model and 2) replicated LSI [17] which is based on latent semantic indexing.

TABLE V
PERFORMANCE COMPARISON OF REPLICATED BASIC VSM AND PROPOSED TECHNIQUE (BLUAMIR)

| #Bugs | #Methodology | Top 1 % | Top 5 % | Top 10 % | MRR@10 | MAP@10 |
|---|---|---|---|---|---|---|
| 3071 | Basic VSM | 23.09% | 47.54% | 57.57% | 0.33 | 0.24 |
| | VSM + Association | 27.45% | 53.79% | 63.30% | 0.38 | 0.27 |

**VSM vs BLuAMIR** In our proposed approach, we combine VSM score and association scores in order to produce a ranked result. We compare the performance of baseline VSM technique and our proposed combined approach on Eclipse dataset (i.e., 3071 bug reports). This performance comparison can assist us to investigate whether adding association score with VSM score can improve the bug localization performance or not. The comparison is presented in table V. We also use a 10-fold cross validation i.e., divide all 3071 bug reports into-10 fold and perform train on total 9 folds and test on the rest fold. For 10-fold validated Eclipse codebase, 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 63.30% whereas it is 58.26% for VSM. So. to answer our *RQ1(a)*, we can go to Table V, 63.30% bugs are successfully located in Top-10 for Eclipse dataset by BLuAMIR. This result also suggest that association score improves the retrieval performance of BLuAMIR (*RQ1(b)*).

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 5. 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 5. Therefore, the result is significant at p¡=0.05.

**Replicated LSI vs BLuAMIR:** Latent Semantic Indexing(LSI) is a VSM based method for indexing and representing aspects of the meanings of words, which reflects their usages. One of the problem with basic VSM that the user usually wants to retrieve on the basis of conceptual content. Most of the time individual words provide unreliable evidence about the conceptual topic or meaning of a document [4]. Therefore, to solve this problem [16, 18] utilize an advanced information retrieval technique (i.e., LSI) to extract the meaning of the documentation and source code. Based on any chosen similarity measure, they use this information to identify traceability

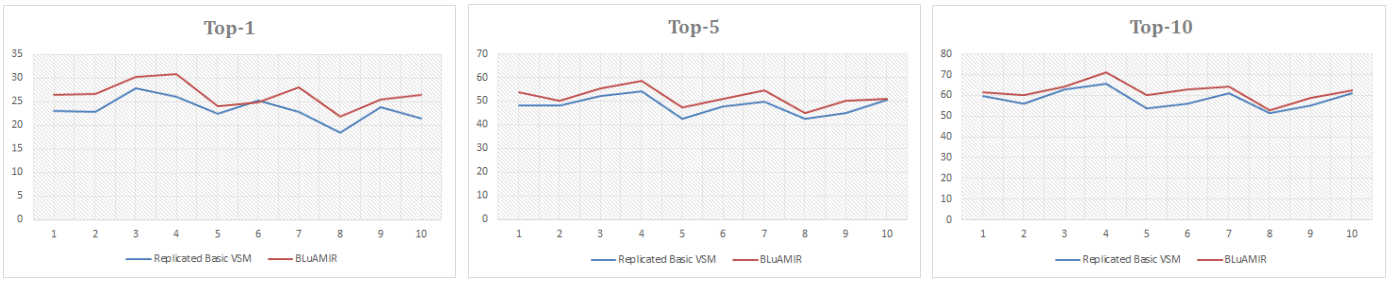


Fig. 3. Performance Comparison between replicated basic VSM and BLuAMIR for Top-1, Top-5 and Top-10 Retrieval

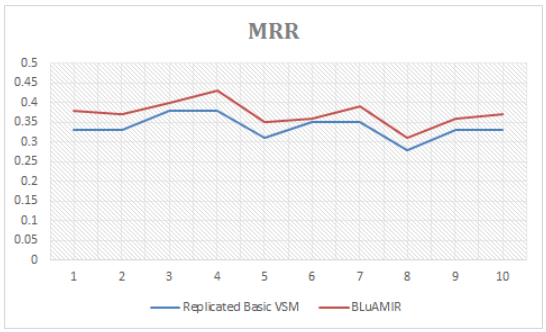


Fig. 4. MRR Comparison between replicated basic VSM and BLuAMIR

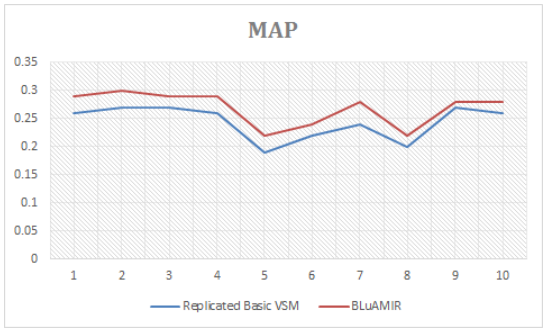


Fig. 5. MAP Comparison between replicated basic VSM and BLuAMIR

links. However, we follow A LSI-based documentation to source code traceability links recovering technique proposed by Marcus and Maletic [17].

At first we created a term-by-document matrix for corpus. Our corpus contains both the source code base and query bug reports. Then, we apply a Singular Value Decomposition (SVD) to construct a subspace (i.e., LSI subspace) [30]. Then, we compute similarity between documents by the cosine or inner product between the corresponding vectors collected from LSI subspace matrix. Typically, two documents are considered similar if their corresponding vectors in the VSM spcae point in the same direction. Note that we preprocess both source code and bug reports by spiliting each word, remove stop words etc. Then, we create the term-by-document matrix. However, finally we retrieve the rankled list of source codes for bug reports based on cosine similarity. We compare between

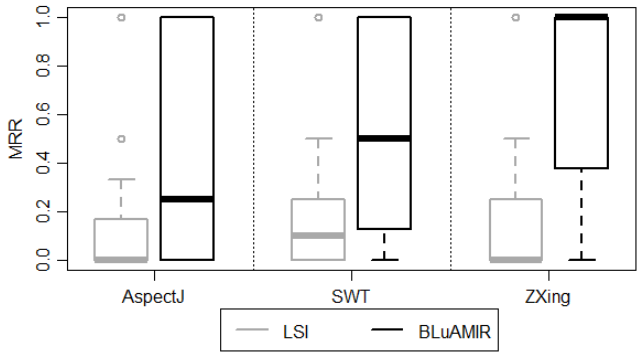


Fig. 6. MRR Comparison between replicated basic LSI and BLuAMIR for AspectJ, SWT and ZXing dataset

replicated LSI and BLuAMIR for three datasets AspectJ, SWT and ZXing, which are depicted in Table VI

TABLE VI
PERFORMANCE OF REPLICATED LSI AND BLUAMIR FOR ASPECTJ, SWT AND ZXING DATASET

| #System | Methodology | Top 1 % | Top 5 % | Top 10 % | MRR | MAP |
|---|---|---|---|---|---|---|
| AspectJ | Replicated LSI | 9.42% | 24.59% | 29.10% | 0.15 | 0.07 |
| | BLuAMIR | 33.20% | 54.92% | 66.39% | 0.43 | 0.23 |
| SWT | Replicated LSI | 13.26% | 30.61% | 53.06% | 0.22 | 0.17 |
| | BLuAMIR | 45.93% | 75.00% | 82.29% | 0.58 | 0.50 |
| ZXing | Replicated LSI | 15.00% | 35.00% | 45.00% | 0.23 | 0.21 |
| | BLuAMIR | 55.00% | 80.00% | 85.00% | 0.67 | 0.62 |

We compare the performance of our proposed approach in terms of top 1, 5, 10 rank (depicted in Table V) and MRR and MAP (shown in Box plot Fig 6 and Fig 7 respectively). We can see that our proposed approach outperforms in all case. So, BLuAMIR successfully retrieve 66.39%, 82.29%, and 85.00% bugs for AspectJ, SWT and ZXing datasets respectively in Top-10. These results provide the answer of our *RQ1(a)*. The box plots in Fig 6 and Fig 7 also demonstrate that BLuAMIR performs better than the baseline in each of the measures with higher medians.

### *F. Answering RQ4*

To answer RQ4, we compare the performance of BLuAMIR with three state-of-the-art techniques - BugScout [22], BugLocator [39] and BLUiR [28] for the the same four dataset

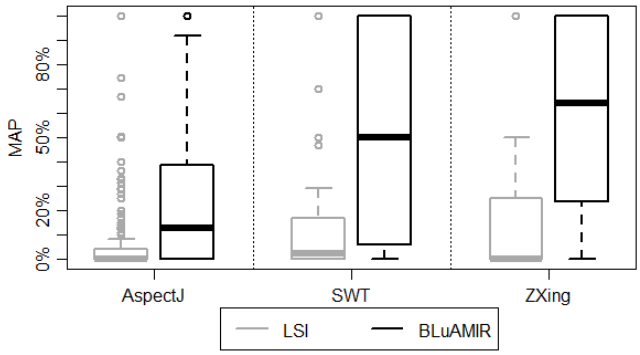


Fig. 7. MAP Comparison between replicated basic LSI and BLuAMIR for AspectJ, SWT and ZXing dataset

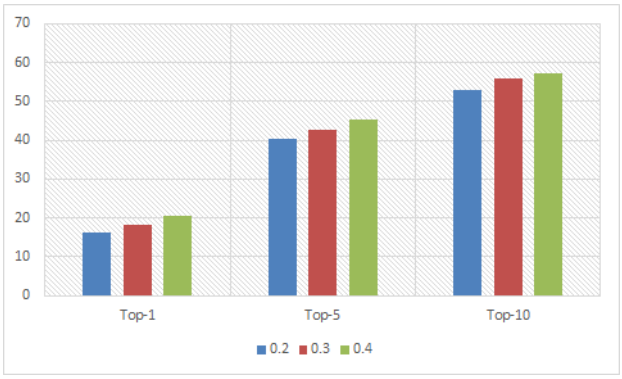


Fig. 8. The impact of $\alpha$ on bug localization performance (Top-1, Top-5, Top-10)

.

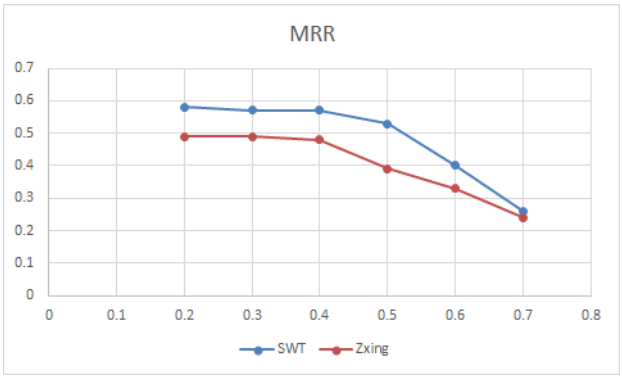


Fig. 9. The impact of $\alpha$ on bug localization performance (MRR)

(i.e., Eclipse, AspectJ, and Zxing depicted in Table VII). Nguyen et al. [22] used two dataset as subject system (i.e., Eclipse and AspectJ) to evaluate BugScout. Saha et al. [28] interpreted the recall at top-1, top-5, and top-10 of BugScout [22] results from a Figure presented in their paper [22]. However, we could compare the performance of BLuAMIR with BugScout for Eclipse and AspectJ dataset for top-1, top-5 and top-10. Therefore, due to data unavailability, we could not compare that performance for MRR@10 and MAP@10 between BugScout and BLuAMIR. For Buglocator [39] and BLUiR [28], 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 BugScout [22] for Eclipse, AspectJ, SWT and Zxing dataset in three performance metrics (i.e., Top-1, Top-5, and Top-10). 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 | 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 | 27.45 | 53.79 | 63.30 | 0.38 | 0.27 |
| AspectJ | 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 | 33.20 | 54.92 | 66.39 | 0.43 | 0.23 |
| SWT | BugLocator | 31.63 | 65.31 | 77.55 | 0.47 | 0.40 |
| | BLUiR | 55.10 | 76.53 | 87.76 | 0.65 | 0.56 |
| | BLuAMIR | 45.93 | 75.00 | 82.29 | 0.58 | 0.50 |
| Zxing | BugLocator | 40.00 | 55.00 | 70.00 | 0.48 | 0.41 |
| | BLUiR | 40.00 | 65.00 | 70.00 | 0.49 | 0.38 |
| | BLuAMIR | 55.00 | 80.00 | 85.00 | 0.67 | 0.62 |

## G. 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-ocuurence Ranking) in Figure 8.

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

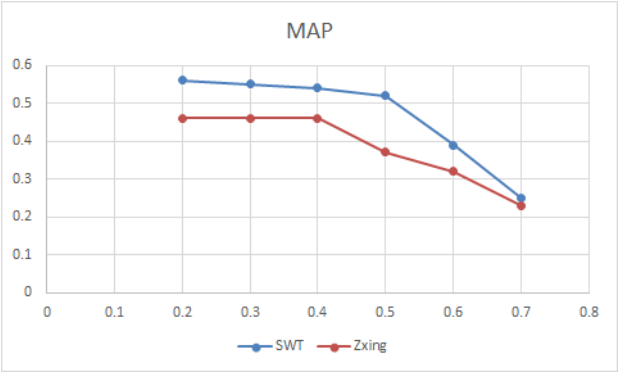


Fig. 10. The impact of $\alpha$ on bug localization performance (MAP)

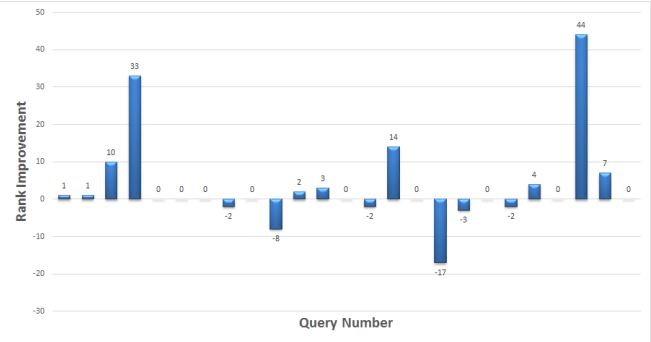


Fig. 11. Quey wise comparison between baseline VSM and BLuAMIR for 30 Eclipse Query

increases. However, after a certain point, further increase of the b value will decrease the performance. For example, Figure 9 and 10 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 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 missmatch 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 11. 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 association 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. 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. [39], 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 [28, 39] and are well- known IR metrics. Thus, we argue that our research has strong construct validity.

TABLE IX
QUERY-WISE RANK COMPARISON ON ECLIPSE DATASET

| Technique | Total | #Improved | Mean | Min. | Max. | #Worsened | Mean | Min. | Max. | # Preserved |
|---|---|---|---|---|---|---|---|---|---|---|
| BugLocator | 3075 | | | | | | | | | |
| BLUiR | 3075 | | | | | | | | | |
| BLuAMIR | 3072 | 1261 (41.05%) | 13 | 1 | 88 | 595 (19.37%) | 38.10 | 1 | 917 | 1216 (39.58%) |

**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 program elements sorted by their suspiciousness. Some of the well known dynamic approaches are spectrum-based fault localization, e.g., [1, 10, 12, 27], model-based fault localization, e.g., [6, 20], dynamic slicing [38], delta debugging [37].

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 [9]. Therefore, FindBug does not even need a bug report. However, it often detects too many false positives and misses many real bugs [33]. 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 [29], the "probabilistic approach" known as BM25 [26], or more recent language modeling [23]. Another empirical study [5] show that all three approaches perform comparably when well-tuned. However, Rao and Kak [25] 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 [8], its probability model was found to be deficient. Lukins et al. [15] use Latent Dirichlet Allocation (LDA), which is a well-known topic modeling approach, to localize bug [15]. However, LSI is rarely used in practice today due to errors in induced concepts introducing more harm than good [8] and LDA is not be able to predict the appropriate topic because it followed a generative topic model in a probabilistic way [14].

Sisman and Kak [32] propose a history-aware IR-based bug localization solution to achieve a better result. Zhou et al. [39] propose BugLocator, which leverages similarities among bug reports and uses refined vector space model to perform bug localization. Saha et al. [28] 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. [21] 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 [35] propose AmaLgam, a new method for locating relevant buggy files that put together version history, similar report, and structure, to achieve better performance. Later [36] 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. Zoeteweij, R. Golsteijn, and A. J. C. van Gemund. A Practical Evaluation of Spectrum-based Fault Localization. *J. Syst. Softw.*, 82(11):1780–1792, November 2009.

[2] 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] A. Bachmann and A. Bernstein. Software process data quality and characteristics: A historical view on open and closed source projects. In *Proc. IWPSE*, pages 119–128, 2009.

[4] S. Deerwester, S. T. Dumais, G. W. Furnas, T. K. Landauer, and R. Harshman. Indexing by Latent Semantic Analysis. *Journal of the American Society for Information Science*, 41(6):391–407, 1990.

[5] 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.

[6] 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.

[7] 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.

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

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

[10] 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.

[11] A. Kontostathis. Essential Dimensions of Latent Semantic Indexing (lsi). In *2007 40th Annual Hawaii International Conference on System Sciences (HICSS'07)*, 2007.

[12] 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.

[13] S. K. Lukins, N. A. Kraft, and L. H. Etzkorn. Bug Localization Using Latent Dirichlet Allocation. *Inf. Softw. Technol.*, 52(9):972–990.

[14] 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.

[15] 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.

[16] 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.

[17] 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.

[18] 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.

[19] A. Marcus, A. Sergeyev, V. Rajlich, and J.I. Maletic. An Information Retrieval Approach to Concept Location in Source Code. In *Proc. WCRE*, pages 214–223, 2004.

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

[21] 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.

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

[23] 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.

[24] M. M. Rahman and C. K. Roy. Improving ir-based bug localization with context-aware query reformulation. In *Proc. ESEC/FSE*, page 12, 2018.

[25] 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.

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

[27] 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.

[28] 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.

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

[30] G. Salton and M. J. McGill. *Introduction to Modern Information Retrieval*. McGraw-Hill, Inc., 1986.

[31] G. Salton, A. Wong, and C. S. Yang. A vector space model for automatic indexing. *Commun. ACM*, 18(11):613–620, 1975.

[32] 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.

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

[34] Q. Wang, C. Parnin, and A. Orso. Evaluating the usefulness of ir-based fault localization techniques. In *Proc. ISSTA*, pages 1–11, 2015.

[35] 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.

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

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

[38] 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.

[39] 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.