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 [27, 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 [23, 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 [15, 17, 18] 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 [17, 18], program comprehension [15] and traceability link recovery problems [16], 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 [10]. The results returned by LSI can also be difficult to interpret, as they are expressed using a numeric spatial representation. Other related studies [12, 21] 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). 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 [27, 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 [16, 17].

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) [16] 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 [21], BugLocator [39] and BLUiR [27] (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)				
Retrieved Documents	S_{VSM}	**GT**	**Retrieved Documents**	S_{VSM}	S_{Assoc}	S_{Total}	**GT**
`ClasspathLocation.java`	1.00	✗	**`CompletionEngine.java`**	**0.40**	**1.00**	**0.80**	✓
`JavaCore.java`	0.74	✗	`AbstractDecoratedTextEditor.java`	0.41	0.73	0.70	✗
`SearchableEnvironmentRequestor.java`	0.73	✗	`AntEditor.java`	0.41	0.73	0.70	✗
`Compiler.java`	0.71	✗	`JavaCore.java`	0.45	0.64	0.70	✗
`AccessRuleSet.java`	0.70	✗	`Engine.java`	0.42	0.70	0.70	✗

GT = Ground **T**ruth

TABLE II
AN EXAMPLE BUG REPORT (#95167, ECLIPSE.JDT.CORE)

Field	Content
Title	[content assist] Spurious "Access restriction" error during code assist
Description	(1) OSGi, Runtime, SWT, JFace, UI, Text loaded from head, (2) open type on `AbstractTextEditor`, (3) at start of createPartContro method, type: PartSite `<Ctrl+Space>`, and (4) it has no effect in the editor, but the status line flashes in red: Access restriction: The type `SerializableCompatibility` is not accessible due to restriction on required project `org.eclipse.swt`. The type name doesn't seem to matter. "abcd" has the same effect. I notice that org.eclipse.ui.workbench.texteditor's classpath has an access rule forbidding **/internal/** refs.

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 [16, 31] but also *three* state-of-the-art approaches – BugScout [21], BugLocator [39] and BLUiR [27] 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 95167) on an Eclipse subsystem namely `eclipse.jdt.core`. Table II shows the *title* and *description* of the bug report. We capture both fields and construct a baseline query by employing standard natural language preprocessing (e.g., stop word removal, token splitting) on them. Then we execute the query with Vector Space Model (VSM) and our approach–BLuAMIR, and attempt to locate the buggy source documents.

According to the ground truth based on bug-fixing history, one source code document (`CompletionEngine.java`) was changed to fix the reported bug. As shown in Table I, we see that traditional *lexical similarity* based approach (VSM) fails to retrieve any buggy source document within the Top-5 positions. On the contrary, our approach, BLuAMIR, combines both *lexical similarity* and *implicit association*, and returns the target buggy document at the top most position of the result list. This is not only promising but also the best possible outcome that an automated approach can deliver.

We also investigate why BLuAMIR performs better than VSM in localizing the buggy document(s). Table I shows different scores from both approaches. We see that several source documents (e.g., `ClasspathLocation.java`) that are retrieved by VSM are strongly similar to the query. However, such similarity does not necessarily make them buggy. In fact, VSM returns the ground truth document at the lowest position of the Top-10 results (not shown in Table I). Thus, *lexical similarity* alone might not be sufficient enough for effective bug localization. However, our approach overcomes such challenge by exploiting the *implicit association* between the query and the buggy documents (Section III-B), and returns the target ground truth at the top-most position. Although the lexical similarity is low (e.g., 0.40), our approach correctly identifies the buggy document using its strong implicit association score (e.g., 1.00) with the given query.

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 Keyword–Document 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 (Fig. 1-(a)) but also provide the corresponding pseudo-code (Algorithm 1). We discuss each of these steps as follows:

(1) Keyword Extraction from Bug Reports: First, we collect *title* and *description* of each bug report extracted from

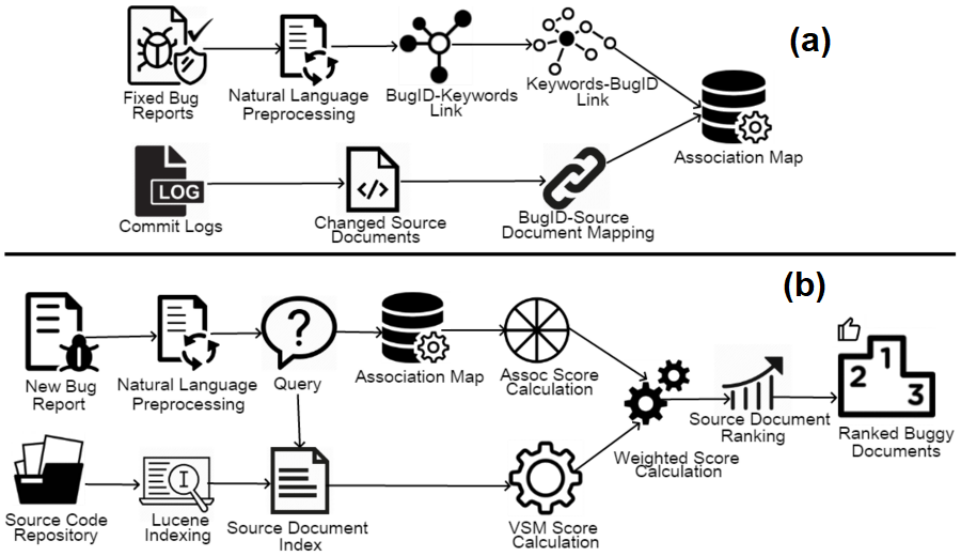

Fig. 1. Schematic diagram of BLuAMIR: (a) Construction of association map between keywords and source documents, and (b) Bug Localization using VSM and implicit association

a subject system. Then we perform standard natural language preprocessing, and remove punctuation marks, stop words and small words from them. Stop words convey very little semantics in a sentence. We use appropriate regular expressions to discard all the punctuation marks and a standard list[1] for stop word removal. Finally, we select a list of remaining keywords from each bug report. Then we create an inherent mapping between the ID of each bug report and their corresponding keywords (Lines 5–6, Algorithm 1). We also construct an inverted index between keywords and their corresponding bug report IDs (Lines 7-8, Algorithm 1).

(2) Extraction of Changed Source Documents from Bug-Fix Commits: We consult with version control history of each subject system, and identify the bug-fixing commits using appropriate regular expressions [3, 36]. In particular, we go through all the commits and identify such commits that contain keywords related to bug fix or resolution in their messages. Then, we collect the changeset (i.e., list of changed documents) from each of these bug-fix commits for our study (Lines 9–10, Algorithm 1). We use several utility commands such as `git, clone` and `log` on Git-based repository for collecting the above information.

(3) Construction of Mapping between Keywords and Changed Source Documents: The above two steps deliver (1) an inverted index (MAP_{kb}) that maps each individual keyword to numerous bug report IDs, and (2) a map (MAP_{bs}) that links each bug report ID to its corresponding changed source documents based on the bug-fixing history. Since both of these maps are connected through bug report IDs, keywords from each bug report also enjoy an implicit relationships with the corresponding changed source code documents. We leverage such implicit, transitive relationships and construct a bipartite graph (e.g., Fig. 2) by explicitly connecting the keywords from each bug report and their corresponding changed source documents (Lines 8–24, Algorithm 1). Here, one or more keywords could be linked to single buggy source document. Conversely, single source document could be linked to one or more keywords from multiple bug reports.

Algorithm 1 Construction of Association Map

1: **procedure** MAPCONSTRUCTION(BRC, BFC)
2: ▷ BRC: a collection of past bug reports
3: ▷ BFC: bug-fix commit history
4: $MAP_{KS} \leftarrow \{\}$ ▷ an empty association map
5: ▷ Create map between Bug ID and keywords
6: $MAP_{bk} \leftarrow$ createBugIDtoKeywordMap(BRC)
7: ▷ Create an inverted index between keywords and ID
8: $MAP_{kb} \leftarrow$ createKeywordtoBugIDMap(MAP_{bk})
9: ▷ Map between Bug ID and changed source documents
10: $MAP_{bs} \leftarrow$ createBugIDtoSourceMap(BFC)
11: ▷ Mapping keywords to the changed source documents
12: $KW \leftarrow$ collectKeywords(MAP_{kb})
13: **for** Keyword $kw \in KW$ **do**
14: $ID_{kw} \leftarrow$ extractBugIDs (kw, MAP_{kb})
15: **for** BugID $id_{kw} \in ID_{kw}$ **do**
16: ▷ Get the linked source documents
17: $SD_{link} \leftarrow$ getDocuments(MAP_{bs}, id_{kw})
18: ▷ Map all source documents to this keyword
19: $MAP[kw].link \leftarrow MAP[kw].link \cup SD_{link}$
20: **end for**
21: **end for**
22: ▷ collect all keyword–source document mappings
23: $MAP_{KS} \leftarrow MAP[KB]$
24: **return** MAP_{KS}
25: **end procedure**

An Example Association Map with Bipartite Graph: In Mathematics, *bipartite graph* is defined as a special graph that (1) has two disjoint set of nodes and a set of edges and (2) each of its edges connects two nodes from the two

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

TABLE III
A BUG REPORT (#322401, ECLIPSE.UI.PLATFORM)

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

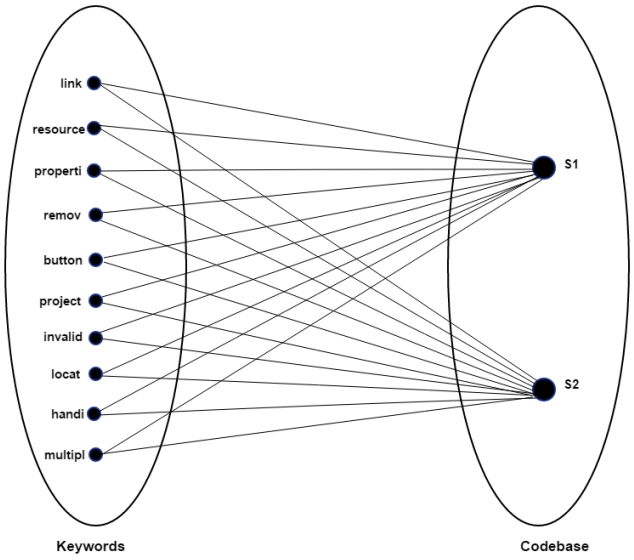

Fig. 2. An example association map using bipartite graph

different sets but not from the same set [2]. In our study, these two sets correspond to keywords and source code documents. Thus, no connection between any two keywords or any two source documents is allowed. We construct an example bipartite graph for a bug report (ID 322401) from Eclipse UI Platform. Table III shows the *title* and *description* of the bug report. According to the bug-fix history, these two documents - **S1:** `IDEWorkbenchMessages.java` and **S2:** `LinkedResourceEditor.java`– constitute the ground truth for the bug report. Fig. 2 shows the bipartite graph for the bug report along with its ground truth. We see that ten unique keywords and two source documents represent the two set of nodes and each node is connected to all the nodes from another set. In our approach, we iteratively update such a bipartite graph with new nodes and connections generated from each of the bug reports of a subject system.

B. Bug Localization using VSM and Implicit Association

Fig. 1-(b) shows the schematic diagram and Algorithm 2 presents the pseudo-code of our bug localization component. We have constructed an association map that connects the keywords from a bug report to its corresponding changed source documents (Section III-A). We leverage this association map, and return a list of source code documents that are not only lexically similar but also strongly associated with a given query (bug report at hand) through bug-fixing history. We thus calculate two scores for each candidate source document and then suggest the Top-K documents as follows:

Algorithm 2 Proposed Bug Localization Approach

procedure BUGLOCALIZATION(Q, $SCrepo$, MAP_{KS})
  ▷ Q: a given query (bug report)
  ▷ $SCrepo$: a source code repository
  ▷ MAP_{KS}: keyword-source document map
  $RL \leftarrow \{\}$ ▷ list of buggy source documents
  ▷ Create an index of source documents
  $Index \leftarrow$ createLuceneIndex($SCrepo$)
  ▷ Collect lexically similar documents
  $VSM \leftarrow$ getLexSimDocuments(Q, $Index$)
  ▷ Collecting associated documents
  $Assoc \leftarrow$ getAssociatedDocuments(Q, MAP_{bk})
  ▷ Combine both candidate document lists
  $C \leftarrow \{VSM.docs \cup Assoc.docs\}$
  for SourceDocument $d \in C$ **do**
    ▷ Combine lexical and association scores
    $C[d].score \leftarrow VSM[d].score$
    $C[d].score \leftarrow C[d].score + Assoc[d].score$
  end for
  ▷ Sort the candidates and collect Top-K documents
  $RL \leftarrow$ getTopKDocuments(sortByScore(C))
  return RL
end procedure

Lexical Similarity Score: Source code documents often share a major overlap in vocabulary with a submitted bug report. Many of the existing studies [27, 28, 36, 39] consider such vocabulary overlap (i.e., lexical similarity) as a mean to localize the buggy source documents. These studies generally use Vector Space Model (VSM) for calculating the vocabulary overlap. VSM is a classical approach for constructing vector representation of any text document (e.g., bug report, source document) [31]. First, it encodes a document collection using a term-by-document matrix where each row represents a term and each column represents a document. Second, each matrix cell is defined as the frequency of a term (i.e., term frequency) within a specific document. Thus, lexical similarity between a given query (bug report) and a candidate source document is computed as the cosine or inner product between their corresponding vectors from the matrix. Since term frequency (TF)-based vector representation might be biased toward large documents, several studies [27, 39] represent their vectors using TF-IDF. TF-IDF assigns higher weights to such terms that are frequent within a document but not not frequent across the document collection [29]. TF-IDF stands for term frequency times inverse document frequency. In classic VSM, term frequency $tf(t,d)$ and inverse document frequency $idf(t)$ are defined as follows:

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

Here f_{td} refers to the frequency of each unique term t in a document d, n_t denotes the document frequency of term t and $\#docs$ is the total number of documents in the collection. Thus, the lexical similarity between a given query Q (bug

report) and a candidate source document d is calculated as follows:

$$lexicalSimScore(Q,d) = cosine(Q,d) = \sum_{t \epsilon Q \bigcap d} (\frac{f_{tQ}}{\#terms}) \times (\frac{f_{td}}{\#terms}) \times log(\frac{\#docs}{n_t})^2 \times \frac{1}{\sqrt{\sum_{t \epsilon Q}((\frac{f_{tQ}}{\#terms}) \times log(\frac{\#docs}{n_t}))^2}} \times \frac{1}{\sqrt{\sum_{t \epsilon d}((\frac{f_{td}}{\#terms}) \times log(\frac{\#docs}{n_t}))^2}}$$

Here, $\#terms$ refers to the total number of terms in a document collection. $lexicalSimScore$ takes a value between 0 and 1 where 0 means means total dissimilarity and 1 means strong lexical similarity between the query Q and the candidate source code document d. We use Apache Lucene for calculating the lexical similarity (Lines 6–9, Algorithm 2).

Implicit Association Score: We analyse the implicit associations between a given query (bug report) and each candidate source document using our association map (i.e., bipartite graph) (Section III-A). In this map, while each keyword could be linked to multiple candidate documents, each document could also be linked to multiple keywords. We perform standard natural language processing on a given query, and extract a list of query keywords. We then identify such source documents in the map that are linked to each of these keywords. Since these links were established based on the bug-fixing history, they represent an *implicit relevance* between the keywords and the documents. It should be noted that such relevance does not warrant for lexical similarity. We analyse such links for all the keywords ($q \in Q$) of a query, and determine how frequently each candidate source document d has associated with these keywords as follows:

$$associationScore(Q,d) = \sum_{q \in Q} \#Link(q,d)$$

Here $\#Link(q,d)$ returns the frequency of association between the keyword q and the source document d across the bug-fixing history of a subject system. That is, $associationScore$ assigns a score to each candidate document by capturing their historical co-occurrences with the query keywords across the bug-fixing history.

Final Score Calculation: The above two sections deliver two different scores (*lexical similarity*, *implicit association*) for each of the candidate source code documents. Since these scores could be of different ranges, we normalize both of them, combine them, and then calculate the final score (Lines 7–12, Algorithm 2) as follows:

$$FinalScore = (1-\alpha) \times Norm(lexicalSimScore) + \alpha \times Norm(associationScore)$$

Here, the weighting factor α varies from 0.2 to 0.4, and the detailed justification is provided during experiment and discussions (Section IV-G).

Once the final score is calculated for each of the candidate source documents from the corpus, we return the Top-K results as the buggy documents for the given query Q.

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. [27] used to evaluate BugLocator and BLUiR respectively. This dataset contains 3479 bug reports in total from four popular open source projectsEclipse, SWT, AspectJ and ZXing along with the information of fixed files for those bugs. The detail of our dataset is presented in IV. Eclipse[2] is a well-known large-scale open source system and it is widely used in empirical software engineering research. SWT[3] is a component of Eclipse. The AspectJ[4] project is a part of the iBUGs public dataset provided by the University of Saarland. Zxing[5] 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.

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

Here, N may be 1, 5 or 10. We assume that if at least one associated file is presented in the resulted ranked list, then the given bug is located. The higher the metric value, the better the bug localization performance.

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

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

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:

$$AP = \sum_{k=1}^{M}\frac{P(k)\times pos(k)}{number\,of\,positive\,instances} \tag{2}$$

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:

$$P(k) = \frac{\#buggy\,Files}{k} \tag{3}$$

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 [16] 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 [21], BugLocator [39] and BLUiR [27] 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 [16] 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 Hit@1, Hit@5, Hit@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 Hit@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 Hit@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

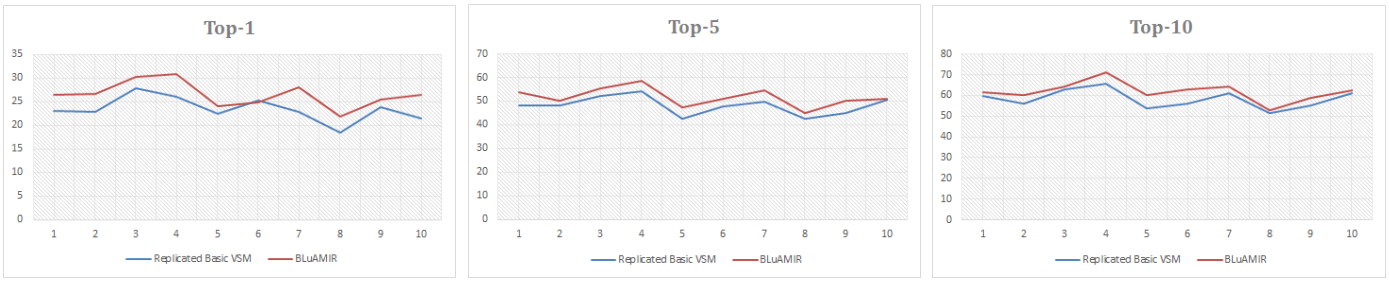

Fig. 3. Performance Comparison between replicated basic VSM and BLuAMIR for Top-1, Top-5 and Top-10 Retrieval

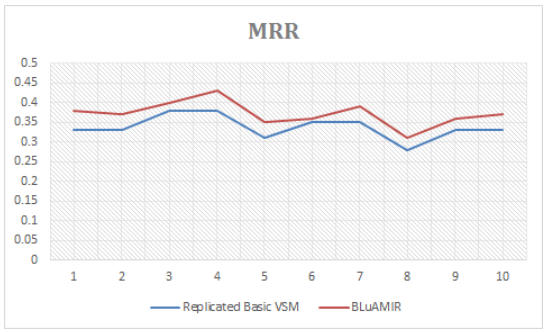

Fig. 4. MRR Comparison between replicated basic VSM and BLuAMIR

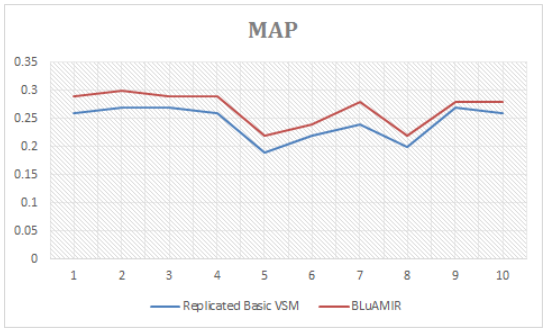

Fig. 5. MAP Comparison between replicated basic VSM and BLuAMIR

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 [15, 17] 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 links. However, we follow A LSI-based documentation to source code traceability links recovering technique proposed by Marcus and Maletic [16].

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 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 Hit@k 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 Hit@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 [21], BugLo-

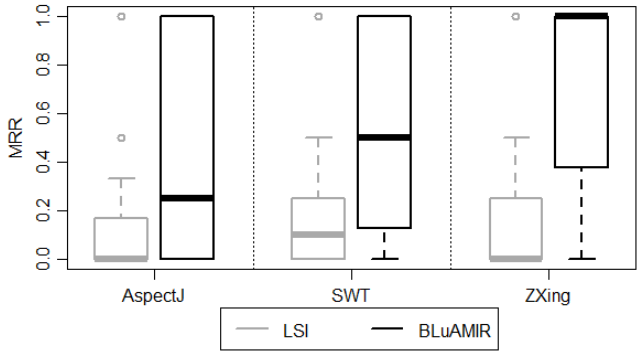

Fig. 6. MRR Comparison between replicated basic LSI and BLuAMIR for AspectJ, SWT and ZXing dataset

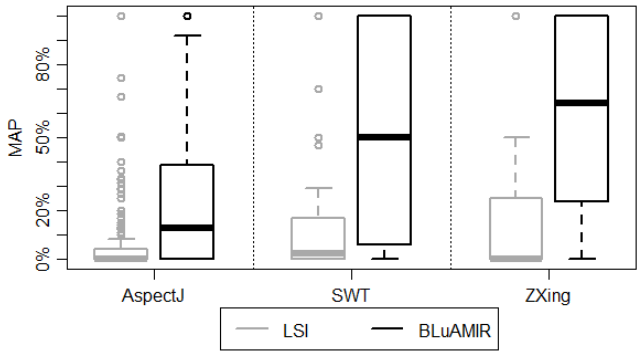

Fig. 7. MAP Comparison between replicated basic LSI and BLuAMIR for AspectJ, SWT and ZXing dataset

cator [39] and BLUiR [27] for the the same four dataset (i.e., Eclipse, AspectJ, and Zxing depicted in Table VII). Nguyen et al. [21] used two dataset as subject system (i.e., Eclipse and AspectJ) to evaluate BugScout. Saha et al. [27] interpreted the recall at Hit@1, Hit@5, and Hit@10 of BugScout [21] results from a Figure presented in their paper [21]. However, we could compare the performance of BLuAMIR with BugScout for Eclipse and AspectJ dataset for Hit@k. Therefore, due to data unavailability, we could not compare that performance for MRR@10 and MAP@10 between BugScout and BLuAMIR.

TABLE VII
PERFORMANCE COMPARISON BETWEEN BUGSCOUT, BUGLOCATOR, BLUIR 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

For Buglocator [39] and BLUiR [27], 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 both BugScout [21] and BugLocator [39] consistently for Eclipse, AspectJ, SWT and Zxing dataset in three performance metrics (i.e., Hit@1, Hit@5 and Hit@10). Moreover, BLuAMIR can localize bugs with 27% higher reciprocal ranks (i.e., MRR) and 32% higher precision (i.e., MAP) than BugLocator [39]. For AspectJ and SWT dataset, we can see our tool BLuAMIR outperforms BLUiR [27] for Hit@1, Hit@5, Hit@10 and MRR@10. However, though BLuAMIR shows higher precision (i.e., MAP) than BLUiR [27] for Zxing dataset, this also shows comparable precision with BLUiR [27] for AspectJ. For Eclipse dataset BLuAMIR produce better results than BLUiR [27] for Hit@5 and Hit@10 retrieval, but BLUiR [27] shows better performance than BLuAMIR for Hit@1, MRR and MAP. On the other hand, for SWT dataset BLuAMIR shows comparable performance for Hit@5 and does not outperform for Hit@1, Hit@10, MRR and MAP. So, we investigate the SWT dataset with several weighting function and the results are demonstrated in Table VIII. We can see, if we lower the value of weighting function, which produces better result. Among these four weighting, when $alpha$=0.2, BLuAMIR shows better Hi@1, Hit@5 retrieval, higher Reciprocal Rate (i.e.,MRR@10) and higher Precision (ie.e, MAP@10). It is noted that higher accuracy for Hit@10 is found for $alpha$=0.3, which is comparable with BLUiR[27]. However, right now, we can not say for which reason, BLuAMIR produces bad results for some cases, but we plan to keep this as our future study. However, 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 VIII
PERFORMANCE OF BLUAMIR ON SWT DATASET FOR DIFFERENT WEIGHTING VALUES

α	Top 1 %	Top 5 %	Top 10 %	MRR	MAP
0.1	43.75	73.96	83.33	0.57	0.50
0.2	48.96	75.00	83.33	0.60	0.51
0.3	47.92	72.92	85.42	0.59	0.51
0.4	45.93	75.00	82.29	0.58	0.50

G. Answering RQ2

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

Weighting Function Comparison on Eclipse Dataset: We compute performance Hit@k accuracy, MRR@10 and MAP@10 for different weighting function such as α is 0.2, 0.3, 0.4. The results are presented in Table IX. Here, it shows, more α 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. However, we also illustrate the impact of

TABLE IX
PERFORMANCE OF BLUAMIR ON ECLIPSE DATASET FOR DIFFERENT WEIGHTING VALUES

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

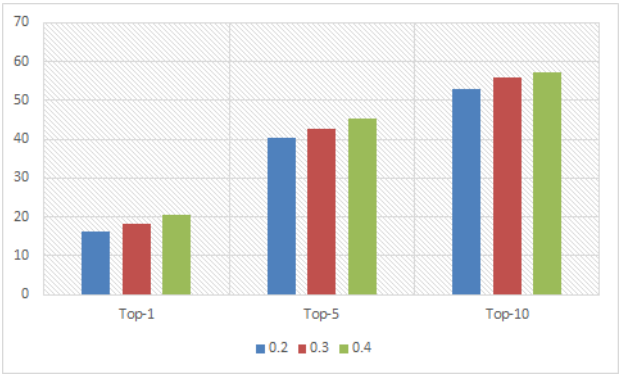

Fig. 8. The impact of α on bug localization performance (Top-1, Top-5, Top-10)

.

weighting function α for Hit@1, Hit@5 and Hit@10 retrieval by BLuAMIR on Eclipse dataset in Figure 8.

We evaluate the impact of association score on bug localization performance, with different α values in terms of MAP@10 and MRR@10 for SWT and Zxing datasets. At the beginning, the bug localization performance increases when the α value increases. However, after a certain point, further increase of the $alpha$ value will decrease the performance. For example, Figure 9 and 10 show the bug localization performance (measured in terms of MRR@10 and MAP@10) for the SWT and Zxing projects. When the α value increases from 0.1 to 0.4, both MRR and MAP values increases consistently. Increasing α value further from 0.4 to 0.7 however leads to lower performance. Note that we obtain the best bug localization performance when α is between 0.3 and 0.4. As association score is based on the association map between

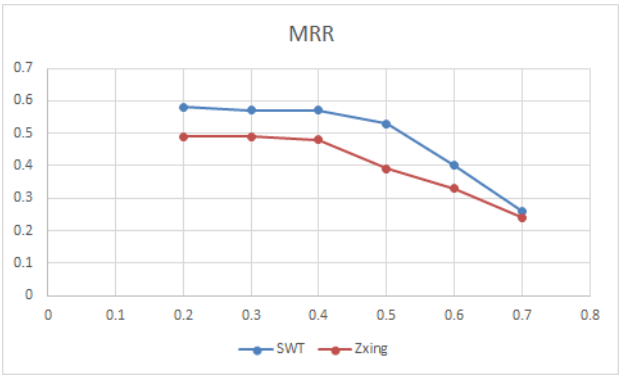

Fig. 9. The impact of α on bug localization performance (MRR)

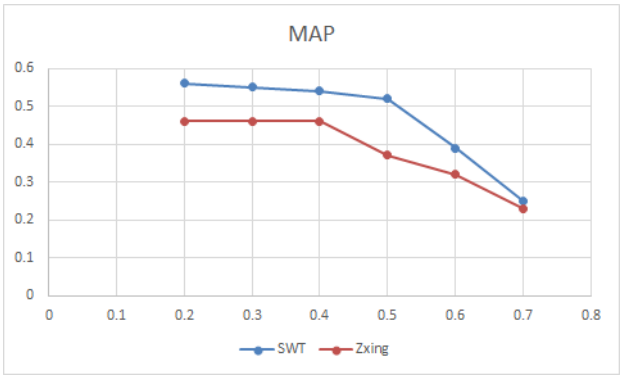

Fig. 10. The impact of α on bug localization performance (MAP)

keywords and their associated source files, thus no direct matching of vocabulary is required. Therefore, the results obtained from the impact of α on bug localization performance (i.e., Hit@k, MRR@10 and MAP@10) also suggest that in the case of vocabulary missmatch issue, our association score can assist to improve the retrieval performance. This also answers RQ2.

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 X. 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 **??**. 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 #95561. The title of this bug is *[Perspectives] Workbench flashes when synchronizing*. We have found 1 source code file (i.e., org.eclipse.ui.internal.WorkbenchPage.java) in rank #2 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 0.45, which is lower than other smaller 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 0.95, which make total score of 0.65 (i.e., 0.95*0.40+0.45*0.60), put this on rank #2. From this case study, it is clear that association mapping between fixed bug report keywords into their

TABLE X
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%)

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. We also noted that the ground truth associated with bug #95561 contains three large files (i.e., Workbench.java, WorkbenchPage.java, and WorkbenchWindow.java), and all of them are too large (i.e., having sizes 26 KB, 41 KB and 32KB) to retrieve by any VSM-based technique. So, this kind of problem can be successfully eliminated by our proposed association score.

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 four dataset - all of them collected from the same benchmark dataset of bug reports and source code shared by Zhou et al. [39]. 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., Hit@k 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 [27, 39] and are well- known IR metrics. Thus, we argue that our research has strong construct validity.

Reliability: In our experiment section, we performed numerous experiments using various combinations of weighting functions to find the optimum parameters and the best accuracy of bug localization. The optimized α 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, 9, 11, 26], model-based fault localization, e.g., [6, 19], 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 [8]. 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 [25], or more recent language modeling [22]. Another empirical study [5] show that all three approaches perform comparably when well-tuned. However, Rao and Kak [24] 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 [7], its probability model was found to be deficient. Lukins et al. [14] use Latent Dirichlet Allocation (LDA), which is a well-known topic modeling approach, to localize bug [14]. However, LSI is rarely used in practice today due to errors in induced concepts introducing more harm than good [7] and LDA is not be able to predict the appropriate topic because it followed a generative topic model in a probabilistic way [13].

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. [27] 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. [20] 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 three sources of information i.e., bug reports, source codes and version history.

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 bug localization technique not only based on lexical similarity but also an implicit association map between bug report keywords with their associated source codes. We perform a large-scale experiments on four projects, namely Eclipse, SWT, AspectJ and ZXing to localize more than 3,000 bugs. Our experiment of those dataset show that on average our technique can locate buggy files with a Top-10 accuracy of 74.06% and a mean reciprocal rank@10 of 0.52 and a mean precision average@10 of 41%, which are highly promising. We also compare our technique with three state-of-the-art IR-based bug localization techniques i.e., BugScout[21], BugLocator[39] and BLUiR[27]. This also confirms superiority of our technique. 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. This also confirms superiority of our proposed bug localization approach.

In the future, we will explore if several other bug related information such as bug report structure, source code structure, stack traces, reporter information, similar bug 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. S. Asratian, T. M. J. Denley, and R. Häggkvist. *Bipartite Graphs and Their Applications*. Cambridge University Press, New York, NY, USA, 1998.

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

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

[9] 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.

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

[11] 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.

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

[13] 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.

[14] 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.

[15] 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.

[16] 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.

[17] 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.

[18] 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.

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

[20] 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.

[21] 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.

[22] 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.

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

[24] 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.

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

[26] 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.

[27] 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.

[28] R.K. Saha, J. Lawall, S. Khurshid, and D.E. Perry. On the Effectiveness of Information Retrieval Based Bug Localization for C Programs. In *Software Maintenance and Evolution (ICSME), 2014 IEEE International Conference on*, pages 161–170, Sept 2014.

[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.