Improved Bug Localization using Keyword-Source Token Co-occurrence.

Shamima Yeasmin   Mohammad Masudur Rahman   Chanchal K. Roy   Kevin A. Schneider

Department of Computer Science, University of Saskatchewan, Canada

shy942@mail.usask.ca, {chanchal.roy, kevin.schneider}@usask.ca

ABSTRACT

Bug localization is one of the most challending tasks undertaken by the developers during software maintenance. Existing studies mostly rely on lexical similarity between the bug reports and source code for bug localization. However, such similarity always does not exist, and these studies suffer from vocabulary mismatch issues. In this paper, we propose a bug localization technique that (1) not only uses lexical similarity between bug report and source code documents but also (2) exploits the co-occurrences between keywords from the past reports and source tokens from corresponding changed code. Experiments using a collection of 6000 bug reports show that our technique performs significantly higher in terms of Hit@K and MAP than one state-of-the-art IR-based bug localization techniques using TF-IDF and other LDA based technique.

Categories and Subject Descriptors

H.4 [**Information Systems Applications**]: Miscellaneous; D.2.8 [**Software Engineering**]: Metrics—*complexity measures, performance measures*

Keywords

Bug Report, Topic Evolution, Summary, Visualization

1. INTRODUCTION

Checking Bug localization is the process of locating the source codes that need to be changed in order to fix a given bug. Locating buggy files is time-consuming and costly if it is done by manual effort, especially for the large software system when the number of bug of that system becomes large. Therefore, effecting methods for locating bugs automatically from bug reports are highly desirable. In automatic bug localization technique, it takes a subject system as an input and produces a list of entities such as classes, methods etc. against a search query. For example, information retrieval

based techniques rank the list of entities by predicted relevance and return a ranked list of entities or source codes which may contain the bug. The bug localization techniques are also affected by the fact of designing effective query. If a query contains inadequate information, then the retrieval results will not be relevant at all. One other thing that the performance of existing bug localization approach did not reach to an accepted level and so far those studies showed good results for a small set of bugs. Therefore, in this paper, we apply a bug localization technique on large dataset that exploits an association link established from bug report repository to source code base through commit logs.

In existing studies, information retrieval techniques [9] [6] [4] [5] [2] are applied to automatically search for relevant files based on bug reports. In one case of Text Retrieval (TR) based approaches, Zhou et al. [9] propose BugLocator using revised Vector Space Model (rVSM), which requires the information extracted from bug reports and source codes. One of the issue association with TR-based technique is treating source codes as flat text that lacks structure. But exploiting source code structure can be a useful way to improve bug localization accuracy. Due to the fuzzy nature of information retrieval, textually matching bug reports against source files may have to concede noise (less relevant but accidentally matched words). However, large files are more likely to contain noise as usually only a small part of a large file is relevant to the bug report. So, by treating files as single units or favoring large files, the techniques are more likely to be affected by the noise in large files. So, Saha et al. [6] present BLUiR, which retrieve structural information from code constructs. However, bug reports often contain stack-trace information, which may provide direct clues for possible faulty files. Most existing approaches directly treat the bug descriptions as plain texts and do not explicitly consider stack-trace information. Here, Moreno et al. [4] combine both stack trace similarity and textual similarity to retrieve potential code elements. To deal with two issues associated with source code structure and stack trace information, Wong et al. [8] proposed a technique that use segmentation i.e., divides each source code file into segments and use stack-trace analysis, which significantly improve the performance of BugLocator. LDA topic-based approaches [5] [2] assume that the textual contents of a bug report and it's corresponding buggy source files share some technical aspects of the system. Therefore, they develop topic model that represents those technical aspects as topic. However, existing bug localization approaches applied on small-scale data for evaluation so far. Besides the problem of small-

scale evaluations, the performance of the existing bug localization methods can be further improved too. For example, using Latent Dirichlet Allocation (LDA), only buggy files for 22% of Eclipse 3.1 bug reports are ranked in the top 10 [25]. But, now it is an important research question to know how effective these approaches are for locating bugs in large-scale (i.e., big data). In the field of query processing Sisman and Kak [7] proposed a method that examines the files retrieved for the initial query supplied by a user and then selects from these files only those additional terms that are in close proximity to the terms in the initial query. Their [7] experimental evaluation on two large software projects using more than 4,000 queries showed that the proposed approach leads to significant improvements for bug localization and outperforms the well-known QR methods in the literature. There are two common issues associated with existing studies. First, most of the bug localization techniques exploit lexical similarity measure for retrieving relevant files from the code base. So, it is expected that the search query constricted from new bug report should contain keywords similar to code constructs in code base. This issue demands developers or users previous expertise on the given project, which can not be guaranteed in real world. Second, closely related to the first issue there is another well known problem called vocabulary mismatch. In order to convey the same concept on both search query (i.e., new bug report) and source code files the developers tend to use different vocabularies. This issue questions the applicability of exploiting lexical similarity measure.

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

We also replicate two state of the art bug localization techniques in order to compare the performance of our proposed approach with these two.

2. AN EXAMPLE USE CASE SCENARIO

Note: Provide an example such as say for bug B1 source code files F1, F2, F3 are modified or changed. Now a new bug B has similar to B1 in terms of cosine similarity. Then prove that F1, F2, F3 or one or two of them are contained in the list of files which would be changed or modified for fixing that new bug B. Lets consider a bug having ID ”” from Eclipse-UI-Platform software. From Git repository it is discoverd that so far n number of file have been modified in order to fix this bug. We also csn find the source code address of those changed files. Now,

In another example, show that the words or keywords of a bug B are aslo presented in the source code files F1, F2 or F3.

3. EXISTING APPROACHES

We have replicated two existing bug localization approaches:

(i) TF-IDF based bug localization approach and (ii) LDA topic based bug localization technique. In the following subsections we will briefly discuss each of them.

3.1 TF-IDF Based Bug Localization Technique:

In this technique, each source code file is ranked based on source code file scores and similar bugs information. Source code file contains words those can be also occurred in the bug reports. This is considered as a hint to locate buggy files. If a new bug is similar to a given previously located bug, then there is a possibility that the source code files located for the past bug can provide useful information in finding buggy files for that new bug. Based on these two assumptions, we compute scores for all source code files for a given project. However, we need to focus on some concepts which are required to understand the system. They are described as follows:

Ranking based on TF-IDF calculation: The basic idea of a TF-IDF model is that the weight of a term in a document is increasing with its occurrence frequency in this specific document and decreasing with its occurrence frequency in other documents [9]. In this approach we have used a revised Vector Space Model (rVSM) proposed by Zhou et al. [9] in order to index and rank source code files. The main difference between classic VSM and revised VSM is that in case of revised version logarithm variant is used in computing term frequency. The equation is:

$$tf(t + d) = log(f_{td}) + 1 \qquad (1)$$

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

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

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

Ranking based on similar bug information The assumption of this ranking is similar bugs of a given bug tend to modify similar source code files. Here, we construct a 3-layer architecture as described in [9]. In the top layer (layer 1) there is a bug B which represents a newly reported bug. All previously fixed bug reports which have non-negative similarity with bug B are presented in second layer. In third layer all source code files are shown. In order to resolve each bug in second layer some files in the corpus were modified or changed, which are indicated by a link between layer 2 and layer 3. Foe all source code files in layer 3, similarity score is computed, which can be referred to as degree of similarity. The score can be defined as:
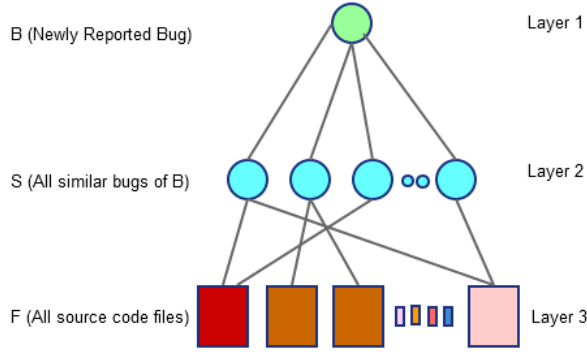
Figure 1: Bug and its Similar Bug Relationship with Source Code Files:

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

(3)

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

Combining Both Ranks: We combine the both scores based on source code score and similar bugs score as in [9] as follows:

$$FinalScore = (1 - \alpha) \times N(rVSMScore) + \alpha \times N(SimiScore)$$

(4)

3.2 LDA Topic Based Bug Localization Technique:

The main assumption behind this technique is the textual content of the bug reports and their associated buggy source code files tend to describe some common technical aspects. So, if we could identify the technical topics extracted from both bug reports and source codes, we could recommend the files shared technical topics with the newly reported bug reports. If a newly reported bug has similar technical topics with some previously fixed bug reports, the fixed files could be a good candidate files for newly reported bug.

LDA (Latent Dirichlet Allocation) is a probabilistic and fully generative topic model. It is used to extract latent (i.e., hidden) topics, which are presented in a collection of documents. It also model each document as a finite mixture over the set of topics [add link here]. In LDA, similarity between a document d and a query q is computed as the conditional probability of the query given that document [3].

$$Sim(q, d_i) = P(q|d_i) = \prod_{q_k \epsilon q} P(q_k|d_i)$$

(5)

Here q_k is the kth word in the query q Thus, a document (i.e., source code file) is relevant to a query if it has a high probability of generating the words in the query.

We perform the following steps in order to localize buggy files for newly reported bug using LDA based approach:

- Apply topic modeling on the source code files. The output contains a certain number of topics and some associated keywords for each topic. We also get some other distribution files such as document-topic, word-topic etc.

- Now work with the documents topic distribution file. Make a list of source code documents or files for each topic. So, we wiill have a list that contain all topics and their associated source code documents.

- Here our query is the newly reported bug. This contains information in the bug reports such as title and short description etc. We all do inference for this query using a topic modeling tool. It will extract all topic associated with the query (i.e., newly reported bug).

- Now we need to work with topic keywords. We are going to perform a comparison between newly reported bug or the given query and source code files using topic information. That means we will compare topic-keywords associated with topics inferred for the query with topic-keywords of each topic extracted from source code documents.

- We will rank them based on topic-keyword similarity. So, now we know which are the top most topics, and we already have information regarding topic-document relationship, we will retrieve all source code files associated with all those top most topic as recommended buggy files.

4. PROPOSED SYSTEM DIAGRAM

Our proposed approach combine lexical similarity and co-occurence similarity measure. Zhou et al. [9] proposed BugLocator based on two different similarity scores- one is rVSM score and the other one is Simi score. In our porposed bug localization approach, we retrieve relevant ranked files based on three different scores- 1) rVSM, 2) Simi and 3) Word Co-occurence. So, we have divided our approach into two different sections or parts- 1) calculate rVSM and Simi scores and co-occurence measure and 2) combine all three scores in order to localize recommended buggy source files for a given newly reported bug. As we have combined two existing scores with our proposed word co-occirence score, we will discuss the system diagram of association map database in Part I and then we represent our overall system diagram in Part II.

4.1 Part I: Mapping Bug Source Code Links

In Part I, we construct an association map database - between keywords extracted from previously fixed bug reports collection and source code links. The system diagram for Part I is given in Fig. 2.

This mapping construction involves two steps. At first we create association map using information contained in bug reports collection and commit logs. We collect information from title and description fields of each bug report. There are also developers comment section in a bug report, which is highly informative. But large documents also tend to reduce performance by taking too long to process. So, we reluctant to include developers comments of each bug report. However, we extract keywords from each bug report after
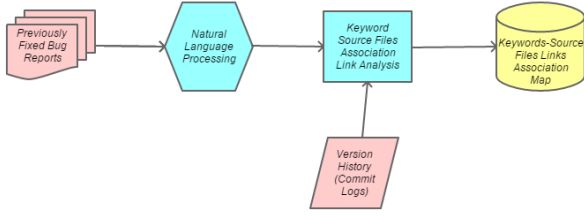
Figure 2: Proposed System Diagram: Mapping

preprocessing them such as stop word removal. In version repository, we have commit logs, where the developers commit for several changes in a software project. For example, when a bug report is fixed, the developer who fixed this bug also creates a commit log containing which type of change he had to made to resolve the bug associated with the location of the source code files where the change has made. So, if we analyze commit logs, we can retrieve source code files information which have been changed in order to fix a given bug. Here the idea is, we first extract keywords from bug report and source code links for the same bug from commit log and then construct an association map database containing this mapping information. The links between keywords and source code links can be described as: each or several keywords can be linked to one or more source code links and each source code link can be linked to one or several keywords. This way we connect keywords and source code links to form an association map database named keyword-source code links.

4.2 Part II: Bug Localization Using rVSM, and Co-occurence Ranks

The system diagram for this part has illustrated in Fig. 3. In our system, we have three different ranks i.e., rVSM, Simi and Co-occurence. We utlize an existing technique proposed by Zhou et al. [9] for computing rVSM and Similarity scores. Basically, rVSM is a TF-IDF based score, which is measured between query and source code files. On the other hand, Simi score refers to that fact that if a bug is similar to another bug, then they both tend to relate to same sources. However, we describe both scores in Section 3. We have constructed an association map databases from keywords collected from bug report to its source code location information in Section 4. For the candidate keyword tokens for the initial query, we exploit association map database (i.e., keyword-source code links) and retrieve the relevant source code files. We use some heuristic functions in order to combine three ranks and recommend buggy relevant source files.

5. PROPOSED APPROACH

Our proposed approach consists of two parts - (i) constructing association map databases and (ii) retrieve relevant buggy source code files.

5.1 Construction of Association Map Database Between Bug Reports and Source Files

In this part, we construct two association map databases - one is between keywords and source code links extracted from bug reports and commit messages respectively and other one is between source code links and code token extracted from commit messages and source code content re-

---

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

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

---

spectively. This section can be further divided into several parts: keyword extraction from bug reports, source code link extraction from commit logs, keyword- source code linking, code tokens extraction from source codes and source code-code token linking. They are discussed in the followings:

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

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

Keyword- Source Code Linking: At this point, in one side we have pre-processed keywords associated with each bug report and on the other side we have a relationship information between bug report ID and buggy source code links. We construct a bipartite graph between keywords collected from a bug report to its buggy source code locations. Here, one or more keywords can be linked to one or more buggy source code files links and a source code file link can be linked to one or more keywords.
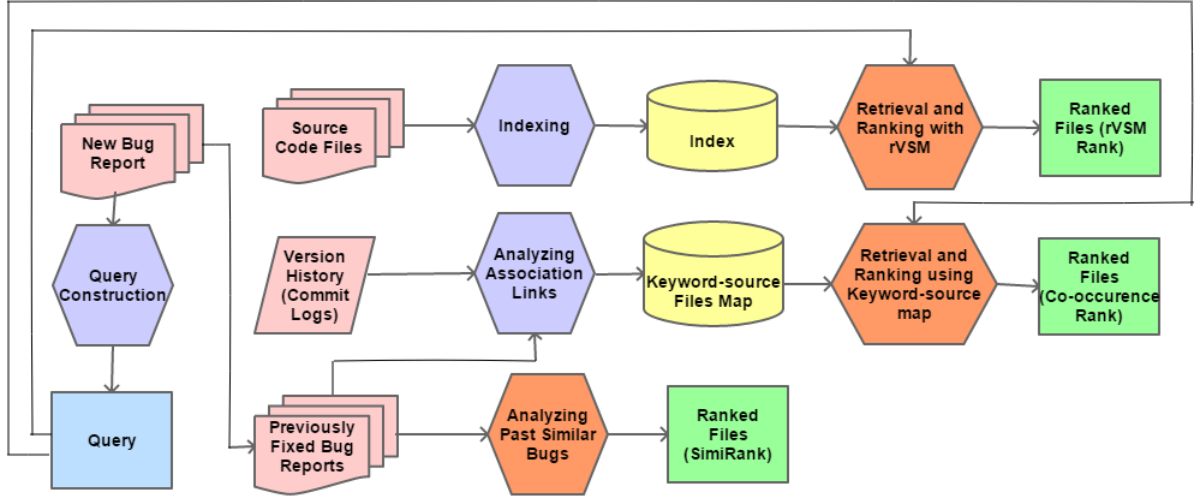
Figure 3: Proposed System Diagram: Bug Localization

Algorithm 2 Recommendation of Buggy Source Code Files

1: **procedure** REFORMULATED QUERY $(Q') \triangleright Q'$: a reformulated query
2:     $RESULT \leftarrow \{\}$ $\triangleright$ recommended source files
3:     $\triangleright$ retrieving source files from keyword-source file map
4:     $SF_{Q'} \leftarrow$ getAllSourceCodeFilesForQuery(MAP_{KS})
5:     **for** Source Files $SF_{Q'_i} \in SF_{Q'}$ **do**
6:        $K_i \leftarrow$ retrieveKeywordsForSourcFile (MAP_{KS})
7:        $K_{Q'_i} \leftarrow$ getKeywordsFromReformulatedQuery(Q')
8:        $\triangleright$ contextual similarity between keywords
9:        $S_{cos} \leftarrow$ getCosineSimilarity$(K_i, K_{Q'_i})$
10:       $R_{SF'_Q} \leftarrow S_{cos}$
11:     **end for**
12:     $\triangleright$ ranking and selection of candidates
13:     $SR'_Q \leftarrow$ sortCandidates(R_{SF})
14:     $SR_{Top'_Q} \leftarrow$ selectTopK(SR'_Q)
15:     $\triangleright$ collecting top k result
16:     $RESULT \leftarrow SR_{Top'_Q}$
17:     **return** $RESULT$
18: **end procedure**

Calculating Co-occurence Scores 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. However, when we analysis these links for all keywords in a query, a relevant file can be found more than once. So we normalize the frequency of source files using standard TFIDF normalization technique. Then we recommend first Top-K files with their CoOccScores. The equation for computing co-occurence score is given belows:

$$SimiScore = \sum_{AllS_i thatconnecttoW_j} (Link(Wj, S_i)) \quad (6)$$

Here, the links between keyword and source file is 1 if they are connected in the association map and 0 otherwise.

Table 1: Description of Data Sets

Project Name	#Source Codes	#Bug Reports
Eclipse Platform Ant	6085	11732

5.2 Localizing Buggy Source Code Files

In this part, we combine existing two approaches with our proposed keyword-source co-occurence relations. One is with rVSM and Simi ranks which are presented in BugLocator and other one is Vector Space Model based technique. So, there are two sub parts of this section.

Combination of rVSM, Simi Rank and Co-occurence Rank: For each query, we compute the rVSM score against all source codes in the database using equation 2 and we also calculate Simi score using equation 3. Then we calculate co-occurence scores for the query using equation 6. We finally combine the three ranks and for that we use a weighting factor *alpha*. The final equation is given in equation 7.

$$FinalScoreApproach1 = (1-\alpha) \times N(rVSMScore) + \alpha \times N(SimiScore) + \alpha \times N(CoOccScore) \quad (7)$$

We work with different values of *alpha*, which are pre presented in the experiment section.

Combinition of VSM and Co-occurence Rank: We compute VSM score using Appache Lucene library. Then we combine that score with our CoOccScore using equation 8.

$$FinalScoreApproach2 = (1-\alpha) \times N(VSMScore) + \alpha \times N(CoOccScore) \quad (8)$$

6. EXPERIMENT AND DISCUSSION

.

6.1 Data Collection

We work with Eclipse data set. We downloaed a git based Eclipse project from git repository [1]. We work with Eclipse Platform UI project. Currently it contains 6085 number of

Table 2: Performance of Bugloactor and proposed technique (rVSM+Simi+Co-Occerence)

# Test Case	Bug Localization Technique	Top 1 %	Top 5 %	Top 10 %	MRR	MAP
1	BugLocator	8.89	24.55	34.32	0.23	0.28
	rVSM+Simi+ Co-Score	14.50	34.91	45.86	0.33	0.41
2	BugLocator	9.9	24.62	34.53	0.24	0.29
	rVSM+Simi+ Co-Score	12.62	39.04	49.85	0.37	0.42
3	BugLocator	7.50	21.32	30.63	0.21	0.24
	rVSM+Simi+ Co-Score	12.01	33.93	48.35	0.34	0.38
4	BugLocator	8.1	21.02	29.13	0.20	0.24
	rVSM+Simi+ Co-Score	15.02	33.63	48.65	0.33	0.42
5	BugLocator	9.6	30.63	42.94	0.29	0.32
	rVSM+Simi+ Co-Score	17.71	43.84	56.45	0.41	0.50
6	BugLocator	10.21	31.53	43.54	0.30	0.33
	rVSM+Simi+ Co-Score	22.22	48.65	58.86	0.45	0.58
7	BugLocator	9.61	30.33	40.84	0.28	0.31
	rVSM+Simi+ Co-Score	18.62	47.15	59.76	0.44	0.53
8	BugLocator	8.4	26.13	39.94	0.26	0.28
	rVSM+Simi+ Co-Score	18.92	46.55	60.36	0.44	0.53
9	BugLocator	11.11	28.83	40.24	0.27	0.32
	rVSM+Simi+ Co-Score	19.82	42.94	56.16	0.40	0.52
10	BugLocator	6.6	20.72	27.93	0.19	0.21
	rVSM+Simi+ Co-Score	9.91	33.33	45.65	0.31	0.33
Average	BugLocator	8.99%	25.87%	36.40%	0.25	0.28
	rVSM+Simi+ Co-Score	16.13%	40.40%	52.94%	0.38	0.46

Table 3: Performance of proposed technique (VSM+Co-Occerence) Ranks

# Test Case	#Bugs for locating issues	Top 1 %	Top 5 %	Top 10 %	MRR	MAP
1	VSM	23.28	45.50	55.82	0.43	0.60
	VSM + Co-Score	25.98	48.82	58.27	0.47	0.66
2	VSM	22.81	45.09	54.91	0.43	0.60
	VSM + Co-Score	26.53	48.81	58.89	0.47	0.67
3	VSM	19.95	39.63	50.27	0.39	0.52
	VSM + Co-Score	21.22	46.42	56.76	0.44	0.57
4	VSM	21.22	42.97	52.52	0.42	0.57
	VSM + Co-Score	21.75	46.68	58.35	0.44	0.59
5	VSM	26.79	51.72	61.80	0.49	0.67
	VSM + Co-Score	29.18	53.31	60.74	0.50	0.72
6	VSM	23.47	49.33	60.27	0.46	0.62
	VSM + Co-Score	24.93	51.72	62.07	0.49	0.67
7	VSM	28.45	52.13	62.77	0.49	0.70
	VSM + Co-Score	33.15	57.29	67.64	0.54	0.80
8	VSM	26.73	47.59	56.68	0.45	0.67
	VSM + Co-Score	32.36	53.58	61.27	0.50	0.77
9	VSM	26.52	49.87	61.27	0.47	0.66
	VSM + Co-Score	27.32	57.82	67.10	0.54	0.73
10	VSM	18.45	38.77	51.07	0.38	0.49
	VSM + Co-Score	20.69	47.75	58.36	0.44	0.57
Average	VSM	23.77%	50.14%	56.74%	0.44	0.61
	VSM + Co-Score	26.31%	51.22%	60.19%	0.48	0.68

Java source codes. These source codes are contained in our source code repository. On the other hand, currently Eclipse Platform UI project contains more than 10K number of bugs where we only work with the bugs which are fixed. We create quires from each bugs considering their title and short summary. All bug reports are collected from *Bugzilla*. 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.

6.2 Cross Validation

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

6.3 Evaluation Metrices

Ton N-Rank: 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.

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*.

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

MAP(Mean Average Precision) Mean average precision for a set of queries is the mean of the average precision scores for each query.

$$MAP = \frac{\sum_{i=1}^{Q} AveP_i}{Q} \qquad (10)$$

Wilcoxon signed-rank test The Wilcoxon signed-rank test is a non-parametric statistical hypothesis test used to compare two related samples, matched samples, or repeated measurements on a single sample to assess whether their population mean ranks differ.

6.4 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.

For checking this, we experimented with several number of fixed bug reports and their change sets, is given in Table..

We compare our proposed bug localization approach with two existing techniques - 1) BugLocaotor which is based on rVSM and Simi scores and 2) VSM which is based on vector space model. In the following two subsections we will describe the performance comparison between these two existing approaches and our proposed approaches

6.5 BugLocator VS Our Proposed Tool

We combine rVSM and simi ranks with our co-occurence rank. Here, co-occurence rank is computed based on keyword-

Table 4: Performance of (rVSM+Simi+Co-Occerence) for different weighting factors

alpha	Bug Localization Technique	Top 1 %	Top 5 %	Top 10 %	MRR	MAP
0.2	BugLocator	8.89	24.55	34.32	0.23	0.28
	rVSM+Simi+ Co-Score	14.50	34.91	45.86	0.33	0.41
0.3	BugLocator	9.9	24.62	34.53	0.24	0.29
	rVSM+Simi+ Co-Score	12.62	39.04	49.85	0.37	0.42
0.4	BugLocator	9.61	30.33	40.84	0.28	0.31
	rVSM+Simi+ Co-Score	18.62	47.15	59.76	0.44	0.53
Average	BugLocator	8.99%	25.87%	36.40%	0.25	0.28
	rVSM+Simi+ Co-Score	16.13%	40.40%	52.94%	0.38	0.46

source code mapping database.In table 2 we, compare the performance of our proposed approach in terms of top 1, 5, 10 rank, MRR and MAP. We can see that our proposed approach outperforms in all cases. For example, our Top-10 performance 52.94% has an improvement than BugLocator (36.40%).

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

6.6 VSM vs Our proposed Tool

We combine VSM score and co-occurence scores in order to produce ranked result. We also compare the performance of baseline VSM technique and our proposed combined approach. The comparison is presented in table 3. We compute Top-1, Top-5, Top-10 performance and MRR and MAP for both approaches. In all cases our proposed approach outperforms VSM-based bug localization approach. The Top-10 performance of our tool is 60.19% whereas it is 56.74% for VSM.

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

6.7 Weighting Factor Analysis

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

Weighting Function for rVSM+Simi+Co-ocuurence Ranking:

Weighting Function for VSM+Co-ocuurence Ranking:

6.8 Method Level Computation

In our proposed techniques, we also implement another important consideration. Our primary intention was to retrieve relevant methods of a class as well as a whole class. But, during implementation, we found that this process is harder for validation. Our validation depends on the git

commit messages and in the git commit commands the change set for a bug are provided just as a file or class level. Which method is changed contained in the change set are difficult to process in the current situation. However, even if we could not validate the method level retrieval, we introduce method level similarity measure in our proposed techniques. The idea is, after

7. THREATS TO VALIDITY

8. CONCLUSION AND FUTURE WORK

References

[1] Eclipse platform ui git project. https://git.eclipse.org/c/platform/eclipse.platform.ui.git/.

[2] 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.

[3] 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.

[4] 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.

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

[6] 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.

[7] 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.

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

[9] 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.