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:

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

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:

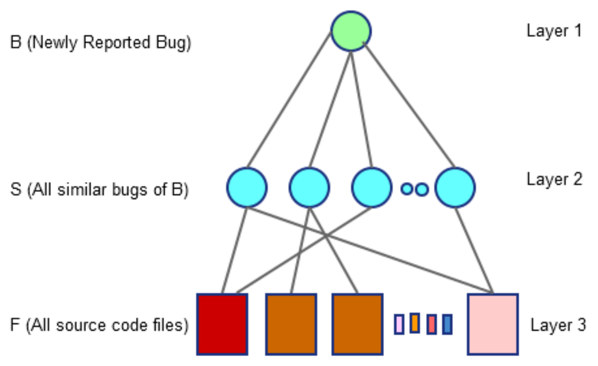

Figure 1: Bug and its Similar Bug Relationship with Source Code Files:

$$SimiScore = \sum_{All S_i that connect to F_j} (Similarity(B, S_i)/n_i) \tag{3}$$

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:

$$FinalScore = (1-\alpha)\times N(rVSMScore)+\alpha\times N(SimiScore) \tag{4}$$

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].

$$Sim(q, d_i) = P(q|d_i) = \prod_{q_k \epsilon q} P(q_k|d_i) \tag{5}$$

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-occicrence 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 preprocessing them such as stop word removal. In version

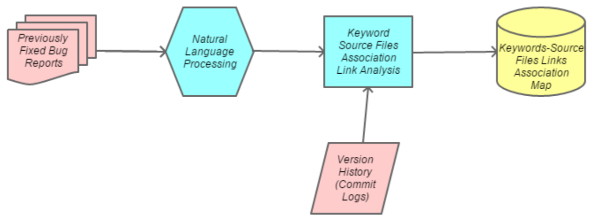
Previously Fixed Bug Reports
Natural Language Processing
Keyword Source Files Association Link Analysis
Keywords-Source Files Links Association Map
Version History (Commit Logs)

Figure 2: Proposed System Diagram: Mapping

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

The system diagram for this part has illustrated in Fig. 3. We have constructed two association map databases from keywords collected from bug report to its source code location information and source code location to their content keywords as described in Section 4. We also have two adjacent list databases both for bug reports and source code files from Section 4. Now we are going to use our formulated query to retrieve relevant source code files. 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 rank all those relevant source code links and this is one part of our predicted locations for buggy source code files. Then we investigate another association map database (i.e., source code-code token links) that results some source code token. These source code tokens are also connected to some other source code files. We retrieve those links and use some heuristics to rank all those relevant files. This is the extended part of our predicted source code locations for a given newly reported bugs.

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

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

```
 1: procedure BUG REPORTS ($BRC$)    ▷ $BRC$: a collection of bug reports
 2:     $MAP_{KS} \leftarrow \{\}$    ▷ an association map
 3:     ▷ creating adjacency map database from the bug reports collection
 4:     $MAP_{adj} \leftarrow$ createAdjacencyDatabase($BRC$)
 5:     ▷ creating a map that links keywords into their bug ids
 6:     $MAP_{kb} \leftarrow$ createKeywordsToBugMaps($MAP_{adj}$)
 7:     ▷ collecting unique keywords from keywords to bug map
 8:     $KB \leftarrow$ collectKeywords($MAP_{kb}$)
 9:     ▷ 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:             ▷ 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:     ▷ collecting all keyword-source files links
19:     $MAP_{KS} \leftarrow MAP[KB]$
20:     return $MAP_{KS}$
21: end procedure
```

other one is between source code links and code token extracted from commit messages and source code content respectively. 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

Algorithm 2 Construction of Association Map Database Between Source Files and Code Tokens

```
1: procedure SOURCE FILES ($SF$) ▷ $SF$: a collection of source code files
2:     $MAP_{FT} \leftarrow \{\}$                    ▷ an association map
3:     ▷ creating adjacency map database from source code files to its code tokens
4:     $SF_{all} \leftarrow$ getAllSourceCodeFiles($SF$)
5:     ▷ Linking files from a project corpus into its code tokens
6:     for Source Files $SF_i \in SF_{all}$ do
7:         $CD_i \leftarrow$ retrieveCodeTokens ($SF_i$)
8:         $CD_{pi} \leftarrow$ preprocessSourceCodeTokens($CD_i$)
9:         ▷ generate a link between source files to its preprocessed code contents
10:        $MAP[SF_i].tokens \leftarrow MAP[SF_i].tokens + CD_{pi}$
11:    end for
12:    ▷ collecting all source files-token links
13:    $MAP_{FT} \leftarrow$ collect ($MAP[KB]$ )
14:    return $MAP_{FT}$
15: end procedure
```

Algorithm 3 Recommendation of Buggy Source Code Files

```
1: procedure REFORMULATED QUERY ($Q'$)▷ $Q'$: a reformulated query
2:     $RESULT \leftarrow \{\}$              ▷ recommended source files
3:     ▷ 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:         ▷ 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:    ▷ ranking and selection of candidates
13:    $SR'_Q \leftarrow$ sortCandidates($R_{SF}$)
14:    $SR_{Top'_Q} \leftarrow$ selectTopK($SR'_Q$)
15:    ▷ collecting top k result
16:    $RESULT \leftarrow SR_{Top'_Q}$
17:    return $RESULT$
18: end procedure
```

or more buggy source code files links and a source code file link can be linked to one or more keywords.

Code Tokens Extraction from Source Codes: We extract source code content from buggy source code files. We perform standard natural language pre-processing on the corpus such as stop word removal, splitting. We remove several source code specific keywords. We split camel case identifier extracted from source code corpus. We also split each word containing any punctuation mark (e.g., ?,:,;), and transforms it into a list of words.

Source Code-Code Token Linking At this point, we create another association map database for source code corpus. To construct this database, we make several links between one or more buggy source code files into code-token extracted from those changed source code files. This association maps buggy source code files into source code tokens. This association map is represented by a bipartite graph, where there is no relationship either among source code files or code-tokens.

5.2 Localizing Buggy Source Code Files

This is the final part of our proposed approach. In this part two sets of predicted buggy source code files are resulted. At first, we search through keyword-source code links association map database using candidate query terms as search query terms. This step predicts some source code links as buggy files, but we need to apply some heuristics to rank those buggy source code files. Finally we can locate top-n source code files as relevant candidates for bug localization. Then we retrieve our second set of predicted source code files. For this, we retrieve code-tokens which are linked to our first set of top-n source code files exploiting source code-code token association map database. This retrieved code token are also linked with some other source code files locations. We apply another heuristics into these code token-source code links and select another top-n relevant source code files as predicted location for a given bug. This second set of source code files are used to locate our extended source code locations that could be relevant fo a given query.

Table 1: Description of Data Sets

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

6. EXPERIMENT AND DISCUSSION

6.1 Big Data Set

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 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.

6.2 Data Collection

We have two parts in our corpus. One is source code files downloaded as git based project and another part is bug reports collection. 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.

7. EXPERIMENTAL RESULTS AND DISCUSSION

7.1 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

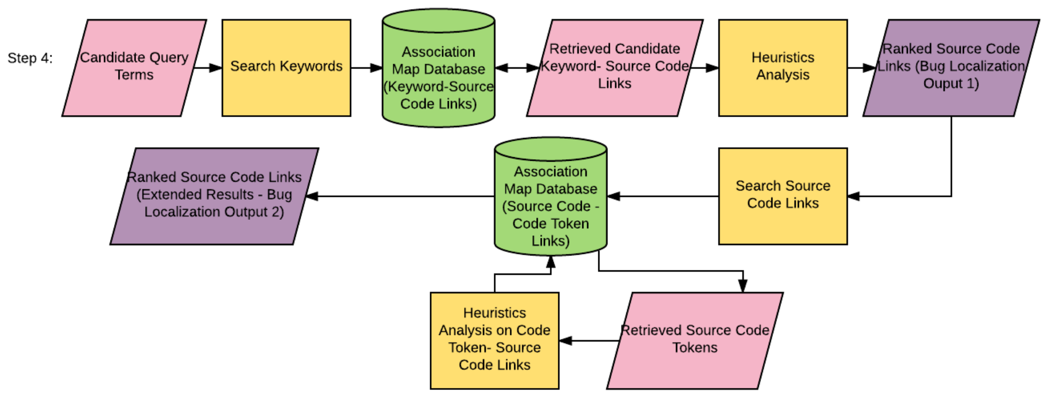
Step 4:
Candidate Query Terms
Search Keywords
Association Map Database (Keyword-Source Code Links)
Retrieved Candidate Keyword- Source Code Links
Heuristics Analysis
Ranked Source Code Links (Bug Localization Ouput 1)
Search Source Code Links
Association Map Database (Source Code - Code Token Links)
Ranked Source Code Links (Extended Results - Bug Localization Output 2)
Retrieved Source Code Tokens
Heuristics Analysis on Code Token- Source Code Links

Figure 3: Proposed System Diagram: Bug Localization

Table 2: The Performance for Top 1, Top 5 and Top 10 for TF-IDF based approach

Year	#Bugs	Top 1	Top 5	Top 10	MRR
2006	1159	16.37%	56%	62%	0.55
2011	377	44%	54%	58%	0.83
2012	404	35%	55%	62%	0.69
2013	507	26%	54%	60%	0.63
Overall	2447	30.34%	54.75%	60.5%	0.675

Table 3: The Performance for Top 1, Top 5 and Top 10 in our proposed approach

#Bugs	Top 1	Top 5	Top 10	MRR
100	23.40%	51.06%	61.70%	0.62
500	19.08%	35.26%	48.55%	0.47
1000	%	%	%	

the reciprocal ranks of results of a set of queries Q.

$$MRR = \frac{1}{|Q|} \sum_{i=1}^{|Q|} \frac{1}{rank_i} \tag{6}$$

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

$$MAP = \frac{\sum_{i=1}^{Q} AveP_i}{Q} \tag{7}$$

7.2 Experimental Results

To compare our proposed approach with two existing bug localization techniques, we implement both existing techniques. The results are presented in table 2, which is experimented for TF-IDF based bug localization approach.

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.

Table 4: The Performance 10 for different size of datasets.

#Bugs	Top 10	MRR
100	36.67%	
200	50%	
300	%	

We also test with gradually increasing dataset. In our proposed approach we create map databases by linking keywords with change set. We believe, as long as the map is bigger, this should be resulting more accurate recommendation. We divide our data set into two parts: in Part I we select a specific number of bug reports for creating map databases, and Part II we test that system with one bug report. For next iteration we add previously tested bug report into Part I and create the map databases again and then test our proposed system with the next immediate bug report. We continue this process as long as required. For example, we first create mapping databases with 100 bug report sequentially and test with 101th bug report as query. For checking this, we experimented with several number of fixed bug reports and their change sets, is given in Table..

We also experiment with several approaches such as considering both mapping databases and cosine relationship between query and source code contents, keeping the information retrieved from mapping database we also include TF-IDF technique between query and recommended source code files. We compare their performances for corpus with several dimensions (i.e., size), and investigate their performances in order to find the best techniques.

7.3 Bug Localization using Mapping Database and Cosine Relationship

In this technique I, we compute two different scores based on keyword-source code mapping database and cosine relationship between query keyword and keyword associated with each recommended source codes. The main objective of our proposed approach is to narrow down the search spaces of information retrieval based bug localization technique. In this approach, we can reduce the number of source code files, which are investigated to locate a newly reported bug. To measure the effectiveness of our approach we calculate the number of files used during bug localization both for our proposed approach and existing bug localization techniques. On the other, we also measure the performance of performance of our proposed approach in terms of top 10 rank and MRR. The performance of our proposed approach are given below.

7.4 Bug Localization Technique using Mapping Database and Source Code's Cosine Relationship

In this technique II, at first we retrieve source codes from keyword-source code mapping database and then for each keyword in the query, we retrieve associated source code links. Then, we compute the cosine relationship between query keywords and all keywords associated with each source code link. The difference between technique I and II are happened in the second stage of score calculation. For technique I, we compute the cosine similarity between query keywords and keywords associated with each recommended source code files from stage one computation. That means, we actually consider those keywords that are globally linked to a source code. On the other hand, in technique II, the cosine similarity measure is computed between query keywords and keywords extracted from each recommended source code extracted in stage one using linked databases. So, here we are considering the source code files as locally.

7.5 Analysis on Required Number of Source Files for Proposed Techniques and Existing Technique

In this section, we note down the number of source code files, which are required to investigate for proposed and existing bug localization techniques. To do so, for each bug localization we count the number of source code files, which were investigated during localizing that bug. Then, we compute average of all result. In the existing literature, the similarity between the query bug report and all source code files are computed for retrieving recommended source files for localizing that bug. When the number of source files are not too many, this similarity measure technique is not an issue. But, the number of files in the source code repository increases over time and makes the computation even harder. In our corpus we have 6085 source files, if we apply TF-IDF technique on our corpus, for each query the content of the query will be compared to all source files, i.e., 6085 number of files. But in our proposed technique, we are narrowing down the search space dramatically. In the first step of our proposed approach, we are mapping query keywords into source files utilizing keyword-source files mapping database. Then for further computation we only consider those source files which are mapped by the query keywords. In Table (), we are showing the number of source file, which are compared with query keywords.

7.6 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

8. RELATED WORK

9. THREATS TO VALIDITY

10. 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.

Table 5: Performance of proposed technique II

#Bugs for developing map databases	#Bugs for locating issues	Top 1 %	Top 5 %	Top 10 %	MRR	MAP
100	100	11.11%	37.78%	48.89%	0.27	0.19
200	100	13.89%	30.56%	44.44%	0.27	0.17
300	100	14.81%	40.74%	51.85%	0.39	0.24
400	100	16.67%	23.33%	76.67%	0.33	0.29
500	100	20%	40%	45.71%	0.43	0.25
600	100	23.33%	53.33%	53.33%	0.58	0.32
Average		13.30%	37.79%	53.48%	0.38	0.24