An Appropriate Method Ranking Approach for Localizing Bugs using Minimized Search Space

Shanto Rahman and Kazi Sakib
Institute of Information Technology, University of Dhaka, Dhaka, Bangladesh

Keywords: Bug Localization, Search Space Minimization, Information Retrieval, Static and Dynamic Analysis.

Abstract: In automatic software bug localization, source code analysis is usually used to localize the buggy code without manual intervention. However, due to considering irrelevant source code, localization accuracy may get biased. In this paper, a Method level Bug localization using Minimized search space (MBuM) is proposed for improving the accuracy, which considers only the liable source code for generating a bug. The relevant search space for a bug is extracted using the execution trace of the source code. By processing these relevant source code and the bug report, code and bug corpora are generated. Afterwards, MBuM ranks the source code methods based on the textual similarity between the bug and code corpora. To do so, modified Vector Space Model (mVSM) is used which incorporates the size of a method with Vector Space Model. Rigorous experimental analysis using different case studies are conducted on two large scale open source projects namely Eclipse and Mozilla. Experiments show that MBuM outperforms existing bug localization techniques.

1 INTRODUCTION

In automatic software bug localization, developers provide bug reports and buggy projects to an automated tool, which identifies and ranks a list of buggy locations from the source code. After getting the list of buggy locations, developers traverse the list from the beginning until they find the actual one. Hence, the accurate ranking of buggy locations is needed to reduce the searching time for localizing bugs.

Automatic bug localization is commonly performed using static, dynamic or both analysis of the source code (Zhou et al., 2012), (Saha et al., 2013). In static analysis, buggy locations are identified by following probabilistic approach which may produce biased results when unnecessary information i.e., whole source code is considered for a bug. Dynamic analysis based techniques analyze the execution trace of the source code with suitable test suite to identify the executed method (Eisenbarth et al., 2003), (Poshyvanyk et al., 2007). Although dynamic analysis provides executed methods call sequence, it cannot extract the method contents which is necessary for Information Retrieval (IR) based bug localization techniques.

Although a large number of literature addresses software bug localization with bug report, to the best of the authors knowledge those do not specifically focus on the minimization of search space. Lukins et al. (Lukins et al., 2008) propose a Latent Dirichlet Allocation (LDA) approach, while Ngyuen et al. customizes LDA by proposing BugScout (Nguyen et al., 2011). Zhou et al. (Zhou et al., 2012) propose *BugLocator* where VSM is modified by proposing tf-idf formulation. An extended version of *BugLocator* is proposed by assigning special weights on structural information (e.g., classes, methods, variables names, comments) of the source code (Saha et al., 2013). *PROMISER* combines both static and dynamic analysis of the source code (Poshyvanyk et al., 2007). Since most of the techniques suggest classes as buggy, developers need to manually investigate the whole source code class to determine more granular buggy locations (e.g., methods). Besides, all the aforementioned techniques consider the whole source code during static analysis, though some techniques incorporate dynamic analysis. As a result, localization accuracy may be biased by unnecessary information.

This paper proposes an automatic software bug localization technique namely MBuM where buggy locations are identified by eliminating irrelevant search space. MBuM identifies a relevant information domain by tracing the execution of the source code for a bug. As dynamic analysis provides a list of executed methods names, static analysis is performed to extract the contents of those methods. Several pre-processing techniques are applied on these relevant source code along with the bug report to produce code and bug corpora. During the creation of bug corpora, stop

Rahman, S. and Sakib, K.
An Appropriate Method Ranking Approach for Localizing Bugs using Minimized Search Space.
In *Proceedings of the 11th International Conference on Evaluation of Novel Software Approaches to Software Engineering (ENASE 2016)*, pages 303-309
ISBN: 978-989-758-189-2

words removal, multiword splitting, semantic meaning extraction and stemming are applied on the bug report. In addition to these, programming language specific keywords removal is applied for generating code corpora. Finally, to rank the buggy methods, similarity scores are measured between the code corpora of the methods and bug corpora by applying mVSM. mVSM modifies existing VSM where larger methods get more priority to be buggy.

In experiments, three case studies are performed where Eclipse and Mozilla are used as the subject. Results are compared with four existing bug localization techniques namely PROMISER, BugLocator, LDA and LSI. In Eclipse, MBuM ranks the buggy method at the first position in three (60%) among five bugs, while other techniques rank no more than one (20%) bug at the top. Similarly in Mozilla, LDA, LSI and BugLocator rank no bug at the top position whereas MBuM ranks three (60%) and PROMISER ranks two (40%) bugs at the first position. Above results show that, MBuM performs better than other existing state-of-the-art bug localization techniques.

2 LITERATURE REVIEW

This section focuses on recent researches which were conducted to increase the accuracy of bug localization (Kim et al., 2013; Zhou et al., 2012; Rahman et al., 2015). The following discussions first hold static analysis and later describe the dynamic analysis based bug localization approaches.

Lukins et al. proposed a LDA based static bug localization technique where word-topic modeling and topic-document distribution were prepared (Lukins et al., 2008). Similarity score was measured between each word of the bug report and topics of the LDA model. By considering previous bugs information, another bug localization technique was proposed in (Nichols, 2010) where method documents were generated which contained the concepts of each method. Latent Semantic Indexing (LSI) was applied on the method documents to identify the relationships between the bug report and concepts of the method documents. This approach may fail due to depending on the predefined dictionary words and inadequate previous bugs. Besides, results may be biased, as the whole source code is considered.

BugLocator measured similarity between bug report and source code where previous bug reports were considered to give previously fixed classes more priority than others (Zhou et al., 2012). As this technique did not consider semantic meanings, exact matching between source code and bug report may not provide accurate results. An improved version of BugLocator was proposed in (Saha et al., 2013) where special weights were assigned in structural information (e.g., class, method, variable names and comments). Due to including programming language keywords, the accuracy may be degraded. For example, if a bug report contains “In dashboard, **public** data cannot be viewed” and a source code class contains a large number of occurrences of **public** as programming language keyword. Besides, all these approaches suggest a list of class level buggy locations where developers need to manually find buggy location in more granular level (e.g., methods).

Poshyvanyk et al. proposed a method level bug localization technique where both static and dynamic analysis were combined (Poshyvanyk et al., 2007). Initially two analysis techniques produced similarity scores differently without interacting with each other and finally calculated a ranking score using the weighted sum of those scores. Although this technique used dynamic information of the source code, it cannot minimize the solution search space rather the whole source code was considered during static analysis. This may increase the biasness, as a consequence the accuracy of bug localization deteriorates.

From the above discussions, it appears that existing bug localization techniques follow static, dynamic or both analysis of the source code. Among those, most of the approaches suggest a list of buggy classes which demand manual investigation into the class file to find more granular buggy locations such as methods. Moreover, none of the IR based bug localization techniques eliminate irrelevant information rather consider whole source code information which degrades the bug localization accuracy.

3 PROPOSED SOLUTION

In this section, Method level Bug localization using Minimized search space (MBuM) is proposed which increases the ranking accuracy by considering only the relevant source code for a bug. Thus false positive rate is minimized. MBuM also ensures that the actual buggy methods must reside within the extracted relevant domain. The overall process of MBuM is briefly discussed as follows.

At the beginning, source code dynamic analysis is performed to minimize the solution space which provides only the relevant methods for generating a bug. Afterwards, static analysis is used to retrieve the contents of those relevant methods. These valid and relevant information of the source code is further processed to create code corpora. Similarly, bug report is

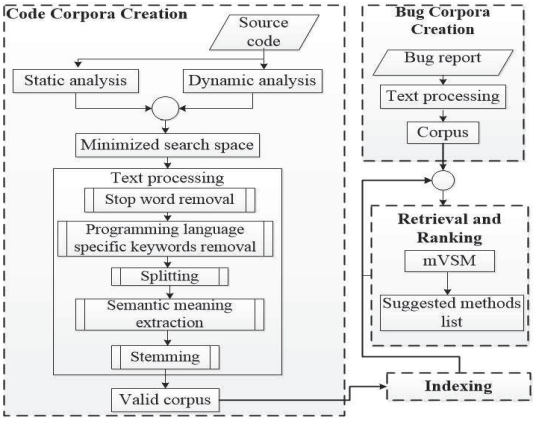


Figure 1: Functional block diagram of MBuM.

processed to produce bug corpora. Finally, similarity between the bug and code corpora is measured using mVSM to rank the source code methods. The overall process of MBuM is divided into four steps and those are Code Corpora Creation, Indexing, Bug Corpora Creation, Retrieval and Ranking where each step follows a series of tasks as shown in Figure 1.

3.1 Code Corpora Creation

Code corpora are the collection of source code words which are used to measure the similarity with bug corpora (Nichols, 2010), (Wang and Lo, 2014). So, the more accurate the code corpora generation is, the more accurate matching can be obtained which increases the accuracy of bug localization. For generating valid code corpora, dynamic analysis is performed followed by static analysis. In dynamic analysis, execution traces are recorded by reproducing the bug using buggy scenario. From this trace, method call graph is generated which does not contain the method contents. The contents of the source code methods are retrieved by parsing the source code using static analysis. This is done by traversing the Abstract Syntax Tree (AST) to extract different program structures such as package, class, method and variable names. This static analysis is conducive to dynamic analysis because it provides the contents of dynamically traced methods, results in minimized search space.

Contents of the minimized search space are processed to generate relevant code corpora as shown in Figure 1. This is needed because source code may contain lots of unnecessary keywords such as programming language specific keywords (e.g., public, static, string), stop words (e.g., am, is). These keywords do not provide any bug related information and so these are discarded from the source code corpora.

Within source code, one word may consist of multiple words such as '*beginHeader*' consists of '*begin*' and '*Header*' terms. Therefore, multiword identifiers are also used for separating these two words. Moreover, statements are splitted based on some syntax specific separators such as '.', '=', '(', ')', '{', '}', '/', etc. Semantic meanings of each word are extracted using WordNet [1] because one word may have multiple synonyms. To describe a single case, the word choice of developers and QA may be different, though the semantic meanings of developers and QA described scenario is same. For example, 'close' word has multiple synonyms such as 'terminate', 'stop', etc. To describe a scenario if a developer uses 'close' but QA uses 'terminate', the system cannot identify these words without using semantic meanings of the words. Hence, semantic meaning extraction plays vital role in accurate ranking of buggy methods.

The last step for generating code corpora is Porter Stemming (Frakes, 1992) which transforms the words to the original form so that 'searching', 'searched' and 'search' are identified as the same word. After completing these, source code corpora are produced.

3.2 Indexing

In this step, the generated code corpora are indexed where packages, classes, methods and method contents are stored according to the structural hierarchy of the program. Each method contains multiple words and those are stored sequentially. For this purpose, LSI is used (Deerwester et al., 1990).

3.3 Bug Corpora Creation

A software bug report contains bug title and description which provide important information about a bug. However, these information may also contain stop words and words may be in present, past or future tense. So, bug report needs to be pre-processed to remove these noisy information. At the beginning of bug corpora generation, stop words are removed from the bug report. Porter stemming is applied (as used for code corpora generation) and valid bug corpora are generated which provide only the relevant words.

3.4 Retrieval and Ranking of Buggy Methods

In this phase, the probable buggy methods are ranked by applying mVSM on the code and bug corpora. We use mVSM where Vector Space Model (VSM) is modified by giving emphasis on large sized methods.

[1] A large lexical database of English, for details - https://wordnet.princeton.edu/

Table 1: The ranking of buggy methods using different bug localization techniques in Eclipse.

#Bug	BugLocator	PROMISER	LSI	LDA	MBuM	Methods
5138	7	2	7	2	1	JavaStringDoubleClickSelector.doubleClicked
31779	4	1	2	2	1	UnifiedTree.addChildrenFromFileSystem
74149	12	5	8	1	1	QueryBuilder.tokenizeUserQuery
83307	6	5	13	7	2	WorkingSet.restoreWorkingSet
91047	4	6	9	5	3	AboutDialog.buttonPressed

In VSM, ranking scores are measured between each query (bug corpora) and methods as the cosine similarity, according to Equation 1.

$$Similarity(q,m) = cos(q,m) = \frac{\overrightarrow{V_q} \times \overrightarrow{V_m}}{|\overrightarrow{V_q}| \times |\overrightarrow{V_m}|} \quad (1)$$

Here, $\overrightarrow{V_q}$ and $\overrightarrow{V_m}$ are the term vectors for the query (q) and method (m) respectively. mVSM uses the logarithm of term frequency. *imf* is also used to give more importance on rare terms in the methods. *tf* and *imf* are calculated using Equation 2 and 3 respectively.

$$tf(t,m) = 1 + \log f_{tm} \quad (2)$$

$$imf = \log(\frac{\#methods}{n_t}) \quad (3)$$

Here, f_{tm} is the number of occurrences of a term (t) in a method m, *#methods* refers to the total number of methods in the minimized search space, and n_t refers to the total number of methods containing the term t. The VSM score is calculated using Equation 4.

$$cos(q,m) = \sum_{t \in q \cap m} (1 + \log f_{tq}) \times (1 + \log f_{tm}) \times imf^2 \times \frac{1}{\sqrt{\sum_{t \in q} (1 + \log f_{tq}) \times imf^2}} \times \frac{1}{\sqrt{\sum_{t \in m} (1 + \log f_{tm}) \times imf^2}} \quad (4)$$

MBuM also considers method length because previous studies showed that larger files are more likely to contain bugs due to having many information of a software (Zhou et al., 2012). The impact of method length is provided by Equation 5.

$$w(terms) = 1 - exp^{-Norm(\#terms)} \quad (5)$$

Here, *#terms* is the number of terms in a method and $Norm(\#terms)$ is the normalized value of *#terms*. The normalized value of a is calculated using Equation 6.

$$Norm(a) = \frac{a - a_{min}}{a_{max} - a_{min}} \quad (6)$$

Where, a_{max} and a_{min} are the maximum and minimum value of a. Now the weight of each method, $w(terms)$ is multiplied with the cosine similarity score to calculate mVSM score which is shown in Equation 7.

$$mVSM(q,m) = w(terms) \times cos(q,m) \quad (7)$$

After measuring mVSM score of each method, a list of buggy methods is ranked according to the descending order of scores. The method with maximum score is suggested at the top of the ranking list.

4 CASE STUDY

In this section, a comparative analysis between MBuM and existing bug localization techniques such as PROMISER, LDA, LSI and BugLocator has been performed by conducting several case studies. Most of the case studies are similar to PROMISER (Poshyvanyk et al., 2007) and LDA (Lukins et al., 2008). Elements of the case studies followed by experimental details, are described in the following subsections.

4.1 Objectives of the Case Studies

Since MBuM performs method level bug localization, methods are chosen as the level of granularity in all the case studies. Documented bugs with corresponding published patches are considered, where each patch specifies the methods which are actually changed to fix a bug. The considered bugs are well-acquainted and reproducible which meet the following criterion.

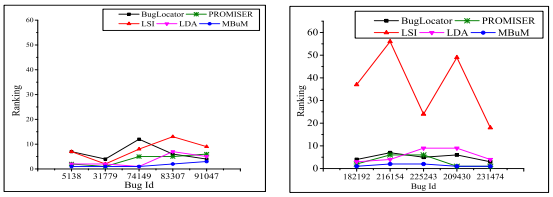
- The bug reports do not contain method and class names directly to prevent the bias.
- The bugs have large similarity with multiple scenarios. For example, Bug #74149 states 'search from help in Eclipse' and there are several packages of the source code related to 'search'.
- The bugs are chosen with published patches.

4.2 Elements of the Case Studies

Two large scale open source projects named as Eclipse and Mozilla are used as the subjects of case studies. Eclipse is a widely used Integrated Development Environment (IDE). For the experimental purpose, version 2.1.0, 2.1.3, 3.0.1, 3.0.2, and 3.1.1 are used, and the volume of each version is too large. For example, version 2.1.3 contains 7,648 classes with 89,341 methods. Another subject Mozilla is a web browser, which is also used for the experiments. Mozilla version 1.5.1, 1.6 (a) and 1.6 are considered, each of which has a large volume of code. For instance, version 1.5.1 contains 4,853 classes and 53,617 methods (Poshyvanyk et al., 2007).

Table 2: The ranking of buggy methods using different bug localization techniques in Mozilla.

#Bug	BugLocator	PROMISER	LSI	LDA	MBuM	Methods
182192	4	2	37	3	1	nsAbAddressCollecter::CollectAddress
216154	7	6	56	4	2	nsMailboxService::NewURI
225243	5	6	24	9	2	nsPostScriptObj::begin_page
209430	6	1	49	9	1	nsPlaintextEditor:DeleteSelection
231474	3	1	18	4	1	Root::MimeObject_parse_begin



(a) Eclipse (b) Mozilla

Figure 2: Ranking provided by different methods.

The actual buggy methods which are identified from the published patches for Eclipse and Mozilla are presented in (Rahman, 2016). In case of more than one published patches for a bug, the union of the most recent and earlier patches are considered. A brief overview of each bug title, description and the generated queries are provided in (Rahman, 2016).

4.3 Case 1: Searching from Help in Eclipse

The goal of this case study is to show the effectiveness of the minimized search space for a specific bug. Bug report #74149 titled as, "the search words after ' " ' will be ignored", tells about searching from 'Help' in Eclipse. In this case, MBuM executes the following scenario to retrieve the relevant methods.

- Expand the 'Help' menu from Eclipse and click on the search option.
- Enter a query within the search field.
- Finally, click on 'Go' button or press enter.

Although Eclipse contains a large number of classes and methods, MBuM finds only 20 classes and 100 methods as relevant to this bug. If a query contains lots of ambiguous keywords, in this case MBuM may suggest the buggy method at most 100^{th} position, while all other existing bug localization techniques suggest buggy method at the $53,617^{th}$ position.

A query is prepared using the bug description which contains 'search query quot token'. After applying query, MBuM suggests the actual buggy method 'org.eclipse.help.internal.search.QueryBuilder.tokenize.UserQuery' at the 1^{st} position of ranking. Same query is applied on PROMISER, LSI and BugLocator to find the buggy location. PROMISER and LSI rank the actual buggy method at the 5^{th} and 8^{th} position respectively. So, comparing with PROMISER and LSI, the effectiveness of MBuM is 5 and 8 times better respectively. On the other hand, BugLocator suggests buggy class at 12^{th} position which shows that MBuM performs m times better, where m represents total number of methods in the suggested first 12 buggy classes. LDA creates a different query as 'query quoted token' which discards the 'search' term from the query due to obtaining 'search' term in multiple packages (Lukins et al., 2008). That is the reason for suggesting the buggy method at the 1^{st} position. Hence, it can be concluded that dynamic execution trace followed by static analysis of the source code improves the ranking of the buggy method.

4.4 Case 2: Bug Localization in Eclipse

This study considers five different bugs in Eclipse, and the details of these bugs are available in (Rahman, 2016). Table 1 presents the ranking of the first relevant method using MBuM as well as PROMISER, LDA, LSI and BugLocator. It is noteworthy, BugLocator suggests classes instead of methods.

The results show that MBuM ranks the actual buggy methods at the 1^{st} position for three (60%) among five bugs. Table 1 and Figure 2 (a) show that for bugs #5138, #83307 and #91047, MBuM performs better than four other techniques. MBuM ranks equal with PROMISER for bug #31779 but ranks better than other techniques. In case of bug #74149, although LDA ranks equal as MBuM, LDA discards the 'search' term. Therefore, it can be concluded that due to performing proper pre-processing for generating valid code corpora, and applying mVSM, the accuracy of ranking buggy methods is improved.

4.5 Case 3: Bug Localization in Mozilla

For this case study, five widely used bugs are also taken from Mozilla bug repository, which are described in (Rahman, 2016). Similar queries are executed by all the techniques and the results demonstrate that MBuM provides better ranking over BugLocator, LDA, PROMISER and LSI techniques.

Table 2 and Figure 2 (b) show that three (60%) out of five bugs are located at the 1^{st} position and another two are ranked at the 2^{nd} position by MBuM. Among other techniques, only PROMISER suggests two (40%) of the five bugs at the 1^{st} position and other three techniques rank no bug at the 1^{st} position (see Table 2). Although for bugs #209430 and #231474, PROMISER provides the same ranking as MBuM, it produces noticeably poor ranking in other three bugs as shown in Figure 2 (b). In case of #182192, #216154 and #225243, MBuM ranks the actual buggy methods more accurately than other four techniques. This comparative analysis of results also shows the significant improvement of ranking by MBuM.

5 THREAT TO VALIDITY

Although MBuM performs better than the existing bug localization techniques, still the improvement of accuracy may be affected by the following reasons.

If the bug report does not contain proper reproducible scenario, it is sometimes difficult to find the accurate execution trace of the source code. However, without reproducing a bug, even a developer cannot manually fix the bug. Besides, if a developer does not follow proper naming conventions, the performance of MBuM may be affected. In practice, developers follow good naming conventions in most of the projects. Moreover, bug report containing inadequate information may affect the performance of MBuM, though it may mislead in manual bug localization.

6 CONCLUSION

This position paper proposes a software bug localization technique named as MBuM where relevant search space is produced by discarding irrelevant source code. For this purpose, source code execution trace is considered. Since bug localization from bug report is an information retrieval based technique, static analysis is applied on the relevant source code and bug reports to create code and bug corpora. Finally, mVSM is applied on those corpora to rank the source code methods. For the purpose of experimentation, case studies are conducted using two large scale open source projects such as Eclipse and Mozilla. These experiments show that MBuM ranks buggy methods at the 1^{st} position in most of the cases.

In this research, although fine grained ranking (e.g., method) is performed, statement level bug localization can be addressed in future. In addition, MBuM may be applied in industrial projects to assess its effectiveness in practice.

ACKNOWLEDGMENT

This research is supported by the fellowship from ICT Division, Bangladesh. No - 56.00.0000.028.33.028.15-214 Date 24-06-2015.

REFERENCES

Deerwester, S. C., Dumais, S. T., Landauer, T. K., Furnas, G. W., and Harshman, R. A. (1990). Indexing by latent semantic analysis. *JAsIs*, 41(6):391–407.

Eisenbarth, T., Koschke, R., and Simon, D. (2003). Locating features in source code. *IEEE Transactions on Software Engineering*, 29(3):210–224.

Frakes, W.B.(1992). Stemming algorithms. pages 131–160.

Kim, D., Tao, Y., Kim, S., and Zeller, A. (2013). Where should we fix this bug? a two-phase recommendation model. *IEEE Transactions on Software Engineering*, 39(11):1597–1610.

Lukins, S. K., Kraft, N., Etzkorn, L. H., et al. (2008). Source code retrieval for bug localization using latent dirichlet allocation. In *15th Working Conference on Reverse Engineering, (WCRE)*, pages 155–164. IEEE.

Nguyen, A. T., Nguyen, T. T., Al-Kofahi, J., Nguyen, H. V., and Nguyen, T. N. (2011). A topic-based approach for narrowing the search space of buggy files from a bug report. In *Automated Software Engineering (ASE), 26th IEEE/ACM International Conference on*, pages 263–272. IEEE.

Nichols, B. D. (2010). Augmented bug localization using past bug information. In *48th Annual Southeast Regional Conference*, page 61. ACM.

Poshyvanyk, D., Gueheneuc, Y.-G., Marcus, A., Antoniol, G., and Rajlich, V. C. (2007). Feature location using probabilistic ranking of methods based on execution scenarios and information retrieval. *IEEE Transactions on Software Engineering*, 33(6):420–432.

Rahman, S. (4/1/2016). shanto-rahman/mbum: 2016. https://github.com/shanto-Rahman/MBuM.

Rahman, S., Ganguly, K., and Kazi, S. (2015). An improved bug localization using structured information retrieval and version history. In *18th International Conference on Computer and Information Technology (ICCIT)*.

Saha, R. K., Lease, M., Khurshid, S., and Perry, D. E. (2013). Improving bug localization using structured information retrieval. In *28th International Conference on Automated Software Engineering (ASE, 2013)*, pages 345–355. IEEE.

Wang, S. and Lo, D. (2014). Version history, similar report, and structure: Putting them together for improved bug localization. In *22nd International Conference on Program Comprehension*, pages 53–63. ACM.

Zhou, J., Zhang, H., and Lo, D. (2012). Where should the bugs be fixed? more accurate information retrieval-based bug localization based on bug reports. In *34th International Conference on Software Engineering (ICSE)*, pages 14–24. IEEE.