An Improved Method Level Bug Localization Approach Using Minimized Code Space

Shanto Rahman(✉), Md. Mostafijur Rahman, and Kazi Sakib

Institute of Information Technology, University of Dhaka, Dhaka, Bangladesh
{bit0321,bit0312,sakib}@iit.du.ac.bd

Abstract. In automatic software bug localization, source code classes and methods are commonly used as the unit of suggestions. However, existing techniques consider whole source code to find the buggy locations, which degrades the accuracy of bug localization. In this paper, a Method level Bug localization using Minimized code space (MBuM) has been proposed which improves the accuracy by only considering bug specific source code. Later, this source code is used for identifying the similarity to the bug report. These similarity scores are measured using a modified Vector Space Model (mVSM), and based on that scores MBuM ranks a list of source code methods. The validity of MBuM has been checked by providing theoretical proof using formal methods. Case studies have been performed on two large scale open source projects namely Eclipse and Mozilla, and the results show that MBuM outperforms existing bug localization techniques.

Keywords: Method level bug localization · Search space minimization · Retrieval and ranking

1 Introduction

In general, bug fixing is initiated when the Quality Assurance (QA) team or user reports against a faulty scenario. Developer receive the reports and try to find the buggy locations into the source code. Generally developers use their experiences on the source code, or debug the code using the debugger of an Integrated Development Environment (IDE). A source code project often contains millions of lines (e.g., Eclipse version 3.0.2 contains 1,86,772 nonempty lines) from which identifying the actual buggy location is always challenging. In case of automatic software bug localization, developers usually provide the bug reports and corresponding buggy project to an automated tool, which provides a ranked list of buggy locations. Developer 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.

Automatic software bug localization is commonly performed using static, dynamic or both analysis of the source code by which failure locations of a software can be identified [1–3]. Most of the bug localization techniques follow

L.A. Maciaszek and J. Filipe (Eds.): ENASE 2016, CCIS 703, pp. 179–200, 2016.
DOI: 10.1007/978-3-319-56390-9_9

static analysis where Information Retrieval (IR) techniques are used [4,5]. Static analysis uses probabilistic approach, so the more the unnecessary information is considered, the more the biasness will be introduced, and the biasness lead to inaccuracies. Dynamic analysis based techniques analyze the execution trace of the source code with suitable test suites to identify the executed methods for a bug. As dynamic analysis only provides method call sequences except method contents, the solution search space has become very small. And using this small search space, it is hard to find the buggy locations.

In recent years, several researches on bug localization have been conducted using static analysis where buggy locations are identified using bug reports and source code [6,7]. In static analysis, authors create two lists of corpora from the source code and bug report. Then corpora are processed so that proper similarities between source code and bug report can be measured. Finally, IR based techniques are applied for ranking probable buggy files [4,5]. Zhou et al. [5] propose such a technique named as BugLocator where Vector Space Model (VSM) is modified by proposing tf-idf formulation. As BugLocator only uses static information of the whole source code, this considers irrelevant information for a bug. An extended version of BugLocator is proposed by assigning special weights on structural information (e.g., class names, method names, variable names and comments) of the source code which also ranks classes as buggy [4]. Similar to BugLocator, this technique also considers the whole source code, as a consequence biasness may be raised. Several dynamic analysis based bug localization techniques have already been proposed [8,9]. Wilde et al. introduce a technique where source code execution traces are considered using passing and failing test cases [8]. However, due to considering passing test cases, irrelevant features may be included in the domain of search space which may hamper the accuracy of bug localization. Poshyvanyk et al. propose PROMISER which suggests methods as buggy by combining both static and dynamic analysis of the source code [9]. Unfortunately, this technique considers whole source code in static analysis which may produce biasness on the ranking.

This paper proposes an a automatic software bug localization technique, namely Method level Bug localization using Minimized code space (MBuM) where buggy locations of the source code are identified by eliminating irrelevant source code (Code space and search space basically represent the same thing. That is why, in this article, code space and search space are used interchangeably; a preliminary version of this work can be found in [10]). At first, MBuM identifies a relevant search space by tracing the execution of the source code for a bug. As dynamic analysis provides a list of executed methods without method contents, static analysis is performed to extract those. Several pre-processing techniques are applied on these relevant source code along with the bug report, which produce code and bug corpora. During the creation of bug corpora, pre-processing techniques such as stop words removal, multiwords splitting, semantic meaning extraction and stemming are applied on the bug report. In addition, programming language specific keywords removal is applied for generating scenario specific code corpora. Finally, to rank the buggy methods, similarity scores are measured

between the code corpora of the methods and bug corpora of the bug report by applying mVSM. It modifies existing VSM by providing more priority to be buggy to the larger sized methods than the small sized methods [5].

The effects of search space on ranking is formulated as a proposition, which has been proved using formal methods. To evaluate the results of the proposed solution, the experiments (i.e., Sect. 5) contain three case studies where Eclipse and Mozilla are used as the subject. Results are compared with four existing bug localization techniques namely PROMISER [9], BugLocator [5], LDA [11] and LSI [12]. In Eclipse, MBuM ranks the actual 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 case of Mozilla, LDA, LSI and BugLocator rank none of the bugs at the top whereas MBuM ranks three (60%) and PROMISER ranks two (40%) bugs at the first position. Above results show that, MBuM outperforms other existing state-of-the-art bug localization techniques.

Rest of the paper is structured as follows. In Sect. 2, existing literature on bug localization are given. Section 3 presents the model of the proposed technique, and the implementation of that model is described in Sect. 4. The result is analyzed in Sect. 5, and later several kinds of threats are discussed in Sect. 6. Finally, Sect. 7 concludes the contribution with future remarks.

2 Related Work

This section focuses on the researches which are conducted to increase the accuracy of bug localization. Following discussion first holds some of the static analysis based bug localization techniques. Later, dynamic analysis based techniques are depicted.

2.1 Source Code Static Analysis Based Techniques

Brent D. Nichols proposes a method level bug localization technique [6] using source code static analysis. At first, the semantic meanings of each method has been extracted by applying several text processing techniques such as stop words removal, separation of identifiers and stemming. In the second phase, authors add extra information from the previous bug history to the methods. When a new bug is arrived, Latent Semantic Indexing (LSI) is applied on the method documents to identify the relationships between the terms of the bug report and the concepts of the method documents. Based on that relationships, a list of buggy methods has been suggested. Due to depending on the predefined dictionary keywords and inadequate previous bug reports, this greedy approach may fail. Furthermore, the accuracy of this technique may not be satisfactory enough due to considering the whole source code information rather than the bug specific information.

Zhou et al. propose BugLocator where buggy locations are identified at the class level using static analysis of the source code [5]. At first authors process bug report and source code separately, resulting two sets of corpora, one for bug report and another for source code. These corpora are processed using several

text processing techniques such as stop words removal, programming language specific keywords removal, multi-words identification and stemming. Later, these two sets of corpora are compared using a revised Vector Space Model (rVSM). For a specific bug, BugLocator suggests a list of classes as buggy where developers need to manually investigate the source code for finding more granular buggy locations (e.g., buggy methods of the source code). As this technique considers whole source code during static analysis, accuracy may be hampered because large unnecessary information creates more biasness in the ranking.

An improved version of BugLocator [5] is addressed (titled as BLUiR) by Ripon et al. where structural information including class names, method names, variable names and comments of the source code get more priority than others [4]. All identifiers and comments are tokenized using above mentioned text processing approaches except the removal of programming language specific keywords. However, most of the cases the consideration of programming language specific keywords may introduce irrelevant information. Along with this, BLUiR may increase unnecessary information by considering whole source code. And these large irrelevant information for a bug may increase false positive rate in the ranking of buggy locations.

Alhindawi et al. [13] introduce another method level feature location based technique by enhancing source code with stereotypes. Stereotypes represent the details of each word which is commonly used in programming. For example, the stereotype named as 'get' means that a method returns a value. Similarly, the stereotype 'set' represents that the value of a data member has been set. In this approach, the corpus of the source code is enriched with the combination of stereotypes which describes the abstract role of the source code method. These stereotype information are derived automatically from the source code via program analysis. After adding stereotype information with the source code methods, Information Retrieval (IR) based technique is used to run queries for feature location. The basic assumption is that adding stereotype information to the source code corpus will improve the results of bug localization.

Wang et al. introduce another bug localization technique, where similar bug reports, version history and structure of the source code are amalgamated [7]. This technique also suggests file level buggy locations and so developers have to spend lots of searching time to identify more granular level (i.e., methods of the source code). Here, the accuracy may be deteriorated significantly because of the consideration of large and irrelevant source code for a bug. Recently, Rahman et al. consider version histories and structural information of the source code to identify buggy files [14]. Here, the scores of rVSM [5] are combined with the frequently changed files information. Later, the source code files whose structural information (such as class names, method names) are available in the bug report get more priority. Based on the above assigned scores, a list of buggy files are ranked. Unfortunately, due to using whole source code in static analysis, the accuracy of this technique may also be biased.

2.2 Source Code Dynamic Analysis Based Techniques

In dynamic analysis based techniques, the run time behavior of the source code is obtained using proper test suits. Using source code dynamic analysis, data flow of the execution are recorded and irrelevant source code are discarded.

The first dynamic analysis based bug localization technique is proposed by Wilde et al. where source code execution traces are analyzed using multiple test cases [8]. Authors consider two types of test cases such as passing and failing test cases. Using the passing test cases, desired features are extracted. Similarly, failing test case provides the features which are not desirable. To identify the buggy locations, these two types of test cases are considered which provide a large volume of features. However, due to using passing test cases, irrelevant features may be included in the domain of search space.

Eisenbarth et al. propose an improved version of [8] where both dynamic and static analysis of the source code are combined [15]. Here, static analysis identifies the dependencies among the data to locate the features in a program while dynamic analysis collects the source code execution traces for a set of scenarios. These traces are analyzed with a view to categorizing the subroutines based on the degree of a feature. However, here during static analysis whole source code is used which degrades the accuracy of the technique.

PROMESIR is another source code dynamic analysis based technique, addressed by Poshyvanyk et al. [9]. Through dynamic analysis, executed buggy methods are extracted for a bug. Meanwhile, static analysis is also applied here which collects the whole source code. Initially, these two analysis techniques produce bug similarity scores differently without interacting with one another. Finally, these two scores are combined and obtained a weighted ranking score for each source code method. Although this technique uses dynamic information of the source code, it fails to discard the irrelevant source code for a bug. Rather the whole source code is considered during static analysis which may increase the biasness. As a consequence, the accuracy of bug localization is declined.

From the above discussions, it is clear that the existing bug localization techniques commonly follow static, dynamic or combination of both analysis of the source code and all of the existing techniques consider whole source code rather than discarding irrelevant source code for a bug. As a result, the accuracies of the existing bug localization techniques hamper significantly.

3 Does Minimized Code Space Can Improve the Accuracy of Bug Localization Techniques?

To answer this question, a model named as Method level Bug localization using Minimized code space (MBuM) has been developed. At first, the elements of the model are defined. Source code and bug report act as the input, while a ranked list of buggy methods is the system output. The input are processed using a bug localizer, and the buggy methods are ranked. During the processing of input, it is assumed that a bug report and source code share some common

information. Since the size of a software project is too large (with respect to the total statements of the source code), it is quite difficult to find the actual buggy locations. The details of the model is described below using Z notations [16].

MBuM

D : *Dictionary*
M : *List of methods*
B : *Bug report*
S : *whole source code*
b_i : *terms of the bug report*
m : *accurate set of buggy methods*
s_i : *terms of the source code*

$B \leftarrow b_i \mid b_i \in D$
$S \leftarrow s_i \mid s_i \in D$
$m \leftarrow$ *find accurate relation between* $b_i \bigwedge s_i$

Here, D represents the set of Dictionary words of bug report and source code. b_i and s_i are the number of the terms of bug report and source code respectively. As the main objective is to increase the suggestion of accurate numbers of buggy methods (m), an accurate relationship should be established between the bug report and source code terms. However, it is difficult to find exact buggy locations from the whole source code, and so removal of irrelevant source code is desired. Moreover, to find the list of buggy methods, a good source code terms and bug report processing technique is needed.

Now, a new proposition is developed by which valid search space can be extracted. The proof of the proposed proposition is described in the followings using Z notations [16].

A small and relevant search space can increase the accuracy of bug localization because localization follows a probabilistic way and accuracy depends on the volume of search space. The relevant search space may be obtained by executing the source code for a specific bug. Since only the bug specific source code is considered, it ensures that the actual buggy methods must reside within the relevant extracted domain. After discarding the irrelevant source code, more accurate bug localization techniques can be obtained. This hypothesis is described in **Lemmas 1** and **2**.

Lemma 1: In bug localization, the selection of relevant domain can produce more accurate ranking than considering the whole-domain.

For developing a software, a large number of source code files or classes are created. The number of selected source code class can significantly affect the ranking score of bug localization. In case of accurate bug localization, to show the number of classes' effects, a representative ranking model namely Vector

Space Model (VSM) can be used [17]. VSM depends on the inverse document frequency (idf), and idf also depends on the number of documents or source code files which is used to increase the weights of rare terms as Eq. 1.

$$idf = \log(\frac{\sharp docs}{n_t}) \tag{1}$$

Here, $\sharp docs$ and n_t are the total number of documents and the number of documents containing the term t respectively. Equation 1 shows that idf increases with the increment of $\sharp docs$. The VSM depends on the idf and the final score of VSM is calculated using Eq. 2.

$$VSM(q,d) = cos(q,d) = (\sum_{t \in q \cap d} (\log f_{tq} + 1) \times (\log f_{td} + 1) \times idf^2) \times \frac{1}{\sqrt{\sum_{t \in q}((\log f_{tq} + 1) \times idf)^2}} \times \frac{1}{\sqrt{\sum_{t \in d}((\log f_{td} + 1) \times idf)^2}} \tag{2}$$

In Eq. 2, t, q and d represent the term, query and document respectively. f_{tq} and f_{td} are the term frequencies within the query and documents respectively. In this study, inverse method frequency (imf) has been used instead of idf to give more priority to rare terms in methods. So, the consideration of large number of irrelevant methods can deteriorate the ranking scores significantly. The effect of imf is illustrated by the following mathematical model when whole source code is considered for finding buggy locations.

RareTermPriority

$weight(x) : x$ *gets weight*
$r? :$ *Rare Term*
$b? :$ *Bug Report*
$t? :$ *Term*
$\upsilon :$ *Buggy Methods*
$\psi :$ *Non Buggy Methods*

$\exists\, t? \in b? : (t? \in r?) \wedge (t? \in \upsilon) \wedge (t? \notin \psi)$
$\bullet\ weight(\upsilon) \Rightarrow rank_{high}(\upsilon)$ (a)
$\exists\, t? \in b? : (t? \in r?) \wedge (t? \notin \upsilon) \wedge (t? \in \psi)$
$\bullet\ weight(\psi) \Rightarrow rank_{high}(\psi)$ (b)

The above model shows the impact of rare terms in bug localization. Here, two scenarios may be occurred such as $RareTermPriority(a)$ and $RareTermPriority(b)$ which represent the weight of buggy and non-buggy methods respectively. The detail is described below.

1. The methods which hold rare terms and related to a bug, get more priority to be buggy which is desired (shown in $RareTermPriority(a)$).

2. Similarly, rare terms exist in methods which have no relation with the occurrence of reported bugs, get more *imf* weights which is repulsive (shown in *RareTermPriority*(b)). When the solution search space is large, the situation of *RareTermPriority*(b) may be occurred which deteriorates the ranking accuracy significantly. So, if the total number of methods can be restricted by only considering valid and relevant methods, *imf* cannot create large negative impacts on the ranking.

Another problem may be occurred due to considering large solution search space and that is the actual buggy location may be suggested in the $T th$ position in the worst case where the total number of available methods are T. It is noteworthy that an automated bug localization technique provides a list of buggy locations according to the descending order of ranking scores, where developers traverse from the beginning of the suggested list one by one. If the actual buggy location is suggested in $T th$ position, the developers need to inspect T number of suggestions to find the actual one. On the other hand, if it can be ensured that the bug is obtained in the targeted domain (e.g., $T - \psi$), developers have to inspect only $(T - \psi)$ buggy locations in the worst case.

SearchSpaceMinimization

$b? : Bug\ Report$
$m_t? : Total\ methods\ in\ source\ code$
$m_d? : Methods\ relevant\ to\ bug$
$m_u? : Methods\ irrelevant\ to\ bug$
$\vartheta : seq\ bugs$
$\kappa! : Minimized\ search\ space$

$$m_d? \in \mathbb{P}\, m_t?$$
$$m_d? \subseteq m_t?$$
$$m_d? = m_t? \setminus m_u?$$
$$m_t? = m_d? \cup m_u?$$
$$\forall\, b? \in \vartheta \bullet (b? \in m_d?) \wedge (b? \notin m_u?) \Rightarrow \kappa! = m_d? \qquad (c)$$

In the above *SearchSpaceMinimization* model, b is a bug report. m_t, m_d and m_u represent total, relevant and irrelevant methods of source code for a bug respectively. This model states that the relevant methods can be obtained by discarding the irrelevant methods from the source code. From this model, it is clear that the bug must reside into the dynamically traced methods m_d according to *SearchSpaceMinimization*(c).

Lemma 2: Large information domain can increase False Positive Rate of bug localization

In case of software bug localization, False Positive Rate (FPR) means that the identification of non-buggy methods as buggy which misguides developers to identify buggy methods. The large unnecessary information can produce large FPR. The situation of the increment of FPR with respect to the unnecessary information is illustrated using the following mathematical model.

In this model, m is a module which is common within p_1, p_2 and p_3 packages of a project whose basic functionalities are same but implementations are different. b is a bug associated to m, which is actually related to package p_1. Due to obtaining the same feature in three different packages, it may happen that p_2 and p_3 get more ranking scores than p_1. This situation could be raised when the whole source codes have been considered to locate a single bug because the size (that is, terms) of p_2 and p_3 may be larger than p_1. And according to VSM, the small but relevant document will get more priority than others which leads to the following theorem.

FPR

$Package : \{p_1, p_2, p_3, ..., p_n\}$
$p? : Package$
$m? : Module$
$b? : Bug\ report$
$Buggy(x) : x\ is\ buggy$
$pr_{rank}(x) : Ranking\ score\ of\ x$
$f(p?) : Feature\ of\ p?$
$Imp(p?) : Implementation\ of\ p?$
$\beta! = Buggy\ package$

$m? \in f(p_1) \cap f(p_2) \cap f(p_3)$
$b? \in m? \wedge b? \in p_1$
$(f(p_1(m?)) = f(p_2(m?)) = f(p_3(m?))) \wedge$
$(Imp(p_1)! = Imp(p_2)! = Imp(p_3)) \Rightarrow$
$(pr_{rank}(p_2) \geq pr_{rank}(p_1)) \vee (pr_{rank}(p_3) \geq pr_{rank}(p_1))$
$\beta! = p_2 \vee p_3$

Theorem 1: A small but relevant search space can increase the accuracy of bug localization.

Lemma 1 illustrates that *VSM* may incur negative impacts in case of large solution search space and **Lemma 2** shows the effects of using whole source code in case of FPR. From **Lemmas 1** and **2**, it can be derived that a small number of relevant search space can increase the accuracy of bug localization.

4 Method Level Bug Localization Using Minimized Code Space

In this section, a model described in Sect. 3 has been implemented by devising a methodology which increases the ranking accuracy because of considering only relevant search space from the source code for a bug. The methodology of the proposed bug localization technique is described in the following subsections.

The overall process of improving the bug localization accuracy is briefly discussed as follows. From Sect. 3, it is observed that if relevant information domain

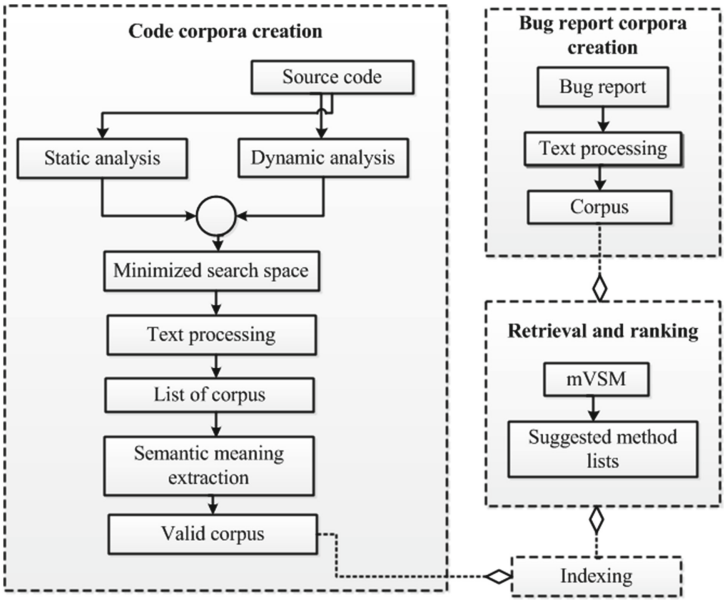

Fig. 1. Functional block diagram of MBuM (N.B. reproduced from [10]).

can be extracted by ignoring irrelevant methods from the large solution space, the accuracy of bug localization can be increased dramatically. At the beginning of bug localization, source code dynamic analysis is performed to minimize the solution search space which extracts only the related methods for generating a specific bug. Static analysis is done with a view to getting the contents of extracted methods. After applying dynamic and static source code analysis, valid and relevant information can be obtained. The contents of extracted methods are processed to create code corpora using static analysis. Since bug report only contains textual information related to a specific buggy scenario, again static analysis is performed to process the bug report. Finally, generated corpora from the bug report and source code are matched with each other to rank the source code methods. The whole process for localizing bugs can be divided into four steps and those are Code corpora creation, Indexing, Bug corpora creation and Retrieval and ranking. Each of the steps follow series of tasks as shown in Fig. 1.

4.1 Code Corpora Creation

Code corpora are the collection of source code words which are used to check the similarity with bug report corpora [4–7]. So, the more accurate the code corpora generation is, the more accurate matching can be obtained which may increase the accuracy of bug localization. For generating valid code corpora, two approaches are conducted and those are dynamic and static analysis [18]. Dynamic analysis produces relevant search space by considering source code

execution trace (e.g., source code methods) for a specific bug by despising the codes which are not responsible for generating the bug. Although dynamic analysis provides relationships between methods or classes, it cannot provide method or class contents. On the other hand, static analysis is related to the code analysis which considers whole source code information and extracts method contents. So, to get the method contents for only relevant methods, the output from dynamic and static analysis are combined and the common methods of those analysis are considered. The code corpora creation can be divided into multiple granular levels which are described below.

For the purpose of dynamic analysis, initially developers need to reproduce the bug after getting the information from the bug report's title, summary and description. Here, execution traces are recorded and analyzed to extract executed methods. From these traces, method call graphs are generated and parsed to obtain the structure of the executed source code. It is noteworthy that the method call graph does not contain the method contents rather it stores the sequentially executed method names. Hence using static analysis, source code is parsed by maintaining the code structures. This is done by traversing the Abstract Syntax Tree (AST)[1] to extract different program structures such as package, class, method and variable names.

Contents of the above minimized search space are processed to get the relevant code corpora as shown in Fig. 1. This is needed because buggy locations are identified by measuring similarity between the contents of bug report and minimized search space. So, trade-off is needed between bug report and source code contents such as the format of all words should be the same (e.g., base form). Minimized source code are pre-processed because source code may contain lots of unnecessary keywords such as programming language specific keywords (e.g., public, static, void, int, string, etc.), stop words (e.g., has, is, a, the, etc.) which do not provide any bug specific information rather may create impacts on ranking and thus stop words are discarded.

Within source code, one word may consist of multiple terms such as '*beginHeader*' consists of 'begin' and 'Header' terms. Therefore, multiword identifiers are also used for creating singular value decomposition. Porter Stemming [19] is applied to get the original form of the word so that 'searching', 'searched' and 'search' are identified as the same word. Moreover, statements are splitted based on some syntax specific separators such as '.', '=', '(', ')', '{', '}', ';', '/', etc. After completing all the aforementioned pre-processing, source code corpora are produced.

The last step for generating code corpora is semantic meaning extraction as shown in Fig. 1. During this step, semantic information of each word is extracted because one word may have multiple synonyms. For example, to describe a single case, developers and QA teams often use different words. Although the semantic meanings of developers and QA described scenarios are the same, the only difference is in their vocabulary choice. Bug localization usually follows IR based

[1] Abstract Syntax Tree, for details - https://eclipse.org/jdt/core/r2.0/dom%20ast/ast.html.

techniques by performing word matching. So, the accuracy depends on the matching of the words. For improving the accuracy, semantic word matching is done. For example, 'close' word has many synonyms such as 'terminate', 'stop', etc. To describe a scenario if a developer uses 'close' and QA uses 'terminate', the system cannot identify the similar words without using semantic meanings of those words. Thus semantic meaning extraction plays vital role in accurate ranking of buggy methods.

4.2 Indexing

In this paper, indexing has been performed according to Fig. 2 where within each package, multiple classes are available with different id, and within each class several methods are stayed with unique id. Each method contains multiple words and each word within a method is stored sequentially. Here, the synonymous words of each word are also stored. Later, each method code corpora and bug corpora is compared by searching only the indices.

Figure 2 is an example of a source code index (taken from, Eclipse project). To better understand about the indexing only one package (that is, *org.eclipse.swt.graphics*) contents are expanded. At first, that package is defined and later the class name, method name of the source code are accumulated in Fig. 2. Here, two classes such as Rectangle and Point are available in package *org.eclipse.swt.graphics*. Among these classes, two methods are stored. As this technique provides a method level ranking, the contents of each method is stored within the method.

```
<?xml version="1.0" encoding="UTF-8" standalone="no"?>
<Source_Code>
    <Package name="org.eclipse.swt.graphics">
        <Class name="Rectangle">
            <Method lineNo="66" name="Rectangle">
                <Word content="rectangl"/>
                <Word content="width"/>
                <Word content="width"/>
                <Word content="height"/>
                <Word content="height"/>
                <Word content="rectangl"/>
            </Method>
        </Class>
        <Class name="Point">
            <Method lineNo="53" name="Point">
                <Word content="point"/>
                <Word content="point"/>
            </Method>
        </Class>
    </Package>
</Source_Code>
```

Fig. 2. Example of source code indexing.

4.3 Bug Report Corpora Creation

Software bug report contains the details of a programs' error. The bug report is usually prepared by Quality Assurance (QA) team or users. A software bug report contains bug title, summary and description which provide important information about a bug.

In bug report, the information may contain some common and irrelevant words such as stop words which do not provide any bug specific information rather create biasness for localizing the bugs. Moreover, words of bug report may be in present, past or future tenses. Bug report needs to be processed to remove these noisy information. At the beginning, stop words (e.g., am, is, are, etc.) are removed from the bug report. Then multiword splitting (if needed) and porter stemming [19] are applied (as used for code corpora generation) to get the base form of the words. After completing these pre-processing, valid bug corpora are generated which provide only the relevant words.

4.4 Retrieval and Ranking of Buggy Methods

In this step, each bug corpus is searched in the minimized solution space. For ranking the source code methods, the proposed technique applies modified Vector Space Model ($mVSM$) [5]. $mVSM$ calculates the similarity between each query (bug corpora) and methods as the cosine similarity with their corresponding vector representations according to Eq. 3.

$$Similarity(q, m) = cos(q, m) = \frac{\overrightarrow{V_q} \times \overrightarrow{V_m}}{\mid \overrightarrow{V_q} \mid \times \mid \overrightarrow{V_m} \mid} \tag{3}$$

Here, $\overrightarrow{V_q}$ and $\overrightarrow{V_m}$ are the vectors of terms for the query (q) and method (m) respectively. $\mid \overrightarrow{V_q} \mid \times \mid \overrightarrow{V_m} \mid$ is the inner product of two vectors. Term weight is calculated by multiplying tf (term frequency) and imf (inverse method frequency). $mVSM$ uses the logarithm of term frequency of a method. The imf ensures that rare or unique terms in the methods are given more importance. tf and imf are calculated using Eqs. 4 and 5 respectively.

$$tf(t, m) = \log f_{tm} + 1 \tag{4}$$

$$imf = \log(\frac{\sharp methods}{n_t}) \tag{5}$$

f_{tm} is the number of occurrences of a term (t) in a method (m), $\sharp methods$ 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. MBuM also considers method length because previous studies showed that larger files are more likely to contain bugs due to carrying many features of a software [5]. The function used to model the method length is provided in Eq. 6.

$$g(terms) = \frac{1}{1 + e^{-Norm(\sharp terms)}} \tag{6}$$

$\sharp terms$ is the number of terms in a method and $Norm(\sharp terms)$ is the normalized value of $\sharp terms$. The normalized value of a is calculated using Eq. 7.

$$Norm(a) = \frac{a - a_{min}}{a_{max} - a_{min}} \tag{7}$$

where, a_{max} and a_{min} are the maximum and minimum value of a. Now this normalized value is multiplied with the cosine similarity score to calculate final $mVSM$ score which is calculated by Eq. 8.

$$mVSM(q, m) = g(terms) \times cos(q, m) \tag{8}$$

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

5 Case Study

The effectiveness of MBuM has been evaluated by conducting several research questions followed by multiple case studies. The case studies are similar to the existing bug localization techniques such as PROMISER [9], LSI [12] and LDA [11]. For this purpose, Top N Rank, Mean Reciprocal Rank (MRR) and Mean Average Precision (MAP) are used as the measurement metrics.

5.1 Elements of the Case Studies

Here, two well-known open-source projects named as Eclipse and Mozilla are considered as the subject of case study. Eclipse is a widely used open source Integrated Development Environment (IDE) which is used for developing Java applications. Meanwhile, Mozilla is a web browser which is used in most of the hardware and software platforms [9]. Different versions of Eclipse (e.g., version 2.1.0, 3.0.1, 3.0.2 and 3.1.0) and Mozilla (e.g., version 1.5.1, 1.6 and 1.6 (a)) are chosen which contain large volume of source code. As an example, 12,863 classes and 95,341 methods are available in Eclipse 3.0.2, while Mozilla 1.5.1 contains 4,853 classes and 53,617 methods [9].

5.2 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. The actual buggy classes and methods corresponding to the bugs are identified from the published patches. Theses are used to evaluate the bug localization techniques where each patch specifies which methods were actually changed to fix a specific bug. In case of a bug, more than one published patches, the union of the most recent and earlier patches are considered. A brief overview of bug title, description and the generated queries for Eclipse and Mozilla are provided in [20]. The considered bugs are well-acquainted and reproducible which meet the following criteria.

(i) Bugs are often categorized as resolved or verified or fixed. So, only the valid bugs that have been fixed are considered.
(ii) The bugs having large similarity with multiple scenarios are chosen. For example, a similar feature is implemented in multiple packages where the implementations are different in every packages. Here, a bug may be occurred in one package. This criterion supports **Lemma 2** (Sect. 3).

5.3 Evaluation Metrics

To measure the performance of MBuM, Top N Rank, MRR (Mean Reciprocal Rank) and MAP (Mean Average Precision) are used as metrics. These are widely-used for measuring the effectiveness of a retrieval and ranking system [17,21]. For all of these, the higher the value, the better the performance is. These metrics are briefly described as follows.

(i) Top N Rank: This is the number of bugs that are localized in the top N ranks ($N = 1, 5, 10, ...$ for this system). For an example, $N = 5$ means that the buggy statements ranked within top 5 suggestions. If one of the fixed files of a bug is in the result set, it is marked as localized [14].
(ii) MRR: A reciprocal rank is the multiplicative inverse of the first correct results' rank of a query [14]. For example, if a bug is localized in rank position 4, the reciprocal rank is $\frac{1}{4}$. So, the range of MRR will be $0 \leq MRR \leq 1$. MRR is the average of all the reciprocal ranks of a set of queries. MRR is calculated using Eq. 9. Here, n and r_i are the number of queries and rank of a query i, respectively.

$$MRR = \frac{1}{n} \sum_{i=1}^{n} \frac{1}{r_i} \tag{9}$$

(iii) MAP: MAP indicates how successfully the system is able to locate all the buggy locations unlike MRR [14]. MAP is the mean of the average precision values of a set of query [5].

5.4 Research Questions

In MBuM, probable buggy methods are ranked by conducting static analysis followed by dynamic analysis of the source code. Hence, few research questions have been emerged such as **RQ1** and **RQ2**. **RQ1** is introduced to validate that, the minimization of search space can improve the localizing accuracy. It is also needed to prove that the actual buggy methods get large similarity scores for a bug. To validate this, **RQ2** is introduced.

RQ1: Does the minimization of search space can improve the accuracy of bug localization?

To answer this question, bug report ♯74149[2] has been introduced which searches from 'Help' in Eclipse titled as "the search words after '"' will be

[2] https://bugs.eclipse.org/bugs/show_bug.cgi?id=74149.

ignored". For this case, the following scenario is executed to retrieve the relevant methods.

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

In this case, MBuM finds only 20 classes and 100 methods as relevant to this bug, shown in Fig. 3. Here, two source code packages namely *org.eclipse.help.internal.search* and *org.eclipse.help.internal.base* are executed. Within the first package, 14 classes are available and only 6 relevant classes are extracted from *org.eclipse.help.internal. base* package. Each extracted class also contains one or more methods. After despising the large irrelevant search space, static analysis is applied only on the relevant information. If a query contains lots of ambiguous keywords (e.g., very few bug related information), MBuM may suggest the actual buggy method at most $100th$ position while all other existing bug localization techniques will suggest buggy method in $53,617th$ position in the worst case. This is because by discarding irrelevant source code methods, MBuM only uses these 100 methods for finding buggy methods while other techniques consider total (i.e., 53, 617) methods of Eclipse.

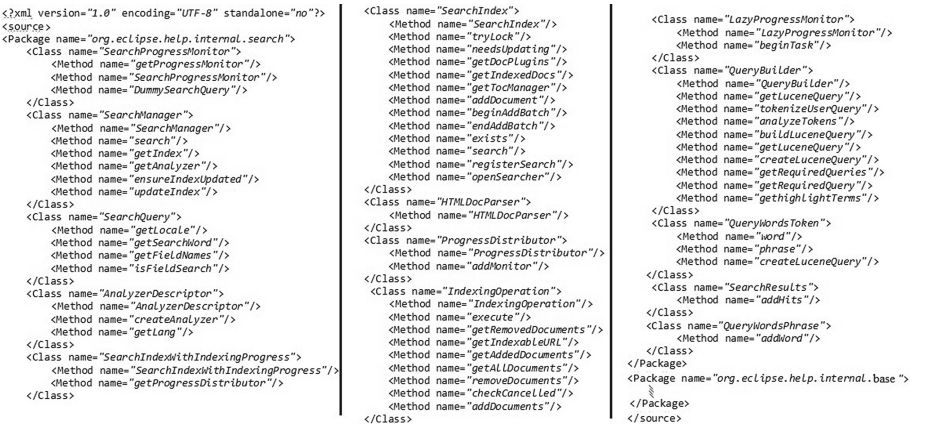

Fig. 3. Extracted methods for Eclipse Bug Id-74149.

A query is formulated using the bug description which contained 'search query quote token'. The actual buggy method is manually retrieved from the published patch which is 'org.eclipse.help.internal.search.QueryBuilder.tokenize UserQuery'. MBuM suggests 'tokenizeUserQuery' method at the 1^{st} position of its ranking (shown in Table 1). Same query is applied on PROMISER, LSI and BugLocator to find the buggy location. PROMISER and LSI rank the actual buggy method at the $5th$ and $8th$ position respectively (shown in Table 1). So,

Table 1. The suggestion of buggy methods using different bug localization techniques in Eclipse (reproduced from [10]).

♯Bug	BugLocator	PROMISER	LSI	LDA	Proposed MBuM
5138	7	2	7	2	1
31779	4	1	2	2	1
74149	12	5	8	1	1
83307	6	5	13	7	2
91047	4	6	9	5	3

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 $12th$ position which shows that MBuM performs m times faster where m represents total number of methods in the suggested 12 buggy classes because BugLocator suggests buggy classes. LDA is not implemented because same bug reports are also used in LDA and that is why the results are taken from that paper [11]. In this case, LDA creates a different query as 'query quote token' which discards the 'search' term from the query due to obtaining 'search' query in multiple scenarios. That is the reason for suggesting the buggy method at the 1^{st} position. Hence, it can be concluded that static analysis followed by dynamic execution trace of the source code reduces the search space which improves the ranking accuracy of bug localization technique.

RQ2: How much effectively MBuM can suggest buggy methods?

The effectiveness of MBuM can be measured by considering the ranking of buggy methods. If the buggy method is ranked at the 1^{st} position, the effectiveness is 100%. To answer **RQ2**, two case studies are conducted on Eclipse and Mozilla where the ranking of buggy methods provided by MBuM are compared with BugLocator, PROMISER, LSI and LDA. These case studies consider five different bugs which were also studied in PROMISER [9] and LDA [11].

Case Study 1: Bug Localization in Eclipse. In this case study, five different bugs in Eclipse are considered for making a comparison with state-of-the-art bug localization techniques named as BugLocator, LDA, PROMISER and LSI. The chosen bugs are described as follows.

- Bug ♯74149[3], titled as "The search words after '" ' will be ignored", exists in the versions 3.0.0, 3.0.1, 3.0.2, and fixed in the version 3.1.1.
- Bug ♯5138[4], titled as "Double-click-drag to select multiple words doesn't work", exists in version 2.1.3 and fixed in the version 3.3.
- Bug ♯31779[5], titled as "UnifiedTree should ensure file/ folder exists", presents in version 2.0.0 and fixed in the version 2.1.0.

[3] https://bugs.eclipse.org/bugs/show_bug.cgi?id=74149.
[4] https://bugs.eclipse.org/bugs/show_bug.cgi?id=5138.
[5] https://bugs.eclipse.org/bugs/show_bug.cgi?id=31779.

– Bug ♯83307[6], titled as "Unable to restore working set item", presents in version 3.1.0 and fixed in the version 3.4.
– Bug ♯91047[7], titled as "About dialog buttons seemingly not responsive", exists in version 3.1.0 and fixed in the version 3.4.

Table 1 presents the ranking of the aforementioned bug localization techniques. It is noteworthy that, BugLocator did not suggest methods rather suggest files or classes. Figure 4 represents the ranking provided by different techniques for five different considered bugs in Eclipse.

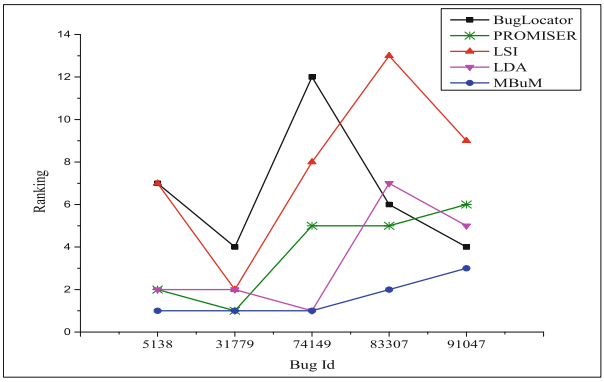

Fig. 4. Ranking provided by different bug localization techniques in Eclipse (reproduced from [10]).

These results show that MBuM ranks the actual buggy methods at the 1^{st} position for three (60%) of the five bugs. Table 1 and Fig. 4 present that for bugs ♯5138, ♯83307 and ♯91047, MBuM performs better than four other techniques. For bug ♯31779, PROMISER only provides equal result as MBuM. In case of bug ♯74149, although LDA ranks equal as MBuM, the term 'search' is omitted from the query which helps for providing better ranking because 'search' is a common term in Eclipse and might create biasness. However, it is not desired because 'search' may be a good candidate for finding buggy locations. From this case study, it can be concluded that the proposed bug localization technique performs better results than others.

Case Study 2: Bug Localization in Mozilla. Similar to the previous case study, here also five mostly used bugs are taken from Mozilla bug repository and the selected bugs are described in [20] and only the title of these bugs are presented in the followings.

[6] https://bugs.eclipse.org/bugs/show_bug.cgi?id=83307.
[7] https://bugs.eclipse.org/bugs/show_bug.cgi?id=91047.

- Bug ♯182192[8], titled as "quotes (') are not removed from collected e-mail addresses", presents in Mozilla version 1.6 and fixed in the version 1.7.
- Bug ♯216154[9], titled as "Anchors in e-mails are broken - clicking anchor doesn't go to target in an email", exists in version 1.5.1 and patched in the version 1.6.
- Bug ♯225243[10], titled as "Page appears reversed (mirrored) when printed", exists in the version 1.6 (a) and fixed in the version 1.7. This bug does not exist in version the 1.6 rather actually presents in the version 1.6(a) [11].
- Bug ♯209430[11], titled as "Ctrl+Delete and Ctrl+BackSpace delete words in the wrong direction", located in version 1.5.1 and fixed in the version 1.6.
- Bug ♯231474[12], titled as "Attachments mix contents", presents in the version 1.5.1 and fixed in the version 1.6.

The results demonstrate that MBuM provides better ranking accuracy over BugLocator, LDA, PROMISER and LSI techniques (shown in Table 2). These results 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. On the other hand, among the other four techniques only PROMISER suggests two (40%) of five bugs at the 1^{st} position and other three techniques' results are far away from the 1^{st} position (according to Table 2).

Table 2. The suggestion of buggy methods using different bug localization techniques in Mozilla (N.B. reproduced from [10]).

♯Bug	BugLocator [5]	PROMISER [9]	LSI [12]	LDA [11]	Proposed MBuM
182192	4	2	37	3	1
216154	7	6	56	4	2
225243	5	6	24	9	2
209430	6	1	49	9	1
231474	3	1	18	4	1

Figure 5 presents the ranking provided by different techniques for five different bugs in Mozilla. 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 Table 2 and Fig. 5. 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.

[8] https://bugzilla.mozilla.org/long_list.cgi?buglist=182192.
[9] https://bugzilla.mozilla.org/show_bug.cgi?id=216154.
[10] https://bugzilla.mozilla.org/show_bug.cgi?id=225243.
[11] https://bugzilla.mozilla.org/show_bug.cgi?id=209430.
[12] https://bugzilla.mozilla.org/show_bug.cgi?id=231474.

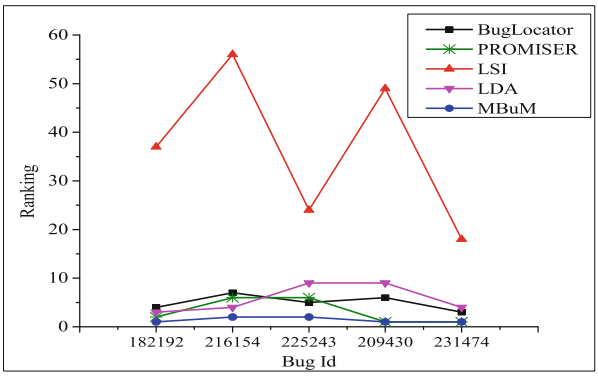

Fig. 5. Ranking provided by different bug localization techniques in Mozilla (reproduced from [10]).

6 Threats to Validity

This section discusses the threats which can affect the validity of the proposed technique. The threats are identified from three perspectives - internal threats, external threats and construct threats.

Internal Threats: The internal threats refer threats that affect the validity of the results which depend on the implementation of the technique and the environmental set up of the experimental procedure. The proposed technique as well as the experimental projects are implemented in Java programming language. Therefore, the result gained through analyzing the experimental projects may differ when experimented in platforms other than java.

External Threats: MBuM requires proper quality of the bug reports. As bug report is one of the important means from which the buggy locations can be identified, the quality of the bug report should contain the bug related information. For example, if a buggy scenario is related to the 'file import' module and the bug report holds another bug modules' information, the quality of the bug report will be significantly deteriorated. In practice, non-informative bug report can also delay to fix a bug. Similarly, if a bug report does not provide enough information, or provides misleading information, the performance of MBuM may be adversely affected. The slight modification is handled by the proposed technique using the semantic meaning extraction from WordNet. However, if the source code is not similar to the bug report, the localization may fail, though it is a common problem in all bug localization schemes.

Finally, if the bug report does not contain proper reproducible approach, it may be hard for developers to find the accurate source code dynamic tracing.

Another factor is the quality of source code, and the accuracy of bug localization depends on the good programming practices in naming variables, methods and classes. If a developer uses meaningless names, the performance of the

proposed technique may be affected. However, in most of the well-managed projects, developers follow good naming conventions and programming practices.

Construct Threats: Construct threats are related to the metrics which are used to analyze the effectiveness of the proposed technique. The results are analyzed based on Top N Rank, MRR and MAP. Therefore, analyzing the results with other metrics can affect the generalization of the results.

7 Conclusion

This paper presents an approach to rank Method level Bug localization using Minimized search space (MBuM). For ranking buggy methods, it discards irrelevant search space by taking the execution trace considering method call sequences of the source code. To retrieve the content of the methods static analysis has been performed. Finally, similarity is measured between the method contents of the source code and bug report which provides a rank list of the methods.

MBuM has been evaluated both theoretically and experimentally. Theoretical evaluation is done using formal methods, and for the purpose of experiments case studies are conducted using two large scale open-source projects named as Eclipse and Mozilla. The case studies show that MBuM ranks buggy methods at the 1^{st} position in most of the cases.

In this research, although fine grained suggestions such as method level bug localization has been conducted, statement level bug localization can be addressed in near future. In addition, since MBuM outperforms other existing techniques for open source projects, it will be applied in industrial projects to assess its effectiveness in practice.

Acknowledgment. This research is supported by the fellowship from ICT Division, Ministry of Posts, Telecommunications and Information Technology, Bangladesh. No - 56.00.0000.028.33.028.15-214 Date 24-06-2015.

References

1. Dit, B., Revelle, M., Gethers, M., Poshyvanyk, D.: Feature location in source code: a taxonomy and survey. J. Softw. Evol. Process **25**(1), 53–95 (2013)
2. Hovemeyer, D., Pugh, W.: Finding bugs is easy. ACM Sigplan Not. **39**(12), 92–106 (2004)
3. Saha, R.K., Lawall, J., Khurshid, S., Perry, D.E.: On the effectiveness of information retrieval based bug localization for c programs. In: IEEE International Conference on Software Maintenance and Evolution (ICSME 2014), pp. 161–170. IEEE (2014)
4. Saha, R.K., Lease, M., Khurshid, S., Perry, D.E.: Improving bug localization using structured information retrieval. In: Proceedings of the 28th International Conference on Automated Software Engineering (ASE 2013) IEEE/ACM, pp. 345–355. IEEE (2013)

5. Zhou, J., Zhang, H., Lo, D.: Where should the bugs be fixed? more accurate information retrieval-based bug localization based on bug reports. In: Proceedings of the 34th International Conference on Software Engineering (ICSE 2012), pp. 14–24. IEEE (2012)
6. Nichols, B.D.: Augmented bug localization using past bug information. In: Proceedings of the 48th Annual Southeast Regional Conference, p. 61. ACM (2010)
7. Wang, S., Lo, D.: Version history, similar report, and structure: putting them together for improved bug localization. In: Proceedings of the 22nd International Conference on Program Comprehension, pp. 53–63. ACM (2014)
8. Wilde, N., Gomez, J.A., Gust, T., Strasburg, D.: Locating user functionality in old code. In: Proceerdings of the Conference on Software Maintenance, pp. 200–205. IEEE (1992)
9. Poshyvanyk, D., Gueheneuc, Y.-G., Marcus, A., Antoniol, G., Rajlich, V.C.: Feature location using probabilistic ranking of methods based on execution scenarios and information retrieval. IEEE Trans. Softw. Eng. **33**(6), 420–432 (2007)
10. Rahman, S., 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, pp. 303–309 (2016)
11. Lukins, S.K., Kraft, N., Etzkorn, L.H., et al.: Source code retrieval for bug localization using latent dirichlet allocation. In: Proceedings of the 15th Working Conference on Reverse Engineering (WCRE 2008), pp. 155–164. IEEE (2008)
12. Deerwester, S.C., Dumais, S.T., Landauer, T.K., Furnas, G.W., Harshman, R.A.: Indexing by latent semantic analysis. JAsIs **41**(6), 391–407 (1990)
13. Alhindawi, N., Dragan, N., Collard, M.L., Maletic, J.I.: Improving feature location by enhancing source code with stereotypes. In: 2013 IEEE International Conference on Software Maintenance, pp. 300–309. IEEE (2013)
14. Rahman, S., Ganguly, K., Kazi, S.: An improved bug localization using structured information retrieval and version history. In: Proceedings of the 18th International Conference on Computer and Information Technology (ICCIT) (2015) (accepted)
15. Eisenbarth, T., Koschke, R., Simon, D.: Locating features in source code. IEEE Trans. Softw. Eng. **29**(3), 210–224 (2003)
16. Woodcock, J., Davies, J.: Using z. specification, refinement, and proof (1996)
17. Manning, C.D., Raghavan, P., Schütze, H., et al.: Introduction to information retrieval, vol. 1. Cambridge University Press, Cambridge (2008)
18. Kim, D., Tao, Y., Kim, S., Zeller, A.: Where should we fix this bug? a two-phase recommendation model. IEEE Trans. Softw. Eng. **39**(11), 1597–1610 (2013)
19. Frakes, W.B.: Stemming algorithms, pp. 131–160 (1992)
20. Rahman, S.: shanto-rahman/mbum: (2016). https://github.com/shanto-Rahman/MBuM. 4/1/2016
21. Pareek, H.H., Ravikumar, P.K.: A representation theory for ranking functions. In: Advances in Neural Information Processing Systems, pp. 361–369 (2014)