International Journal of Software Engineering and Knowledge Engineering
© World Scientific Publishing Company

AUGMENTING BUG LOCALIZATION WITH PART-OF-SPEECH AND INVOCATION

YU ZHOU
College of Computer Science and Technology,
Nanjing University of Aeronautics and Astronautics 210006, China
zhouyu@nuaa.edu.cn

YANXIANG TONG
State Key Laboratory for Novel Software Technology,
Nanjing University 210023, China
tongyanxiang@ics.nju.edu.cn

TAOLUE CHEN
Department of Computer Science,
Middlesex University, London, United Kingdom
t.chen@mdx.ac.uk

JIN HAN
School of Computer and Software,
Nanjing University of Information Science and Technology 210044, China
hjhaohj@126.com

Bug localization represents one of the most expensive, as well as time-consuming, activities during software maintenance and evolution. To alleviate the workload of developers, numerous methods have been proposed to automate this process and narrow down the scope of reviewing buggy files. In this paper, we present a novel buggy source file localization approach, using the information from both the bug reports and the source files. We leverage the part-of-speech features of bug reports and the invocation relationship among source files. We also integrate an adaptive technique to further optimize the performance of our approach. The adaptive technique discriminates *Top* 1 and *Top N* recommendations for a given bug report and consists of two modules. One module is to maximize the accuracy of the first recommended file, and the other one aims at improving the accuracy of the fixed defect file list. We evaluate our approach on six large-scale open source projects, i.e., ASpectJ, Eclipse, SWT, Zxing, Birt and Tomcat. Compared to the previous work, empirical results show that our approach can improve the overall prediction performance in all of these cases. Particularly, in terms of the *Top* 1 recommendation accuracy, our approach achieves an enhancement from 22.73% to 39.86% for ASpectJ, from 24.36% to 30.76% for Eclipse, from 31.63% to 46.94% for SWT, from 40% to 55% for ZXing, from 7.97% to 21.99% for Birt, and from 33.37% to 38.90% for Tomcat.

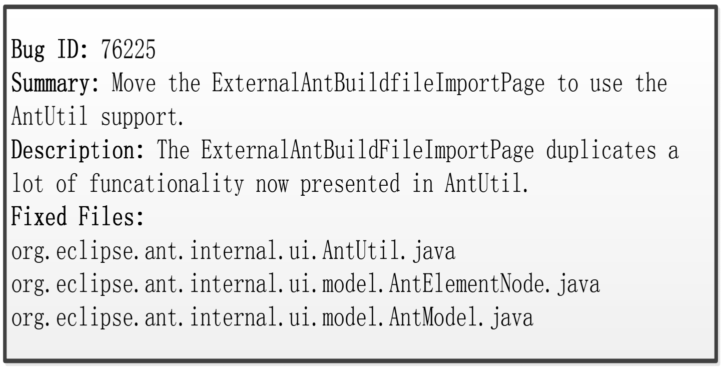
Keywords: software engineering; bug localization; information retrieval; bug report.

1. Introduction

Bug tracking systems (BTS) are a class of dedicated tools to keep track of bug-related issues for software projects. They provide critical supports and are widely used by developers during software development and maintenance phases. Usually, a new software project may set up an account in a robust BTS, such as Bugzilla, to gather potential defects. If multiple shareholders of the software, such as developers, testers or even users, come across a defect, they can resort to the BTS and create an issue report to describe the situation. When a bug report is received and confirmed, it will be assigned to a developer for fixing [37]. The developer must first carefully read the bug report, especially the descriptive parts (e.g., “Summary” and “Description”) and elicit the keywords such as class names or method names, and then review source code files to find and fix the buggy parts. The above activity is indeed time-consuming and tedious, especially for large projects with thousands of source files. Manual localization requires high expertise and imposes a heavy burden to developers, which inevitably hampers productivity. Therefore, it is highly desirable to automate this process and recommend potential buggy source files to developers with a given bug report.

In recent years, some researchers have proposed various approaches to produce a ranking list of buggy files for processing a bug report [33]. The ranking list can narrow down a developer’s search scope and thus help enhance debugging productivity. The basic technique of these approaches is standard information retrieval (IR). It returns a ranking list of buggy files based on the similarity scores between the textual parts of a bug reports and the source code. However, the important information of bug reports does not only come from the textual information, but also from other parts. For example, Sisman et al. extended the IR framework by incorporating the histories of defects and modifications stored in versioning tools [25]. The histories might complement the vague description in the textual parts of the bug reports and improve the accuracy of ranking buggy files. Indeed, the source files are coded in some specific programming language, such as Java or C++, which, compared to natural languages, have different grammatical/semantic features. Therefore, traditional natural language processing techniques from IR field cannot be applied directly to extract the discriminative features of the source code. In light of this, Saha et al. utilized code constructs and presented a *structured* IR based technique [23]. They divided the code of each file into four parts, namely, Class, Method, Variable and Comments. Furthermore, the similarity score between a source file and a bug report was calculated by summing up the eight similarity scores between the source files and bug reports. In [37], Zhou et al. integrated the information of file length and similar bugs to strengthen the traditional Vector Space Model. After that, many other researchers have explored combining other attributes of the bug reports and the source code to further improve the accuracy of bug localization [29, 34, 32].

We observe that most of the existing work, if not all, treats the words (apart from stop words) equally without discrimination. To be more specific, they do not



Bug ID: 76225
Summary: Move the ExternalAntBuildfileImportPage to use the AntUtil support.
Description: The ExternalAntBuildFileImportPage duplicates a lot of funcationality now presented in AntUtil.
Fixed Files:
org.eclipse.ant.internal.ui.AntUtil.java
org.eclipse.ant.internal.ui.model.AntElementNode.java
org.eclipse.ant.internal.ui.model.AntModel.java

Fig. 1: Bug Report example

consider the part-of-speech features of underlying words in the bug reports. The part-of-speech, simply "POS" or "PoS" for short, represents any particular category of words which have similar grammatical properties in nature language, such as noun, verb, adjective, adverb, conjunction, etc. Words with the same part of speech generally display similar behavior in terms of syntax, and play similar roles within the grammatical structure of sentences. In reality, to understand the meaning of a bug report, part-of-speech of each word in a sentence is of particular importance. For example, after traditional IR-based preprocessing, the summary of Eclipse Bug Report #84078: "RemoteTreeContentManager should override default job name" is transformed into "RemoteTreeContentManag override default job name". The noun "RemoteTreeContentManager" directly indicates the buggy file, and the noun phrase "job name" is the substring of a method in the buggy file. By contrast, the verb "override" does not exist in the defect file and the adjective "default" is not a discriminative word for Java code. Thus these words actually provide very little help during debugging.

Textual similarity can indeed help identify potential buggy source files. For example, Figure 1 illustrates a textual snippet of a real bug report (ID: 76255) from Eclipse 3.1 and the bug-fix information. Both the summary and the description focus on the source file "AntUtil.java" and the file is indeed at the first place of the ranking list, but the rest two fixed files "AntElementNode.java" and "AntNode.java" contributing to this defect are at the 4302nd and the 11459th places on the same list ranked solely by similarity [37]. In this case, we observe that most fixed files for the same bug report have invocation relationship between them. For example, the file "AntUtil.java" invokes the other two files. Such underlying logical relationship cannot be captured by the grammatical similarity. This fact motivates us to combine the invocation information with the traditional IR based methods to improve the accuracy of buggy source files identification.

In [15], Kochhar et al. investigated the potential biases in bug localization. They defined "localized bug reports" in which the buggy files have been identified in the report itself. Namely, the class names or method names of the buggy files exist in the bug reports. Motivated by this, in our approach, we filter the source files and only preserve the class names and method names to reduce the noisy localization for the localized bug reports. However, this process also introduces potential issues. If a bug report is a localized one, this method indeed can lift up the rankings of its buggy files. However, this process might also introduce potential problems, i.e., this filtering strategy could also lift other irrelevant files up to the top places as a side effect. Moreover, if a bug report is not a localized one – for example, the bug report does not contain class names or method names but its buggy files are ranked high on the list – this filtering strategy will reduce their rankings.

In light of the above considerations, we need a more comprehensive approach to combine different sources of information to give a more accurate buggy source file localization based on bug reports. We believe that different types of words in bug reports contribute differently to the bug localization process and are worth treating distinctively. Our approach takes the part-of-speech of index terms as well as the underlying invocation relationship into account. In order to take advantages of the localized bug reports and avoid the decrease of global performance, we use different ranking strategies for *Top* 1 and *Top N* recommendations, and propose an adaptive approach, taking the demand of the developers into account.

The main contributions of this paper are as follows:

(1) We propose a part-of-speech based weighting method to automatically adjust the weight of terms in bug reports. Particularly, we emphasize the importance of noun terms. This method sets different weights to terms from the summary and description parts in bug reports in order to distinguish their importance.
(2) We consider the invocation relationship between source code files to lift up the ranking of the files that are invoked by the file mentioned in bug reports with the highest similarity scores. This method can help increase the global performance, like *MRR*[a].
(3) We propose an adaptive approach to maximize the accuracy of recommendations. The approach sets a selection variable *opt* ∈{*true, false*} for users. We conduct a comparative study on the same dataset in [37], which confirms the performance improvement by our approach.

This paper is based on our previous work [27], but with significant extensions. We doubled the size of our empirically studied cases—from three to six open source projects—to minimize the external threats to validity. Moreover, we optimize the process of rendering invocation relationship. In our previous paper [27], we simply used the string-based match to find the invocation files of the highest score file which is easy to implement. However, the performance of this method (implemented in module 2 of our approach; cf. Section 3) is rather poor and the invocation relation

[a] Mean Reciprocal Rank

has to be calculated each time. In order to reduce the overhead, we produce the invocation corpus for module 2 which can be reused once derived.

The rest of the paper is organized as follows. Section 2 describes the background of our work. Section 3 presents the part-of-speech oriented weighting method and our adaptive defect recommendation approach. We experiment with open source data and discuss the results in Section 4. Section 5 and Section 6 give the threat to validity and related work. We conclude the paper in Section 7.

2. Background

2.1. *Basic Ranking Framework*

IR is a process to find the contents in a database related to the input queries. The matching result is not unique, but consists of several objects with different degrees of relevance, forming a ranking list [19]. The basic idea of defect localization using IR is to compute the similarity between textual information of a given bug report and the source code of the related project. It takes the summary and description parts of a bug report as a query, the source files as documents, and ranks the relevance depending on similarity scores.

To identify relevant defect source files, the textual part of bug reports and source code are typically transformed into a suitable representation respecting a specific model. In our approach, we use the *Vector Space Model* (aka *Term Vector Model*) [?] which represents a query or a file as a vector of index terms.

In order to transform texts into word vectors more efficiently, we need to preprocess the textual information. The traditional text preprocessing involves three steps: first, we replace all non-alphanumeric symbols with white spaces, and split texts of bug reports into a stream of terms by white spaces. Second, meaningless or frequently used terms called stop word, such as propositions, conjunctions and articles, are all removed. Usually, the stop word list of the source code is totally different from natural language documents and is always composed of particular words relying on programming languages. Third, all remaining words are transformed into their basic form by the Poster Stemming Algorithm, which can normalize the terms with different forms.

After preprocessing, we take the rest terms of bug reports as index terms to build vector spaces which represent each bug report and source file as vectors. The weight of an index term in a bug report is based on its *Token Frequency (TF)* in the bug report and its *Inverse Document Frequency (IDF)* in the whole bug reports. The same goes for the weight of an index term in a source file. We assume that the smaller the angle of two vectors is, the closer the two documents represented by the two vectors are [8].

2.2. *Part-of-Speech Tagging*

Part-of-speech (POS) tagging is the process of marking up a term as a particular part of speech based on its context, such as nouns, verbs, adjectives, and adverbs, etc. Because a term can represent more than one part of speech at different sentences, and some parts of speech are complex or indistinct, it becomes difficult to perform the process exactly. However, research has improved the accuracy of POS tagging, giving rise to various effective POS taggers such as TreeTagger, TnT (based on the Hidden Markov model), Stanford tagger [4, 10, 12]. State of the art taggers highlight accuracy of circ 93% compared to the human's tagging results.

In recent years, researchers have tried to help developers in program comprehension and maintenance by analyzing textual information in software artifacts [1]. The IR-based framework is widely used and the POS tagging technique has demonstrated to be effective for improving the performance [5, 24]. Tian et al. have investigated the effectiveness of seven POS taggers on sampled bug reports; the Stanford POS tagger and TreeTagger achieved the highest accuracy up to 90.5% [26].

In our study, the textual information of bug reports is composed in natural language. As mentioned before, we have discovered that the noun-based terms are more important for bug localization. Therefore, we have made use of POS tagging techniques to label the terms and adjusted the weight of the terms in vector transforming accordingly.

2.3. *Evaluation Metrics*

Three metrics are used to measure the performance of our approach.

(1) *Top N* is the number of buggy files localized in top N (N=1, 5, 10) of the returned results. A bug is related to many buggy files and if one of the buggy files is ranked in top N of the returned list, we consider the bug to be located in top N. Of course, the higher the metric value is, the better our approach performs.

(2) *MRR (Mean Reciprocal Rank)* is a statistic measure for evaluating the process that produces a sample of the ranking list to all queries. The reciprocal rank of a list is the multiplicative inverse of the rank of the first correct answer. The mean reciprocal rank is the average of the reciprocal ranks for all queries Q:

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

where $rank_i$ is the rank of the first correct recommended file to bug report i and $|Q|$ is the number of all bug reports.

(3) *MAP (Mean Average Precision)* is a global measurement for all of the ranking lists. It takes all of the rankings of the buggy files into account. There are possibly several relevant source code files corresponding to a bug report, the *Average Precision (AP)* for a bug report k can be computed as:

$$AP_k = \sum_{k=i}^{|S|}\frac{P(k) \times pos(k)}{\text{Numbers of Defective Files}} \tag{2}$$

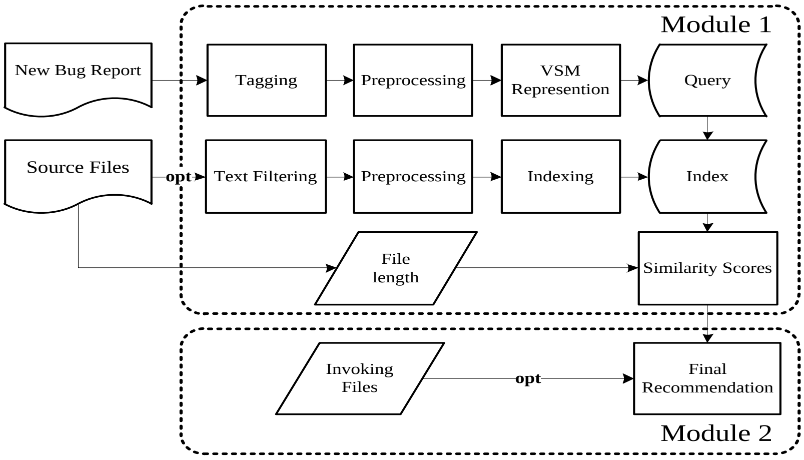


Fig. 2: The Overview of Our Approach

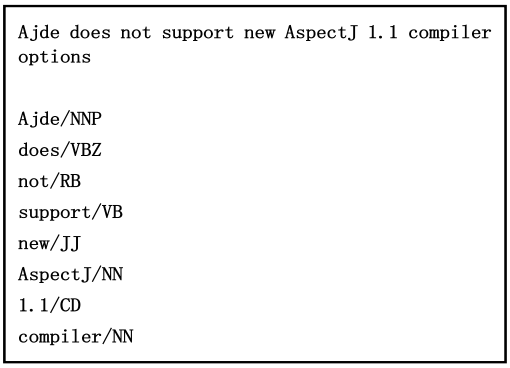
where $|S|$ is the number of source files, and $pos(k)$ is the indicator representing whether or not the file at rank k is a real defect. $P(k)$ is the precision at the given cut-off rank k. MAP is the mean of the average precision values for all bug reports.

3. Approach

Our approach consists of two interconnecting modules and a parameter *opt*. The two modules are:

- *Module 1* is a revised Vector Space Model combining with part-of-speech oriented weighting method. A ranking list for a certain bug report will be produced. In this module, we use a revised Vector Space Model to represent the bug report and index the source code files for similarity calculation. The proposed weighting method was applied automatically to adjust the weight of each term based on its tag. We note that the way of filtering the source code is determined by the parameter *opt*.
- *Module 2* is based on the results of *module 1*. We use the invocation relationship to further augment the accuracy of the results. In this module, we will search the summary and description parts of a bug report for the class-name terms. If the corresponding source files of the class have been ranked high in *module 1*, their invoking files will be raised accordingly in the ranking lists.

The parameter *opt* is a Boolean indicator of our adaptive recommendation depending on the developers' context. If the value of *opt* is set to be true, it means developers want a single decisive, i.e., the most probable file to this bug report; if its value is *false*, it indicates a list of n files would be provided.



```
Ajde does not support new AspectJ 1.1 compiler
options

Ajde/NNP
does/VBZ
not/RB
support/VB
new/JJ
AspectJ/NN
1.1/CD
compiler/NN
```

Fig. 3: The Tagging Results

Our approach mainly uses POS tagging technique to mark up the part-of-speech of each term in bug reports and the invocation relationship between source files can be generated from code comprehension techniques, such as static analysis. Figure 2 gives an overview of our approach. The details will be elaborated below.

3.1. *Module 1 – Similarity Calculation*

In this module, the similarity scores between the new bug report and the candidate source files are calculated, and then an initial ranking list is produced. It's highly important that the part-of-speech must be tagged before the text preprocessing. Namely, the inputs to the POS tagger are all complete sentences. We use the-state-of-the-art POS tagger Stanford-Postagger[b] to mark all of the terms of the bug reports.

Figure 3 illustrates the tagging results for the summary of AspectJ (Bug ID: 29769). The output includes words of the sentences and their parts of speech which have been defined in the English tagging model of Stanford-Postagger. We can see that the words "Ajde", "AspectJ", "complier" and "options" are all noun terms. We duplicate the terms marked as "NN (noun, singular or uncountable)", "NNS (noun, plural)", "NNP (proper noun, singular)" and "NNPS (proper noun, plural)" three times and other terms twice to increase the weights of noun-based terms. Moreover, this weighting strategy wouldn't increase the dimension of Vector Space Model and thus it need not keep the markings until the calculation step. We aim to highlight the nouns comparing to others, thus the weights of the terms with all noun types increase without any difference.

The descriptive parts, i.e., description and summary, of a bug report are regarded as a query, but the significance of these two parts is different [14]. In order to highlight the summary, we follow the heuristics from [30] to increase its terms'

[b] http://nlp.stanford.edu/software/tagger.shtml

frequency twice of that of the description. For source files, we filter the source code before preprocessing, and set the Boolean parameter *opt* to determine what kind of files are recommended. Because the empirical cases studied in our paper are programmed in Java, we leverage API of Eclipse JDT, namely *ASTParser*, to parse the source code. *ASTParser* can analyze the main components of a source file, such as classes, methods, statements and annotations. The source code can be parsed as a compilation unit. By calling the methods of this API, we can remove some useless elements in the source code. In our approach, all annotations of source code are filtered out. Moreover, if the value of the parameter *opt* is set to be true, only class names and method names of the source files will be reserved. We take the filtered source code files as documents and the weight-processed bug reports as queries. In this way, we can build a Vector Space Model to represent both texts based on the index terms of bug reports and source code. The weight $w_{t,d}$ of a term t in a document d is computed based on the *term frequency (tf)* and the *inverse document frequency (idf)*, which are defined as follows:

$$w_{t,d} = tf_{t,d} \times idf_t \tag{3}$$

where $tf_{t,d}$ and idf_t are computed as:

$$tf_{t,d} = \frac{f_{t,d}}{t_d} \tag{4}$$

$$idf_t = log(\frac{n_d}{n_t}) \tag{5}$$

Here, $f_{t,d}$ is the number of the occurrences of term t in document d and t_d is the total number of terms document d includes. n_d refers to the number of all documents and n_t is the number of documents containing term t. Thus, $w_{t,d}$ is high if the occurrence frequency of term t in document d is high and the term t seldom exists in other documents. Obviously, if a term appears 5 times in a document, its importance shouldn't be 5 times compared to the ones appearing once [19]. In view of this point, we use *the logarithm variant* to adjust $tf_{t,d}$ [6]:

$$tf_{t,d} = \log(f_{t,d}) + 1 \tag{6}$$

The similarity score between a query and a document is the cosine similarity calculated by their vector representations computed above:

$$Sim_{t,d} = \frac{\sum_{i=1}^{m} w_{t_i,q} \times w_{t_i,d}}{\sqrt{\sum_{i=1}^{m} w_{t_i,q}^2} \times \sqrt{\sum_{i=1}^{m} w_{t_i,d}^2}} \tag{7}$$

where m is the dimension of the two vectors and $w_{t_i,q}$ (resp. $w_{t_i,d}$) represents the weight of term t_i in query q (resp. document d).

Previous work has shown that large source code files have a high possibility to be defective [21, 35]. Our approach also takes file length into account and sets a coefficient *lens* based on file length to adjust the similarity scores. The range of lengths of source code files is usually large and we must map the lengths to an interval, namely $(0.5, 1.0)$. To this end, we first compute the average length avg of all source files and then calculate the standard deviation sd as:

$$sd = \sqrt{\frac{\sum_{i=1}^{n}(l_i - avg)^2}{n}} \tag{8}$$

where n is the total number of source files. l_i is the length of source code file i. We have an interval $(low, high)$ which is defined as:

$$low = avg - 3 \times sd, high = avg + 3 \times sd \tag{9}$$

and the length l_i of the source file will be normalized as $norm$:

$$norm = \begin{cases} 0.5, & l_i \leq low, & (10) \\ 6.0 \times \dfrac{(l_i - low)}{high - low}, & low < l_i < high, & (11) \\ 1.0, & l_i \geq high. & (12) \end{cases}$$

The coefficient $lens$ is computed as:

$$lens = \frac{e^{norm}}{1 + e^{norm}} \tag{13}$$

Finally, the similarity score between a bug report (the query) and a source code file (the document) can be calculated as:

$$Sim_{t,d} = lens \times \frac{\sum_{i=1}^{m} w_{t_i,q} \times w_{t_i,d}}{\sqrt{\sum_{i=1}^{m} w_{t_i,q}^2} \times \sqrt{\sum_{i=1}^{m} w_{t_i,d}^2}} \tag{14}$$

We then obtain all of the similarity scores of source files and bug report and thus form a ranking list according to the scores.

3.2. *Module 2 – Invocation based calibration*

As usual, the summary only depicts one obvious defect file and seldom contains methods of other buggy files, resulting in poor performance of locating the other hidden buggy files. In order to increase the ranking of all buggy files and improve the overall performance, we also leverage the invocation information between high-ranked buggy files to increase scores of the other buggy files.

The textual information of a bug report has been processed already and may include one or more class-name terms. We define the source files corresponding to the class-name terms as *hitting files*, and the hitting file which ranks the highest on the initial ranking list produced by *module 1* as hf. We hypothesize that the hf has the highest possibility to be the defective source file. Figure 4 shows the detailed processing of the invoking method. First, we extract all class-name terms of a new bug report r and collect the *hitting files* corresponding to these terms. Next, we select the highest ranking source file hf of the *hitting files*. We then review the invocation corpus to find the invocation files. At last, the final score ($FScore_{r,inf}$) of the invoking file inf in *Module 2* is calibrated as follows:

$$FScore_{r,inf} = a \times Sim_{r,hf} + (1 - a) \times Sim_{r,inf} \tag{15}$$

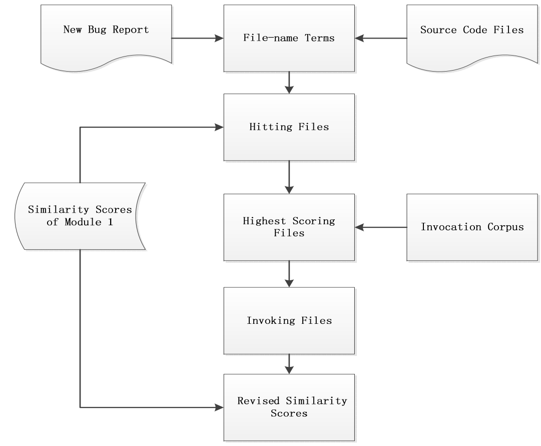


Fig. 4: The Detail of Module 2: Invoking Method

where $Sim_{r,hf}$ is the similarity score between the highest scored file hf of the *hitting file* and the bug report r, and $Sim_{r,inf}$ is the similarity score between the file inf invoked by hf and the bug report r. a is the parameter of the formula which is different in various projects to further adjust the weight of $Sim_{r,inf}$.

The most important part of *module 2* is the invocation corpus which is automatically produced in advance by programs based on API of *Call Hierarchy* in Eclipse. The structure of the invocation corpus of a project is similar to that of its source code. In order to find the invocation files of *hf*, we need to locate the class folder by utilizing the package name of the *hf*. There is a list of method folders under the class folder and there are two main folders in these method folders, namely callers and callees which consist of the invocation information of *hf*. Then, by reading the files of these two folders and extracting the invocation information, we can collect the invocation files of *hf*. The invocation corpus is calculated once and stored as a repository for future use. Module 2 of our approach aims to improve the performance for bug localization by adjusting the similarity scores of invoking files. This invocation method can be combined with most IR-based bug localization approaches, including BugLocator. Of course, the coefficients combining the invocation method and the other two original parts of BugLocator should be updated.

3.3. *Adaptive Strategy*

As mentioned before, *Top 1* recommendation and other *Top N* (e.g., N = 5, 10) recommendations use different identification strategies. We have considered two common situations. If the developers only need a decisive file, the accuracy of *top 1* will get a preferential treatment. In this situation, we remove all of the elements of the source files except for the class names and method names. Otherwise, the developers need *N (for example, N = 5, 10)* candidate files, and thus the overall

Table 1: The Details of Dataset

Projects	#Bugs	#Source Files	Period
AspectJ	286	6485	07/2002-10/2006
Eclipse 3.1	3075	12863	10/2004-03/2011
SWT 3.1	98	484	10/2004-04/2010
ZXing	20	391	03/2010-09/2010
Birt	4166	9765	06/2005-12/2013
Tomcat	851	2174	07/2002-01/2014

performance of *Top N (N = 5, 10)* must be considered and we find that keeping all of the essential elements of the source files except annotation is better.

On top of that, we propose an adaptive approach which can maximize the performance of bug localization recommendation. Our adaptive strategy is based on the analysis of properties in source code files and bug reports, which is implemented by a parameter *opt* set by developers. The parameter controls both the element filtering of source code files and the output of the overall approach shown in Figure 2. When *opt* is set to be *true*, it means developers want a decisive file to the bug, and other elements of source files except for class names and method names must be removed before text preprocessing. The output of our recommendation is then a single file. Otherwise, it means that a list of *N (N = 5, 10)* files would be provided. The output of the our recommendation is then N candidate files accordingly.

4. Experiments

To evaluate our approach, we conduct an empirical study and use the same four cases as in [37], i.e., AspectJ, Eclipse, SWT and ZXing. To demonstrate an even broader applicability, we also include another two cases, i.e., Birt and Tomcat. The information of the dataset is given below in Table 1. We compare our approach with the rVSM model of BugLocator ($\alpha = 0$). BugLocator is an IR-based bug localization approach proposed in [37], it consists of two main parts, i.e., ranking based on source code files and ranking based on similar bugs. The parameter α is the coefficient combining the scores obtained from querying source code files (rVSMScore) and from similar bug analysis (SimiScore). Namely, when α is set to be 0, BugLocator ranks based on rVSMScore solely.

Our experiments are conducted on a PC with an Intel i7-4790 3.6GHz CPU and 32G RAM running Windows 7 64-bit Operating System, and JDK version is 64-bit 1.8.0-65. Table 2 depicts the results achieved by our approach for all of the six projects. If the value of *opt* is set to be *true*, about 114 AspecJ bugs (39.86%), 946 Eclipse bugs (30.76%), 46 SWT bugs (46.94%), 11 ZXing bugs (55%), 916 Birt bugs (21.99%) and 331 Tomcat bugs (38.90%) are successfully located and their fixed files can be found at the *Top* 1 in recommendation. If the value of *opt* is set to be *false*, our approach can locate 76 AspecJ bugs (26.57%), 912 Eclipse bugs (29.66%), 39 SWT bugs (39.79%), 6 ZXing bugs (30%), 382 Birt bugs (9.17%) and 287 Tomcat bugs (33.73%) whose fixed files are at the *Top* 1, 135 AspecJ bugs (47.20%), 1571

Table 2: The Performance of Our Approach

Project	Method	TOP 1	TOP 5	TOP 10	MRR	MAP
AspectJ	opt=true	114 (39.86%)	N/A	N/A	0.44	0.24
	opt=false	76 (26.57%)	135 (47.20%)	168 (58.74%)	0.37	0.21
Eclipse	opt=true	946 (30.76%)	N/A	N/A	0.36	0.23
	opt=false	912 (29.66%)	1571 (51.09%)	1854 (60.29%)	0.40	0.30
SWT	opt=true	46 (46.94%)	N/A	N/A	0.62	0.56
	opt=false	39 (39.79%)	72 (73.47%)	81 (82.65%)	0.55	0.49
ZXing	opt=true	11 (55%)	N/A	N/A	0.69	0.63
	opt=false	6 (30%)	13 (65%)	13 (65%)	0.42	0.36
Birt	opt=true	916 (21.99%)	N/A	N/A	0.25	0.16
	opt=false	382 (9.17%)	851 (20.43%)	1138 (27.32%)	0.15	0.11
Tomcat	opt=true	331 (38.90%)	N/A	N/A	0.47	0.41
	opt=false	287 (33.73%)	489 (57.46%)	554 (65.10%)	0.45	0.41

Eclipse bugs (51.09%), 72 SWT bugs (73.47%), 13 ZXing bugs (65%), 851 Birt bugs (20.43%) and 489 Tomcat bugs (57.46%) whose fixed files are at the *Top* 5 and 168 AspecJ bugs (58.74%), 1854 Eclipse bugs (60.29%), 81 SWT bugs (82.65%), 13 ZXing bugs (65%), 1138 Birt bugs (27.32%) and 554 Tomcat bugs (65.10%) whose fixed files are at the *Top* 10. Besides, the results of *MRR* and *MAP* when *opt* is $true$ are better than the ones when *opt* is $false$ in all of the cases but Eclipse, because the result of *Top* 1 contributes more to the performance of *MRR* and *MAP* than the results of *Top* 5 and *Top* 10, while in Eclipse the difference between the *Top* 1 recommendation is very marginal.

Method 1 defines the process of locating the bugs in our approach when *opt*'s value is true and *Method 2* represents another process of locating the bugs when *opt*'s value is false. *Method 1* takes advantage of the localized bug reports and filters out more noisy data, contributing more to the accuracy of *Top 1* recommendation. From the results of *Top 1* for the six projects with the two methods, we have observed that the results of *Top 1* with *Method 1* are better than the results of *Top 1* with *Method 2* for all of the six projects which confirms the above idea. With the increasing scale of bug reports, the localized bug reports also get increased and play a dominant role in bug localization leading to the better performance of *Top 1*.

Because our approach has filtered the source code in the beginning, particularly when *opt* is $true$, *module 1* seems more time-saving compared to BugLocator without similar bugs module. Table 3 illustrates the execution time of rVSM model and *module 1* of our approach. The execution time of BugLocator $(\alpha = 0)$

Table 3: The Execution Time of BugLocator ($\alpha = 0$) and Module 1 of Our Approach (m: minute; s: second)

Approach \ Projects	AspectJ	Eclipse	SWT	ZXing	Birt	Tomcat
BugLocator	56s	57m	6s	3s	53m	85s
Module 1	49s	9m	12s	6s	8m	40s

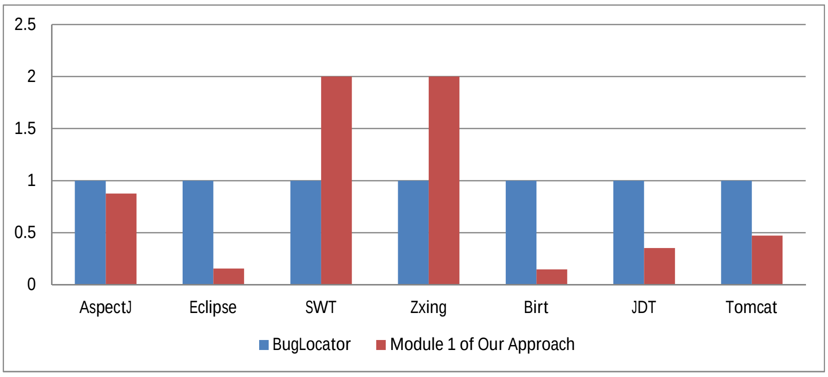
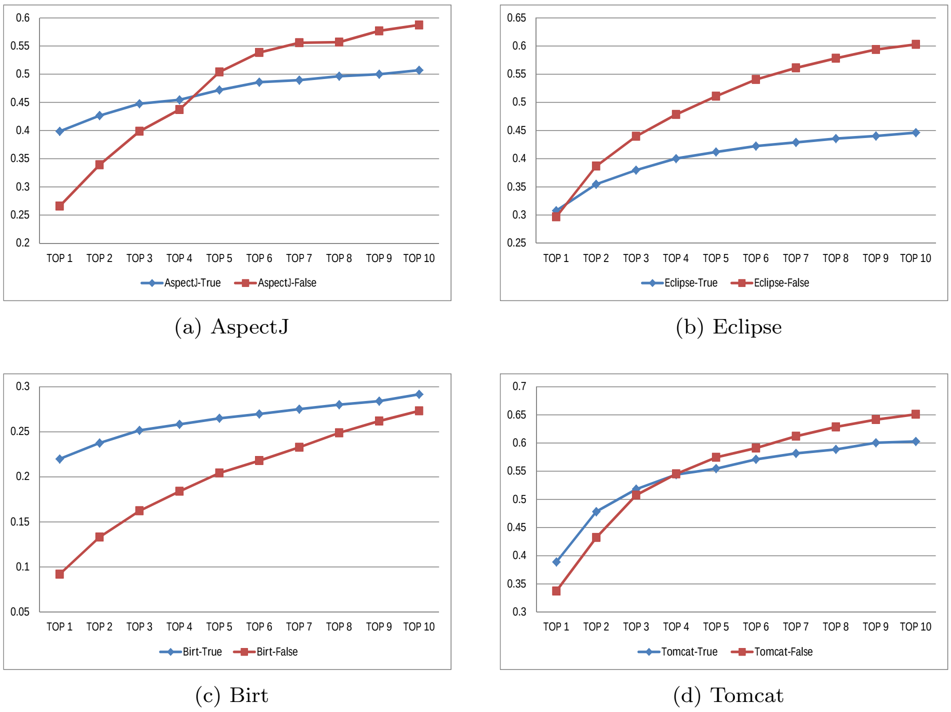


Fig. 5: The Trend of Execution time for The Two Approaches of Comparison

for AspectJ, Eclipse, SWT, ZXing, Birt and Tomcat is 56 seconds, 57 minutes, 6 seconds, 3 seconds and 85 seconds respectively. The execution time of the *module 1* of our approach is 49 seconds, 9 minutes, 12 seconds, 6 seconds and 40 seconds respecitvely. Although the time cost of our approach for SWT and ZXing is higher compared to that of BugLocator, from Table 3, we can find the larger the project is, the better advantage our approach can achieve. Figure 5 pictorially illustrates the execution time comparison of the two approaches. Because the execution time of the six projects is not at the same level, we set the execution time of each project using BugLocator as the unit time 1 and represent the time cost of our approach as the proportion of the execution time of BugLocator. We can discover that the *module 1* relatively decreases the execution time and is more efficient. Moreover, the larger the source code and bug reports are, the more time-saving the *module 1* is.

In our approach, we have made use of the saving time to execute the *module 2* which is considerably time-consuming. It is generally known that extracting the invocation relationship of a large project like Eclipse is very complex and thus costs much time. Although our approach needn't obtain the invocation relationship of all of the source files, it also needs to spend time reviewing thousands of highest scoring source files hf to get the invoking files. But in our approach, this calculation can be conducted once and used in future, since these relationship is stored as a repository.

We have compared the performance of our approach to BugLocator without similar bugs because we try to emphasize the importance of part-of-speech and invocation relationship between source files and don't combine the similar bugs. Table 4 compares the accuracy of our approach with BugLocator. As we can see, the performance of both methods of our approach is better than BugLocator without using similar bugs.



(a) AspectJ

(b) Eclipse

(c) Birt

(d) Tomcat

Fig. 6: The Performance Comparison of Method 1 and Method 2 in Four Cases.

When *opt* is set to be *true*, our approach recommends one file with the highest similarity score to the developers and actually the accuracy of recommended file is sharply high. All of the results have a considerable enhancement. For example, the accuracy of *Top 1* of this method for AspectJ almost improves twice. The performance of *Method 1* are 39.86% for AspectJ compared to 22.73% of rVSM, 30.76% for Eclipse compared to 24.36%, 46.94% for SWT compared to 31.63%, 55% for ZXing compared to 40%, 21.99% for Birt compared to 7.97% and 38.90% for Tomcat compared to 33.37%. Although this method just provides one file, the statistics of *MRR* and *MAP* are based on the ranking lists *Method 1* produces inside. Despite this method sacrifices the results of *top 5* and *top 10*, the metric values of *MRR* and *MAP* are also higher than BugLocator without using similar

Table 4: The Comparison of BugLocator($\alpha = 0$) and Our Approach

Project	Method	TOP 1	TOP 5	TOP 10	MRR	MAP
AspectJ	opt=true	114 (39.86%)	N/A	N/A	0.44	0.24
	opt=false	76 (26.57%)	135 (47.20%)	168 (58.74%)	0.37	0.21
	BugLocator	65 (22.73%)	117 (40.91%)	159 (55.59%)	0.33	0.17
Eclipse	opt=true	946 (30.76%)	N/A	N/A	0.36	0.23
	opt=false	912 (29.66%)	1571 (51.09%)	1854 (60.29%)	0.40	0.30
	BugLocator	749 (24.36%)	1419 (46.15%)	1719 (55.90%)	0.35	0.26
SWT	opt=true	46 (46.94%)	N/A	N/A	0.62	0.56
	opt=false	39 (39.79%)	72 (73.47%)	81 (82.65%)	0.55	0.49
	BugLocator	31 (31.63%)	64 (65.31%)	76 (77.55%)	0.47	0.40
ZXing	opt=true	11 (55%)	N/A	N/A	0.69	0.63
	opt=false	6 (30%)	13 (65%)	13 (65%)	0.42	0.36
	BugLocator	8 (40%)	11 (55%)	14 (70%)	0.48	0.41
Birt	opt=true	916 (21.99%)	N/A	N/A	0.25	0.16
	opt=false	382 (9.17%)	851 (20.43%)	1138 (27.32%)	0.15	0.11
	BugLocator	332 (7.97%)	727 (17.45%)	1003 (24.08%)	0.13	0.09
Tomcat	opt=true	331 (38.90%)	N/A	N/A	0.47	0.41
	opt=false	287 (33.73%)	489 (57.46%)	554 (65.10%)	0.45	0.41
	BugLocator	284 (33.37%)	467 (54.88%)	544 (63.92%)	0.44	0.39

bugs.

When *opt* is set to be *false*, our approach recommends n candidate files based on the ranking list of a bug report to the developers. More defect files ranked at *top N (N=5,10)* may give right inspiration to the developers for finding the location of buggy files. Our approach increases the precision of defect files in *top N (N=5,10)* effectively. The performance enhancement is about 3.84% in *Top 1*, 6.29% in *Top 5* and 3.15% in *Top 10* for AspectJ, about 5.30% in *Top 1*, 4.94% in *Top 5* and 4.39% in *Top 10* for Eclipse, about 8.16% in *Top 1*, 8.16% in *Top 5* and 5.10% in *Top 10* for SWT, about 10% in *Top 5* for ZXing, about 1.20% in *Top 1*, 2.98% in *Top 5* and 3.24% in *Top 10* for Birt and about 0.36% in *Top 1*, 2.58% in *Top 5* and 1.18% in *Top 10* for Tomcat. It is interesting to discover that our approach improve most in *Top 5* on average.

To further explain the performance of the two selective methods in our approach, we extend N to cover more value options, i.e., from 1 to 10. In this experiment, since SWT and ZXing contains relatively smaller number of instances, we conduct the comparison on the rest four projects. Figure 6(a) shows the performance of

AspectJ with 286 bug reports from *Top 1* to *Top 10. AspectJ-True* means *Method 1* and *AspectJ-False* means *Method 2*. It is obvious that the performance of *Method 1* increases sharply at *Top 1* and then slows down. For *Method 2*, the results increase quickly from *Top 1* to *Top 10* at almost the same speed and get better after *Top 5* than *Method 1.*

For the Eclipse project with 3075 bug reports, only the *Top 1* of *Method 1* is still better than the *Top 1* of *Method 2.* The results of *Method 1* from *Top 2* to *Top 10* are all worse than that of *Method 2.* The discovery above is shown in Figure 6(b). As we can see, only the *Top 1* of *Method 1* is better even though the scale of bug reports increase from 286 of AspectJ to 3075 of Eclipse. This is also the case for the Tomcat project, illustrated by Figure 6(d). However, Birt project exhibits different properties. From Figure 6(c), we can observe that the performance of *Method 1* is continuously better than *Mehtod 2*, although the difference between them is decreasing. The fact indicates a converging trend of the two methods. The general suggestion is that, if developers want a recommended file, with our approach they can make use of *Method 1.* If they want *N (N=5,10)* recommended files instead, they should make use of *Method 2.*

5. Threats to Validity

In this section, we discuss the possible threats to the validity in our approach, mainly the concerns of data validity and invocation validity.

(1) *Data Validity.* The experimental dataset we used are all programmed by Java and the keywords of bug reports are mainly class names or method names which make the VSM model more effective than other IR-based models. The performance of *top 1* gets better when we only reserve class names and method names in source code and the results of *top 5, top 10* decrease at this situation and we can get the rule that class names and method names contribute to the results of *top 1.* But we just used the dataset of Zhou et al [37] and two others to assure the fair comparison. Thus, whether or not this heuristic fits all of the Java projects still requires further studied to confirm.

(2) *Invocation Validity.* We generate the invocation corpus by using the JDT's plug-in called Call Hierarchy [20, 16] and search the invocation files from the corpus afterwards. Although the call graph of the projects we use in our experiments is of large scale, especially for Eclipse, and the generation with the large repository can take an additional amount of time, the invocation corpus can be reused once it was produced which seems to be more time-saving in long terms. Moreover, due to the characteristic of the source code, we cannot say that the invocation corpus contains all the invocation files of a particular file. Compared to the simple string-based searching method used in [27], the invocation corpus can avoid re-calculating each time.

6. Related Work

Software debugging is time-consuming but also crucial in software life cycles. Software defect localization becomes one of the most difficult tasks in the debugging activity [31]. Therefore, automatic defect localization techniques that can guide programmers are much-needed. Dynamical bug localization approaches can help developers find defects based on spectrum [2]. A commonly-used method of these approaches is to produce many sets of successful runs and failed runs for computing suspiciousness of program elements via program slicing. The granularity of suspiciousness elements can be a method or a statement. Although the dynamic approach can locate the defect to a statement, the generation of test cases and its selection are also complex [3].

Many researchers have tried to use static information of bugs and source code for coarse-grained localization [18]. They proposed some IR-based approaches combining with some useful attributes of software artifacts and defined the suspicious buggy files depending on the similarity scores between bug reports and source files. Usually, IR-based models are used to represent the textual information of the bug report and source code, such as *Latent Sematic Indexing (LSI)*, *Latent Dirichlet Allocation (LDA)* and *Vector Space Model (SVM)*, which is feasible for numerical calculation [11, 22, 28]. But these works did not consider the POS features of the underlying reports. Gupta et.al [9] attempted to use the POS tagger to help understand the regular, systematic ways a program element is named, but they did not apply the technique to the field of bug localization.

Apart from the efforts in defect localization, there is another thread of relevant work on the bug report classification [36]. Before applying the bug localization techniques, it must be confirmed that the selected bug reports describe the real bugs and then their fixed files are extracted for evaluation, which may save much time and reduce potential noise [15]. A lot of research has been conducted for reducing the noise in bug reports [13]. They used the text of the bug reports and predicted the bug reports to be bug or non-bug with many techniques [7]. Zhou et al. proposed a hybrid approach by combining both text mining and data mining techniques to automate the prediction process [38].

In resent years, Zhou et al. have used the Vector Space Model to represent the texts and taken the length of source files into consideration combining the similar bugs to revise the ranking list. After then many other non-textual attributes are used to enhance the performance, such as version history [25]. Saha et al. found that the code construct is important for bug localization, so they proposed a structure information retrieval approach [23] . Wang et al. combined the above three discoveries to increase the results [29]. Moreover, Ye et al. have used the domain knowledge to cover all accessible features to enhance the IR-based bug location technique [34]. In order to help the developers pick an effectiveness approach proposed in the literature, Le et al. presented the approach APRILE to predict the effectiveness of the localization tools [17].

Our approach leverages nature language processing techniques adjusting the weight of terms depending on their part-of-speech, and takes advantage of heuristics in bug reports to balance the importance of summary and description. Kochhar et al. discovered that the existence of class names in summary or description of bug reports makes contributions to bug localization, which inspires us to propose *Method 1* of our adaptive approach [15].

7. Conclusion and Future Work

In software life cycles, maintenance is the most time-consuming and highly cost phase. An in-time bug fixing is of crucial importance. To mitigate the work of software developers, in this paper, we propose an adaptive approach to recommending potential defective source files given a certain bug report. We take advantages of POS tagging techniques and the logical invocation relationship between source files and present an automatic weighting method to further improve the performance. As far as we know, this is the first work considering the underlying POS features in bug reports for bug localization. The evaluation results on six large open-source projects demonstrate the feasibility of our adaptive approach and also indicate better performance compared to the baseline work, i.e., BugLocator.

In the future, we plan to integrate more features of program to our approach, such as similar bugs, version history and dynamic information. The aim is to propose a more adaptive approach for more complex user demands. More technically, the *module 2* of the our approach will be refined to decrease the number of noisy files, which may produce further enhancement. Moreover, our approach will be expanded to utilize other kinds of dataset, such as bug reports of commercial projects and unresolved bug reports, to demonstrate a broader applicability.

Acknowledgments

The work was partially funded by the Natural Science Foundation of Jiangsu Province under grant No. BK20151476, the National Basic Research Program of China (973 Program) under grant No.2014CB744903, the National High-Tech Research and Development Program of China (863 Program) under grant No.2015AA015303, the Collaborative Innovation Center of Novel Software Technology and Industrialization, and the Fundamental Research Funds for the Central Universities under grant No.NS2016093. Taolue Chen is partially supported by the ARC Discovery Project (DP160101652), the Singapore Ministry of Education AcRF Tier 2 grant (MOE2015-T2-1-137), and an oversea grant from the State Key Laboratory of Novel Software Technology, Nanjing Unviersity.

References

[1] Abebe, S.L., Tonella, P.: Natural language parsing of program element names for concept extraction. In: Program Comprehension (ICPC), 2010 IEEE 18th International Conference on, pp. 156–159. IEEE (2010)

[2] Abreu, R., Zoeteweij, P., Van Gemund, A.J.: Spectrum-based multiple fault localization. In: Automated Software Engineering, 2009. ASE'09. 24th IEEE/ACM International Conference on, pp. 88–99. IEEE (2009)

[3] Bandyopadhyay, A.: Improving spectrum-based fault localization using proximity-based weighting of test cases. In: Automated Software Engineering (ASE), 2011 26th IEEE/ACM International Conference on, pp. 660–664. IEEE (2011)

[4] Brants, T.: Tnt: a statistical part-of-speech tagger. In: Proceedings of the sixth conference on Applied natural language processing, pp. 224–231. Association for Computational Linguistics (2000)

[5] Capobianco, G., Lucia, A.D., Oliveto, R., Panichella, A., Panichella, S.: Improving ir-based traceability recovery via noun-based indexing of software artifacts. Journal of Software: Evolution and Process **25**(7), 743–762 (2013)

[6] Croft, W.B., Metzler, D., Strohman, T.: Search engines: Information retrieval in practice. Addison-Wesley Reading (2010)

[7] Čubranić, D.: Automatic bug triage using text categorization. In: In SEKE 2004: Proceedings of the Sixteenth International Conference on Software Engineering & Knowledge Engineering. Citeseer (2004)

[8] Gomaa, W.H., Fahmy, A.A.: A survey of text similarity approaches. International Journal of Computer Applications **68**(13), 13–18 (2013)

[9] Gupta, S., Malik, S., Pollock, L., Vijay-Shanker, K.: Part-of-speech tagging of program identifiers for improved text-based software engineering tools. In: Program Comprehension (ICPC), 2013 IEEE 21st International Conference on, pp. 3–12. IEEE (2013)

[10] Hasan, F.M., UzZaman, N., Khan, M.: Comparison of different pos tagging techniques (n-gram, hmm and brills tagger) for bangla. In: Advances and Innovations in Systems, Computing Sciences and Software Engineering, pp. 121–126. Springer (2007)

[11] Islam, A., Inkpen, D.: Semantic text similarity using corpus-based word similarity and string similarity. ACM Transactions on Knowledge Discovery from Data (TKDD) **2**(2), 10 (2008)

[12] J.Asmussen, D.: Survey of pos taggers-approaches to making words tell who they are. DK-CLARIN WP 2.1 Technical Report (2015)

[13] Kim, S., Zhang, H., Wu, R., Gong, L.: Dealing with noise in defect prediction. In: Software Engineering (ICSE), 2011 33rd International Conference on, pp. 481–490. IEEE (2011)

[14] Ko, A.J., Myers, B.A., Chau, D.H.: A linguistic analysis of how people describe software problems. In: Visual Languages and Human-Centric Computing, 2006. VL/HCC 2006. IEEE Symposium on, pp. 127–134. IEEE (2006)

[15] Kochhar, P.S., Tian, Y., Lo, D.: Potential biases in bug localization: Do they matter? In: Proceedings of the 29th ACM/IEEE international conference on Automated software engineering, pp. 803–814. ACM (2014)

[16] LaToza, T.D., Myers, B., et al.: Visualizing call graphs. In: Visual Languages and Human-Centric Computing (VL/HCC), 2011 IEEE Symposium on, pp. 117–124. IEEE (2011)

[17] Le, T.D.B., Thung, F., Lo, D.: Predicting effectiveness of ir-based bug localization techniques. In: Software Reliability Engineering (ISSRE), 2014 IEEE 25th International Symposium on, pp. 335–345. IEEE (2014)

[18] Lukins, S.K., Kraft, N., Etzkorn, L.H., et al.: Source code retrieval for bug localization using latent dirichlet allocation. In: Reverse Engineering, 2008. WCRE'08. 15th Working Conference on, pp. 155–164. IEEE (2008)

[19] Manning, C.D., Raghavan, P., Schütze, H., et al.: Introduction to information retrie-

val, vol. 1. Cambridge university press Cambridge (2008)

[20] Murphy, G.C., Kersten, M., Findlater, L.: How are java software developers using the elipse ide? Software, IEEE **23**(4), 76–83 (2006)

[21] Ostrand, T.J., Weyuker, E.J., Bell, R.M.: Predicting the location and number of faults in large software systems. Software Engineering, IEEE Transactions on **31**(4), 340–355 (2005)

[22] Rao, S., Kak, A.: Retrieval from software libraries for bug localization: a comparative study of generic and composite text models. In: Proceedings of the 8th Working Conference on Mining Software Repositories, pp. 43–52. ACM (2011)

[23] Saha, R.K., Lease, M., Khurshid, S., Perry, D.E.: Improving bug localization using structured information retrieval. In: Automated Software Engineering (ASE), 2013 IEEE/ACM 28th International Conference on, pp. 345–355. IEEE (2013)

[24] Shokripour, R., Anvik, J., Kasirun, Z.M., Zamani, S.: Why so complicated? simple term filtering and weighting for location-based bug report assignment recommendation. In: Proceedings of the 10th Working Conference on Mining Software Repositories, pp. 2–11. IEEE Press (2013)

[25] Sisman, B., Kak, A.C.: Incorporating version histories in information retrieval based bug localization. In: Proceedings of the 9th IEEE Working Conference on Mining Software Repositories, pp. 50–59. IEEE Press (2012)

[26] Tian, Y., Lo, D.: A comparative study on the effectiveness of part-of-speech tagging techniques on bug reports. In: Software Analysis, Evolution and Reengineering (SANER), 2015 IEEE 22nd International Conference on, pp. 570–574. IEEE (2015)

[27] Tong, Y., Zhou, Y., Fang, L., Chen, T.: Towards a novel approach for defect localization based on part-of-speech and invocation. In: Internetware, 2015 the Seventh International Symposium on. ACM (2015)

[28] Wang, Q., Parnin, C., Orso, A.: Evaluating the usefulness of ir-based fault localization techniques. In: Proceedings of the 2015 International Symposium on Software Testing and Analysis, pp. 1–11. ACM (2015)

[29] 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)

[30] Wang, X., Zhang, L., Xie, T., Anvik, J., Sun, J.: An approach to detecting duplicate bug reports using natural language and execution information. In: Proceedings of the 30th international conference on Software engineering, pp. 461–470. ACM (2008)

[31] Wen, W.: Software fault localization based on program slicing spectrum. In: Proceedings of the 34th International Conference on Software Engineering, pp. 1511–1514. IEEE Press (2012)

[32] Wong, C.P., Xiong, Y., Zhang, H., Hao, D., Zhang, L., Mei, H.: Boosting bug-report-oriented fault localization with segmentation and stack-trace analysis. In: Software Maintenance and Evolution (ICSME), 2014 IEEE International Conference on, pp. 181–190. IEEE (2014)

[33] Wong, W.E., Debroy, V.: A survey of software fault localization. Department of Computer Science, University of Texas at Dallas, Tech. Rep. UTDCS-45 **9** (2009)

[34] Ye, X., Bunescu, R., Liu, C.: Learning to rank relevant files for bug reports using domain knowledge. In: Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering, pp. 689–699. ACM (2014)

[35] Zhang, H.: An investigation of the relationships between lines of code and defects. In: Software Maintenance, 2009. ICSM 2009. IEEE International Conference on, pp. 274–283. IEEE (2009)

[36] Zhang, J., Wang, X., Hao, D., Xie, B., Zhang, L., Mei, H.: A survey on bug-report

analysis. Science China Information Sciences **58**(2), 1–24 (2015)

[37] Zhou, J., Zhang, H., Lo, D.: 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, pp. 14–24. IEEE (2012)

[38] Zhou, Y., Tong, Y., Gu, R., Gall, H.: Combining text mining and data mining for bug report classification. In: Software Maintenance and Evolution (ICSME), 2014 IEEE International Conference on, pp. 311–320. IEEE (2014)