# Towards A Novel Approach for Defect Localization Based on Part-of-Speech and Invocation

Yanxiang Tong, Yu Zhou, Lisheng Fang
College of Computer Science and Technology,
Nanjing University of Aeronautics and
Aestronautics, Nanjing, China
{tongyanxiang,zhouyu,fls}@nuaa.edu.cn

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

## ABSTRACT

Given a corpus of bug reports, software developers must read various descriptive sentences in order to identify corresponding buggy source files which potentially result in the defects. This process itself represents one of the most expensive, as well as time-consuming, activities during software maintenance and evolution. To alleviate the workload of developers, many 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, leveraging both a part-of-speech based weighting strategy and the invocation relationship among source files. We also integrate an adaptive technique to strengthen the optimization of the performance. The adaptive technique consists of two modules. One is to maximize the accuracy of the first recommended file, and the other aims at improving the accuracy of the fixed defect file list. We evaluate our approach on three large-scale open source projects, i.e., ASpectJ, Eclipse, and SWT. Compared with the baseline work, our approach can improve 17.13%, 6.29% and 3.15% on *top 1*, *top 5* and *top 10* respectively for ASpectJ, 6.40%, 4.94% and 4.39% on *top 1*, *top 5* and *top 10* respectively for Eclipse, and 15.31%, 8.16% and 5.10% on *top 1*, *top 5* and *top 10* respectively for SWT.

## Categories and Subject Descriptors

D.2.7 [**Software**]: Software Engineering—*Distribution, Maintenance, and Enhancement*; H.3.3 [**Information Systems**]: Information Search and Retrieval—*Information filtering, Retrieval models and Relevance feedback*

## Keywords

Mining Software Repositories, Bug Localization, Information Retrieval, Bug Report

## 1. INTRODUCTION



*Internetware '15, November 06 2015, Wuhan, China*



DOI: http://dx.doi.org/10.1145/2875913.2875919

Bug tracking systems (BTS) are widely used for software developers to manage bug related issues 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 go 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 [35]. The developer must first carefully read the bug report, especially the descriptive parts (e.g., "Summary" and "Description") and elicit the keywords such as the 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 annoying, particularly for the one who is unfamiliar with the defective programming modules. As a result, it is highly desirable to automatically locate those potential buggy source files and recommend to the developers with the given bug reports.

In recent years, some researchers have proposed various approaches using static analysis techniques to produce a ranking list of buggy files for a processing bug report [32]. The ranking list can narrow down developer's search scales and thus help augment debugging productivity. The basic technique of these approaches is standard information retrieval (IR). It returns a ranking list of buggy files based on similarity scores between textual parts of bug reports and source code. However, the important information of bug reports does not only exists in the textual information, but also in the other parts. For example, Sisman et al. extended the IR framework by incorporating the histories of defects and modifications stored in the versioning tools [25]. The histories might fill the gaps of the vague description in the textual parts of the bug reports and improve the accuracy of rankings of the buggy files. Indeed, the source files are coded in some specific programming languages, like Java or C++, which have different grammatical/semantic features compared with natural languages. Therefore, traditional natural language processing techniques from IR field cannot be applied directly to extract discriminative features of the source code. In light of this, Saha et al. utilized code constructs and presented a structured information retrieval based technique [23]. They divided the code of each file into four parts, namely, Class, Method, Variable and Comments, and achieve the similarity score between a source file and a bug report by adding the eight similarity scores between the four parts of the source file and the summary, description of a bug report. In [35], Zhou et al. integrated the file length in-

**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

**Figure 1: Bug Report example**

formation to strengthen the traditional Vector Space Model and utilized the similar bugs to calibrate the ranking results. After that, many other researchers have explored combining other attributes of the bug reports and the source codes to further improve the accuracy of bug localization [28, 33, 31].

However, we observe that most of existing works, if not all, treat the words (apart from stop words) equally without discrimination. To be more specific, they do not consider the part-of-speech features of underlying words in bug reports. In reality, to understand the meaning of a bug report, part-of-speech of every 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 overrid 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" doesn't exist in the defect file and the adjective "default" is not a discriminative word for Java code. Thus the later words actually provide 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 fixed 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 4302 and 11459 places in the same list ranked solely by similarity [35]. 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. This 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 [14], 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 buggy files' class names or method names exist in the bug reports. In our approach, we filter the source files and only reserve the class names and method names to reduce the noisy localization for the localized bug reports. If a bug report is a localized one, this method indeed can lift up the rankings of its buggy files. However, this process also introduces potential issues, i.e., this filtering strategy will 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 doesn't contain class names and 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 defect 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, considering the demand of the developers.

The main contributions of this paper include:

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 (Mean Reciprocal Rank)*.

3. We propose an adaptive approach to maximize the accuracy of recommendations. The approach sets a selection variable $ran \in \{true, false\}$ for users. We conduct a comparative study on the same dataset in [35], which confirms the performance improvement by our approach.

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 which are related to the input queries. The matching result is not unique, but consists of several objects with different degrees of correlation, forming a ranking list [18]. 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 summary and description parts of a bug report as a query, the source files as documents and ranks the relevancy 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 vectors 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, stop word list of source code is totally different from natural language documents and is always composed of particular words relying on programming language. Third, all reserved words should be transformed into their basic form by Poster Stemming Algorithm, which can normalize the term 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. It is generally known 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, many researches have 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, etc [4, 9, 11]. State of the art taggers highlight accuracy of circ 93% compared to 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 are composed in natural languages and 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 bug reports 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 the ranking lists. It takes all the buggy files' rankings into account. There are many relevant source code files corresponding to a bug report, the *Average Precision (AP)* for a bug report $n$ can be computed as:

$$AP_n = \sum_{k=1}^{|S|}\frac{P(k)\times pos(k)}{\text{Numbers of Defective Files}} \tag{2}$$

   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 *ran*.

1. *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 to automatically adjust the weight of each term based on its tag. We note that how to filter the source code is determined by the parameter *ran*.

2. *Module 2* is based on the results of *Modules 1*. We use the invocation relationship to further boost 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.

3. *Parameter ran* is a trigger of our adaptive recommendation depending on the developers' context. If *ran* is set to be true, it means developers want a single decisive, most probable file to this bug report; otherwise *ran = false* means a list of $n$ files would be provided.

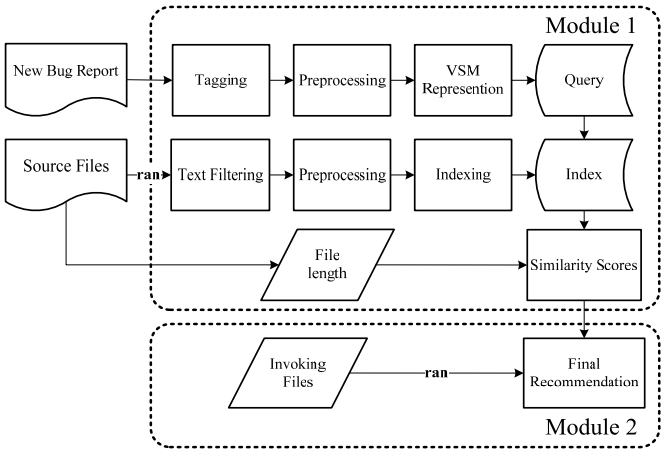


**Figure 2: The Overview of Our Approach**

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
options/NNS

**Figure 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 defined simply by class name searching. 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 will be 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-Posttagger[1] to mark all the terms of the bug reports. Figure 3 shows the tagging results for the summary of AspectJ (Bug ID: 29769). 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 mass)", "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 needn't keep the markings untill the calculation step.

The importance of summary and description of bug reports is different [13], but we treat them together as a query. In order to highlight the summary, we follow the heuristics from [29] and increase its terms' frequency twice to description. For source files, we filter the source code before preprocessing, and set a parameter *ran* to determine whether to reserve class names and method names only. Because the three experimental projects are programmed in Java, we use API of Eclipse JDT named Abstract Syntax Tree (AST) 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 parameter *ran* is set to be true, only the 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, and build a Vector Space Model to represent both texts based on index terms of bug reports. 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 [18]. In view of this point, we use *the logarithm variant* to revise the $tf_{t,d}$ [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 [20, 34]. 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 $(0.5, 1.0)$, which makes the coefficient most effective. To this aim, we first compute the average length $avg$ of all source files and then calculate the

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

standard deviation $sd$ as:

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

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

$$low = avg - 3 \times sd, high = avg + 3 \times sd \quad (9)$$

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

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

The coefficient $lens$ is computed as:

$$lens = \frac{e^{norm}}{1 + e^{norm}} \quad (13)$$

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

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

We then obtain all the similarity scores of source files and the new bug report and 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 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-ranking buggy files to increase scores of other buggy files.

The textual information of a bug report has explicitly directionality and may include one or more class-name (filename) terms. We define the source files corresponding to the file-name terms as *hitting files* and the source file of *hitting files* which ranks highest in initial ranking list produced by *Module 1* as $hf$. We believe that the $hf$ has a higher possibility to be the defective one. Figure 4 shows the detailed processing of the invoking method. First, we extract all filename 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 $hf$ to search the files it invokes as invocation files. At last, the final score ($FScore_{r,inf}$) of the invoking file $inf$ in *Module 2* is calibrated as follows:

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

where $Sim_{r,hf}$ is the similarity score between the highest scoring 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 coefficient of the formula and we set $a = 0.3$ in our approach.

## 3.3 Adaptive Strategy

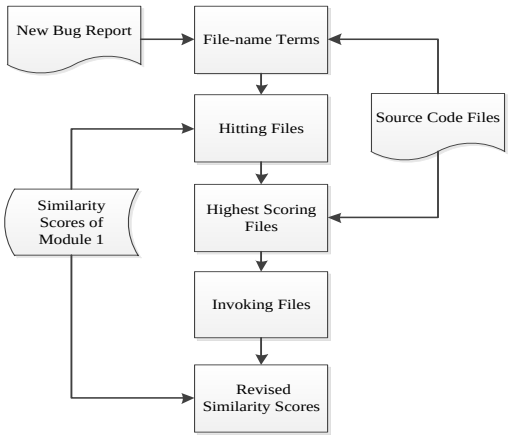


**Figure 4: The Detail of Module 2: Invoking Method**

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 want a decisive file, the accuracy of *top 1* will get preferential treatment. In this situation, we remove all the elements of the source files except the class names and method names. Otherwise, the developers need *N (for example, N = 5, 10)* candidate files, and thus the overall performance of *top N (N = 5, 10)* must be considered and we find that keeping all the 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. We set a parameter *ran* to realize this. If *ran* is set to be true which means developers want a decisive file to the bug, other elements of source files except for class names and method names must be removed before text preprocessing. The output of our recommendation is a single file. Otherwise if *ran* is set to be false which means a list of *N (N = 5, 10)* files would be provided. The output of the our recommendation is then $N$ candidate files.

# 4. EXPERIMENTS

To evaluate our approach, we conduct an empirical study and use three cases as in [35], i.e., AspectJ, Eclipse, and SWT, and the dataset are bug reports of fixed bugs. The information of the dataset is given in Table 1. We compare our approach with the rVSM model of BugLocator ($\alpha = 0$) proposed in [35].

**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 |

Table 2 depicts the results achieved by our approach for all the three projects. If the value of *ran* is set to be *true*, about 114 AspecJ bugs (39.86%), 946 Eclipse bugs (30.76%) and 46 SWT bugs (46.94%) are successfully located and their fixed files can be found at the top 1 in recommendation. If the value of *ran* is set to be *false*, our approach can locate 76 AspecJ bugs (26.57%), 912 Eclipse bugs (29.66%) and

39 SWT bugs (39.79%) whose fixed files are at the top 1, 135 AspecJ bugs (47.20%), 1571 Eclipse bugs (51.09%) and 72 SWT bugs (73.47%) whose fixed files are at the top 5 and 168 AspecJ bugs (58.74%), 1854 Eclipse bugs (60.29%) and 81 SWT bugs (82.65%) whose fixed files are at the top 10. Besides, the results of *MRR* and *MAP* with $ran = true$ are better than the ones with $ran = false$ in AspectJ and SWT. Because, the result of top 1 contributes more to the performance of *MRR* and *MAP* than the results of top 5 and top 10.

**Table 2: The Performance of Our Approach**

| Project | Method | TOP 1 | TOP 5 | TOP 10 | MRR | MAP |
|---|---|---|---|---|---|---|
| AspectJ | ran=true | 114 (39.86%) | N/A | N/A | 0.44 | 0.24 |
| | ran=false | 76 (26.57%) | 135 (47.20%) | 168 (58.74%) | 0.37 | 0.21 |
| Eclipse | ran=true | 946 (30.76%) | N/A | N/A | 0.36 | 0.23 |
| | ran=false | 912 (29.66%) | 1571 (51.09%) | 1854 (60.29%) | 0.40 | 0.30 |
| SWT | ran=true | 46 (46.94%) | N/A | N/A | 0.62 | 0.56 |
| | ran=false | 39 (39.79%) | 72 (73.47%) | 81 (82.65%) | 0.55 | 0.49 |

*Method 1* defines the process of locating the bugs in our approach when *ran* is set to be true and *Method 2* represents another process of locating the bugs when *ran* is set to be false. *Method 1* takes the advantage of the localized bug reports and filters out more noisy data, contributing more to the accuracy of *top 1*. From the results of *top 1* for the three 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 the three projects which confirms the above idea. With the increasing scale of bug reports, the localized bug reports increased and play a dominant role in bug localization leading to the best performance of *top 1*.

As aforementioned, the description part of bug reports is more noisy. It may involve many nouns, verbs and adjectives with or without discrimination. The discriminating terms lead to a good localization. Moreover, reserving the class names and method names of source files is beneficial for localizing the bug reports with specific class-name terms and method-name terms but damages the localization of the bug reports without these terms.

**Table 3: The Execution Time of BugLocator ($\alpha = 0$) and Module 1 of Our Approach**

| Approach \ Projects | AspectJ | Eclipse | SWT |
|---|---|---|---|
| BugLocator | 56s | 57min7s | 6s |
| Our Approach | 49s | 8min51s | 12s |

Because our approach has filtered the source code in the beginning, particularly when $ran = true$, the *Module 1* seems more time-saving comparing to BugLocator without similar bugs module. Table 3 displays the execution time of rSVM model and *Module 1* of our approach. The execution time of BugLocator ($\alpha = 0$) for AspectJ, Eclipse and SWT is 56 seconds, 57 minutes 7 seconds and 6 seconds. The execution time of the *Module 1* of our approach is 49 seconds, 8 minutes 51 seconds and 12 seconds. Figure 5 reveals the trend of execution time for the two approaches of comparison with the increase of data size. Because the execution

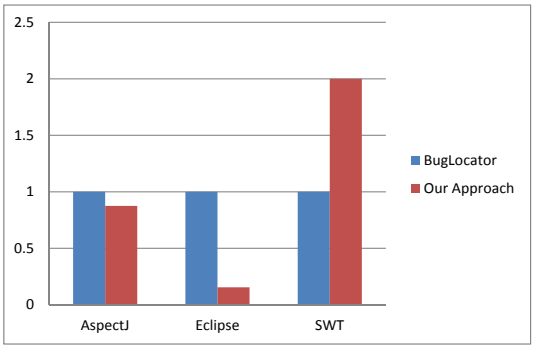


**Figure 5: The Trend of Execution time for The Two Approaches of Comparison**

time of the three projects is not at the same level, we set the execution time of all the projects using BugLocator without similar bugs as unit time 1 and represent the other as the proportion of it. We can discover that the *Module 1* relatively decreases the execution time and more efficient except for SWT because of the small dataset. Thus, 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 of course costs much time. Although our approach needn't obtain the invocation relationship of all the source files, it also needs to spend time reviewing thousands of highest scoring source files $hf$ to get the invoking files. Therefore, how to improve the efficiency of executing the *Module 2* is important for saving the time of executing our approach. We implement our approach in a Intel(R)Core(TM)i7-4790 32G machine with Win7 operating system and the version of JDK is 1.8.0.

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 as many researchers do. Table 4 compares the accuracy of our approach with BugLocator. As we can see, the performance of our approach is better than BugLocator without using similar bugs.

When *ran*'s value is set *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 the results have a considerable enhancement. For example, the accuracy of *top* of this method for AspectJ almost improves twice. The performance of *Method 1* are 39.86% for AspectJ comparing to 22.73% of rVSM, 30.76% for Eclipse comparing to 24.36% and 46.94% for SWT comparing to 31.63%. Although this method just provides one file, the statistics of *MRR* and *MAP* are based on the ranking lists *Method 1* produced 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 bugs.

When *ran*'s value is set *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)*

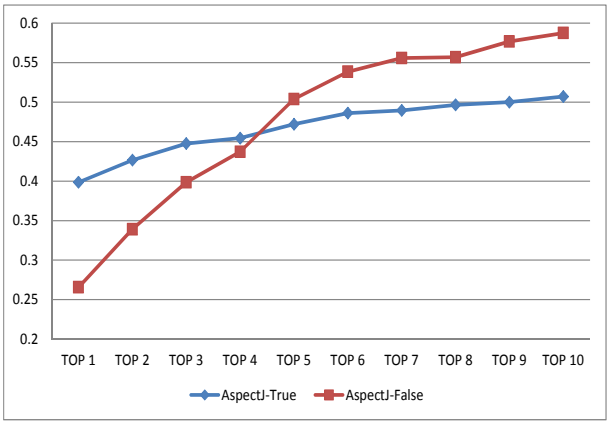


**Figure 6: The Performance for Method 1 and Method 2 in AspectJ**

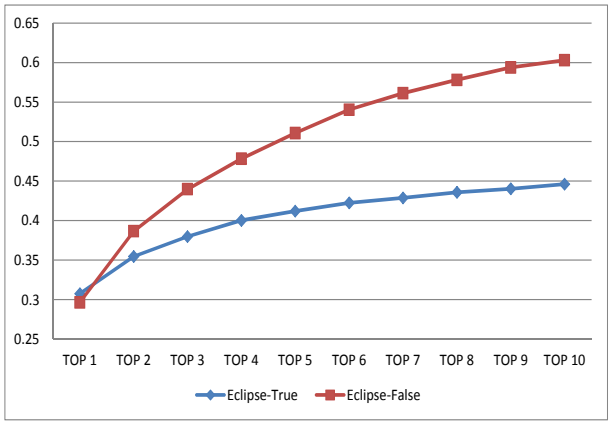


**Figure 7: The Performance for Method 1 and Method 2 in Eclipse**

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 which are 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 and about 8.16% in *top 1*, 8.16% in *top 5* and 5.10% in *top 10* for SWT. It is interesting to discovered that our approach improves most in *top 5*.

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

| Project | Method | TOP 1 | TOP 5 | TOP 10 | MRR | MAP |
|---|---|---|---|---|---|---|
| AspectJ | ran=true | 114 (39.86%) | N/A | N/A | 0.44 | 0.24 |
| | ran=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 | ran=true | 946 (30.76%) | N/A | N/A | 0.36 | 0.23 |
| | ran=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 | ran=true | 46 (46.94%) | N/A | N/A | 0.62 | 0.56 |
| | ran=false | 39 (39.79%) | 72 (73.47%) | 81 (82.65%) | 0.55 | 0.49 |
| | BugLocator | 31 (31.63%) | 64 (63.31%) | 76 (77.55%) | 0.47 | 0.40 |

To further explain the performance of the two selective methods in our approach, we calculate the *top N (N=1,2 ...9,10)* of AspectJ and Eclipse. Figure 6 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 the *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*.

But for the Eclipse with 3075 bug reports, only the *top 1* of *Method 1* is better than the *top 1* of *Method 2*. The results of *Method 1* from *top 2* to *top 10* are all worse than the *Method 2*. The discovery above is shown in Figure 7. 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. Thus, if the developers want a recommended file, our approach makes use of *Method 1* and if the developers want *N (N=5,10)* recommended files, our approach makes use of Method 2.

## 5. THREATS TO VALIDITY

In this section, we discuss the possible threats to 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 [35] to assure the fair comparison. Thus, whether or not this heuristic fits all the Java projects is still left to be further studied in future.

2. *Invocation Validity.* The call graph of the projects we used is of large scale, especially for Eclipse, and the granularity of call graph produced by other tools are at method level which isn't suitable for our approach. The *Module 2* of our approach just needs file invocation relationship. Thus, we have used the simple string-based searching technique to find the invocation files of a certain source file. Despite the fact that the simple searching method is easy to implement, it may decrease the performance of *Module 2* to some degree because of the introduced noisy files. But, we will try to use the JDT's plug-in called *Call Hierarchy* to extract the invocation relationship between source files [19, 15].

## 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 [30].

Therefore, automatic defect localization techniques that can guide programmers are much-needed. Dynamic 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 dynamical 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 [17, 21]. 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 [10, 22, 27]. But these works do not consider the POS features of the underlying reports.

Apart from the efforts in defect localization, there is another thread of relevant work on the bug report classification. 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 [14]. A lot of research has been conducted for reducing the noise in bug reports [12]. 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 [36].

In resent years, Zhou et al. have used the Vector Space Model to represent the texts and take source files' length 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. have combined the above three discoveries to increase the results [28]. Moreover, Ye et al. have used the domain knowledge to cover all accessible features to enhance the IR-based bug location technique [33]. 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 [16].

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 the *Method 1* of our adaptive approach [14].

# 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 which provides the potential defective source files according to the developers' demand, based on the value of parameter *ran*. We take the advantage 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 defect localization. The evaluation results on three open-source projects demonstrate the feasibility of our adaptive approach and it indicates a better performance compared with the baseline work, i.e., BugLocator.

In the future, we plan to integrate more features to our approach, such as similar bugs and version history. We want to propose a more adaptive approach for more complex user demands and the *Module 2* will be refined to decrease the number of noisy files which may result in a more exciting improvement. Besides, our approach will be expanded to utilize other kind of dataset, such as bug reports of unresolved bugs.

## Acknowledgments

The work was partially funded by the National Natural Science Foundation of China under grant No. 61202002, the Natural Science Foundation of Jiangsu Province under grant No. BK20151476, the National High-tech Research and Development Program of China (863 Program) under grant No. 2015AA015303 and the Collaborative Innovation Center of Novel Software Technology and Industrialization.

# 8. REFERENCES

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

[18] C. D. Manning, P. Raghavan, H. Schütze, et al. *Introduction to information retrieval*, volume 1. Cambridge university press Cambridge, 2008.

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

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

[21] D. P. Pathak and S. Dharavath. A survey paper for bug localization. 2012.

[22] S. Rao and A. Kak. 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*, pages 43–52. ACM, 2011.

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

[24] R. Shokripour, J. Anvik, Z. M. Kasirun, and S. Zamani. 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*, pages 2–11. IEEE Press, 2013.

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

[26] Y. Tian and D. Lo. 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*, pages 570–574. IEEE, 2015.

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

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

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

[30] W. Wen. Software fault localization based on program slicing spectrum. In *Proceedings of the 34th International Conference on Software Engineering*, pages 1511–1514. IEEE Press, 2012.

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

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

[33] X. Ye, R. Bunescu, and C. Liu. 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*, pages 689–699. ACM, 2014.

[34] H. Zhang. An investigation of the relationships between lines of code and defects. In *Software Maintenance, 2009. ICSM 2009. IEEE International Conference on*, pages 274–283. IEEE, 2009.

[35] J. Zhou, H. Zhang, and D. Lo. Where should the bugs be fixed? more accurate information retrieval-based bug localization based on bug reports. In *Software*

*Engineering (ICSE), 2012 34th International Conference on*, pages 14–24. IEEE, 2012.

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