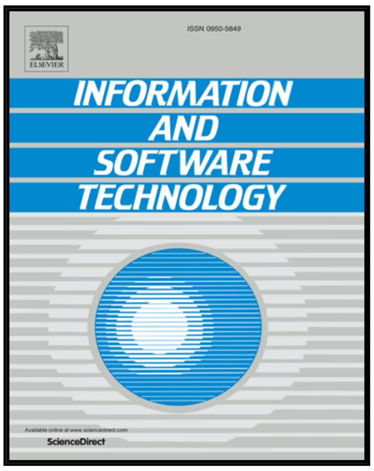


Improved bug localization based on code change histories and bug reports

Klaus Changsun Youm, June Ahn, Eunseok Lee

PII: S0950-5849(16)30305-6
DOI: 10.1016/j.infsof.2016.11.002
Reference: INFSOF 5776

To appear in: *Information and Software Technology*

Received date: 13 March 2016
Revised date: 1 November 2016
Accepted date: 1 November 2016

Please cite this article as: Klaus Changsun Youm, June Ahn, Eunseok Lee, Improved bug localization based on code change histories and bug reports, *Information and Software Technology* (2016), doi: 10.1016/j.infsof.2016.11.002

Improved bug localization based on code change histories and bug reports[☆]

Klaus Changsun Youm[a,b,*], June Ahn[a], Eunseok Lee[a,**]

[a]*Department of Information and Communication Engineering, Sungkyunkwan University, Republic of Korea*
[b]*Mobile Communication and Business, Samsung Electronics, Republic of Korea*

Abstract

Context: Several issues or defects in released software during the maintenance phase are reported to the development team. It is costly and time-consuming for developers to precisely localize bugs. Bug reports and the code change history are frequently used and provide information for identifying fault locations during the software maintenance phase.
Objective: It is difficult to standardize the style of bug reports written in natural languages to improve the accuracy of bug localization. The objective of this paper is to propose an effective information retrieval-based bug localization method to find suspicious files and methods for resolving bugs.
Method: In this paper, we propose a novel information retrieval-based bug localization approach, termed Bug Localization using Integrated Analysis (BLIA). Our proposed BLIA integrates analyzed data by utilizing texts, stack traces and comments in bug reports, structured information of source files, and the source code change history. We improved the granularity of bug localization from the file level to the method level by extending previous bug repository data.
Results: We evaluated the effectiveness of our approach based on experiments using three open-source projects, namely AspectJ, SWT, and ZXing. In terms of the mean average precision, on average our approach improves the metric of BugLocator, BLUiR, BRTracer, AmaLgam and the preliminary version of BLIA by 54%, 42%, 30%, 25% and 15%, respectively, at the file level of bug localization.
Conclusion: Compared with prior tools, the results showed that BLIA outperforms these other methods. We analyzed the influence of each score of BLIA from various combinations based on the analyzed information. Our proposed enhancement significantly improved the accuracy. To improve the granularity level of bug localization, a new approach at the method level is proposed and its potential is evaluated.

Keywords: `bug localization, information retrieval, bug reports, stack traces, code change history, method analysis`

1. Introduction

Software maintenance costs after the release of a product are greater than the cost of the design and implementation phases [2–4]. Issues or defects in the released software are updated and managed during the development phase using the bug or issue management system. Developers who are assigned to resolve the report initiate activities to fix the problem. They attempt to reproduce the same result in the bug report and then find the locations of the defects. However, it is costly and time-consuming to precisely localize bugs, and bug localization is a tedious process for developers. In cases of large software products, it is difficult for hundreds of developers to resolve the large number of bug reports. Therefore, effective methods for locating bugs automatically from bug reports are desirable in order to reduce the resolution time and software maintenance costs.

To this end, various studies regarding Change Impact Analysis (CIA), based on differences in the analysis of source file versions, were proposed during the 2000s [5–12]. In the 2010s, numerous Spectrum-Based Fault Localization (SBFL) techniques have been suggested and evaluated [12–16]. Recently, researchers have applied various Information Retrieval (IR) techniques [17], which are mainly used to search the text domain for software maintenance for the purposes of feature location, developer identification and impact analysis [18]. IR-based bug localization techniques have attracted significant attention due to their relatively low computational cost and improved accuracy compared to change impact analysis or spectrum-based fault localization. In these IR approaches, the main idea is that a bug report is treated as a query and that the source files in the software product to be searched comprise the document collection. To improve the accuracy of locating bugs, the following are all used in various combi-

[☆]A preliminary version of this paper was presented and best paper awarded at the 22th Asia-Pacific Software Engineering Conference (APSEC 2015) [1].
*Corresponding author. Tel.: +82 31 279 8511
**Principal corresponding author. Tel.: +82 31 290 7135
Email addresses: klausyoum@skku.edu, klaus.youm@samsung.com (Klaus Changsun Youm), ahnjune@skku.edu (June Ahn), leees@skku.edu (Eunseok Lee)

nations: similarity analysis of previously fixed bugs [19], the use of structured information of the source files instead of treating them as simple documents [20], version change history analysis [21] and stack trace analysis [22–24].

We analyzed the bug/issue management process to identify the pieces of information useful to developers for localizing bugs. With these considerations in mind, we propose a static and integrated analysis approach for bug localization by utilizing texts, stack traces and comments in bug reports, structured information of the source files, and the source code change history. First, we analyze the similarity between texts in a bug report and the source files using an IR method. We adopt the revised Vector Space Model (rVSM) of Zhou et al. [19], then integrate the structured information analysis of source files, the effectiveness of which was demonstrated by Saha et al. [20]. Second, the similarities of previously fixed bug reports are analyzed using a basic IR method [19]. Third, if a bug report includes stack traces, stack traces are also analyzed to identify suspicious files therein [23]. Fourth, an analysis of the historical information from the source code changes is performed to identify suspicious files and methods for predicting bugs [21]. Fifth, we integrate the above four types of information to localize suspicious files in each bug report; this represents an output of bug localization at the file level. Sixth, from the ranked suspicious files, we analyze the similarity between methods in the files and bug reports. Finally, suspicious methods are ranked and listed based on a combination of the scores of the fourth and sixth steps. This also represents an output of bug localization at the method level.

The contributions of our research are as follows:

1. We propose a novel IR-based bug localization approach termed Bug Localization using Integrated Analysis (BLIA). We utilize the content, stack traces and comments in bug reports, structured information of the source files, and the source code change history. We design a combined method to integrate all analyzed data in order to localize bugs at not only the file level, but also the method level. BLIA v1.0, which is a preliminary version, is focused on bug localization at the file level. The current version, BLIA v1.5, extends its implementation to both the method level and the analysis of the comments in bug reports.
2. We find the optimized range of parameters that control the influence rate of each piece of information analyzed.
3. Bug repositories for our experiments were extended. The comments, fixed methods and fixed commits for each bug report were inserted. The data are available to improve the accuracy of bug localization research.

The remainder of this paper is organized as follows. Section 2 describes the background for this work. Section 3 describes our proposed BLIA approach. In Section 4, we present our experimental design and evaluation metrics. We discuss the experimental results and threats to validity in Section 5 and Section 6, respectively. Section 7 surveys related work. Finally, our conclusion and directions for future work are presented in Section 8.

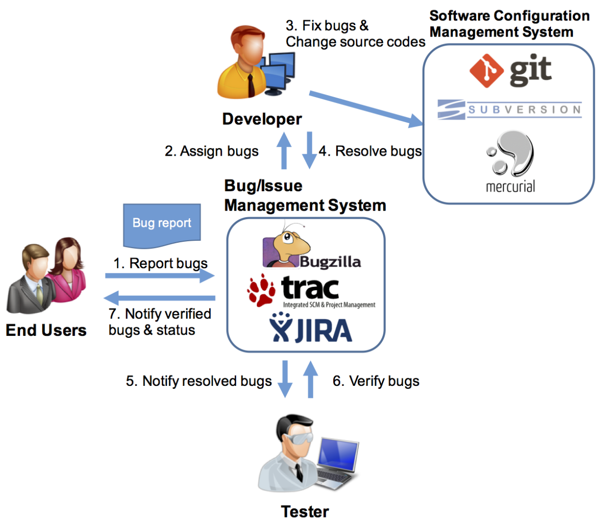


Fig. 1. Bug/Issue management process.

2. Background

In this section, we first present a bug/issue management process to understand how bug reports are resolved. With an understanding of this process, we can identify factors that are helpful in locating bugs. We describe an example of an actual bug report from an open-source project. We demonstrate the general process and techniques of IR-based bug localization. In the last subsection, we provide historical information from the software configuration repository.

2.1. Bug/Issue management process

Fig. 1 illustrates a bug/issue management process used by developers to resolve a bug report after its assignment. Users submit a report to a bug/issue management system when they use a released software product and unexpected errors occur. Not only do industrial projects apply bug/issue management systems such as Bugzilla, Trac and JIRA, but open-source projects do as well. A bug report is assigned to developers who attempt to find the location of the bug. They usually try to reproduce the bug scenario in the bug report. The developers who find the location and root cause of a defect first fix the source files and then test the modified software again. If the error scenario in the bug report does not recur, the status of the bug report is changed to “Resolved” by the assigned developers. After resolution of the bug report, if the development team has a separate testing team, a tester is notified of the updated status. The tester checks the scenario of the reported bug, and then verifies that the bug has been resolved. The status is changed to “Verified” if the testing is completed

Table 1
A bug report of SWT and the fixed information corresponding to it.

Bug ID	113971
Summary	Test failures in **Tree** on N20051027-0010
Status	RESOLVED FIXED
Reported	2005-10-27 10:26:49 EDT by Douglas Pollock
Product	Platform
Component	SWT
Version	3.1
Description	The "org.eclipse.ui.tests" had test failures on MacOS X last night. The failures were in **Tree** code. One was a NPE, which I believe has already been fixed. The other is as follows: java.lang.ArrayIndexOutOfBoundsException at java.lang.System.arraycopy(Native Method) at org.eclipse.swt.widgets.Tree.createItem (**Tree**.java:714) ...
Comment	The JDT/UI test cases fail due to a different exception in the tree. The exception is: Caused by: java.lang.NullPointerException at org.eclipse.swt.widgets.Tree.itemNotificationProc (**Tree**.java:2049) at org.eclipse.swt.widgets.Display.itemNotificationProc (Display.java:2220) ...
Fixed	2005-10-28 14:16:12 EDT
Fixed commit	2cd9c142b2dc2cb5847c910c44c758dfcfaf6ef3
Fixed file	org.eclipse.swt.widgets.**Tree**.java
Fixed method	createItem

without errors. A verification notification of the bug report is then sent to the reporter.

2.2. Bug reports

Bug reports are vital clues for developers for finding the location of defects. Developers guess which source files and lines are buggy after understanding the assigned bug report. Usually, they attempt to reproduce the scenario in the bug report, and then attempt to trace the root cause of the defects and find the location of the source files.

Table 1 presents an existing bug report[1] from Eclipse Bugzilla[2]. A bug report comprises the bug ID, summary, description, reported date, fixed date, status and other related fields. Some bug reports include stack traces of when the exception occurred. This stack trace information is critical for localizing a bug [25]. For example, developers attempt to reproduce an issue scenario and assume that the location of a bug is based on class names or method names in the bug report. Comments included in the bug report provide further information on the error situation.

Furthermore, developers search for similar bug reports in the bug repository. If similar bug reports are found, the fixed files used to resolve the bugs are considered as candidates for patching up the software product [26].

2.3. Information retrieval (IR)-based bug localization

In IR-based bug localization, the source files for a software product comprise the collection of documents to be searched, with each bug report being a search query. Suspicious source files based on the estimated relevance to each bug report are ranked using IR techniques. The better an IR approach is at interpreting the bug report and the source files, the more likely it is to highly rank the source files that need to be fixed.

An IR system typically consists of a three-step preprocess: text normalization, stopword removal and stemming. With preprocessing of the source files and bug report, analyzable terms are created and their similarity is calculated. We describe each preprocessing step briefly.

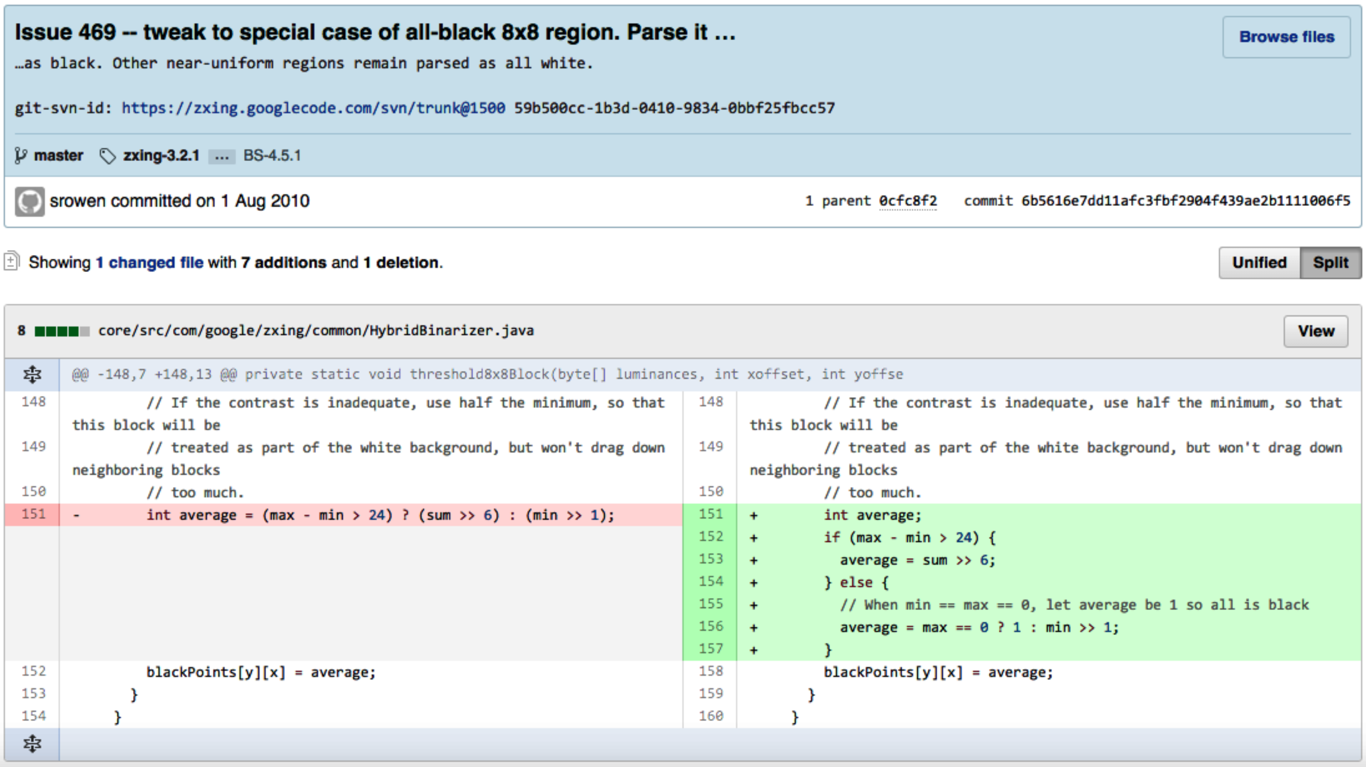
Firstly, text normalization involves the removal of punctuation marks, tokenization of terms and splitting identifiers. The Abstract Syntax Tree (AST) in the source files is used to parse identifiers. These identifiers are split into their constituent words by following Camel Case splitting [27]. For example, a method named "combineAnalyzedScore" is split into "combine", "Analyzed" and "Score". The next step is filtering out of a set of extraneous terms in a stopword list to improve the accuracy and reduce spurious matches. For source files, programming language keywords are also defined as stopwords. The final step is stemming, which reduces inflected or derived words into a common root form. For example, the words "going" and "goes" are reduced to the root form "go". By stemming terms, similar words are represented using the same term. We use the Porter stemming algorithm[3] for this step.

Once the bug report and source files have been preprocessed, the source files are indexed by collecting and storing various statistics, such as the term frequency (TF, the number of times a term occurs in a given document) and document frequency (DF, the number of documents in which the term appears). IDF refers to inverse (dampened) DF and is used to calculate the relevance or similarity between the bug report and the contents of the source files. The bug report and source files are represented as a vector using TF and IDF. In representative IR models, there are SUM [28], LDA [29], LSI [30] and VSM [17]. Zhou et al. [19] proposed an rVSM to improve the bug localization performance compared to other IR models.

2.4. Code change history

Software configuration management (SCM) systems are at the center of software development. Generally, development teams manage software configuration and version

[1]https://goo.gl/POC65q
[2]https://bugs.eclipse.org/bugs/
[3]http://tartarus.org/martin/PorterStemmer/


Issue 469 -- tweak to special case of all-black 8x8 region. Parse it …

…as black. Other near-uniform regions remain parsed as all white.

git-svn-id: https://zxing.googlecode.com/svn/trunk@1500 59b500cc-1b3d-0410-9834-0bbf25fbcc57

srowen committed on 1 Aug 2010 — 1 parent 0cfc8f2 — commit 6b5616e7dd11afc3fbf2904f439ae2b1111006f5

Showing 1 changed file with 7 additions and 1 deletion.

8 core/src/com/google/zxing/common/HybridBinarizer.java

```
@@ -148,7 +148,13 @@ private static void threshold8x8Block(byte[] luminances, int xoffset, int yoffse
           // If the contrast is inadequate, use half the minimum, so that this block will be
           // treated as part of the white background, but won't drag down neighboring blocks
           // too much.
-          int average = (max - min > 24) ? (sum >> 6) : (min >> 1);
+          int average;
+          if (max - min > 24) {
+            average = sum >> 6;
+          } else {
+            // When min == max == 0, let average be 1 so all is black
+            average = max == 0 ? 1 : min >> 1;
+          }
           blackPoints[y][x] = average;
         }
       }
```

Fig. 2. Difference in a commit of ZXing.

history, utilizing Git, SVN or Mercurial, which are open-source projects and SCM systems. This historical information is necessary to help developers trace the defect location in source files [31, 32]. Developers find suspicious buggy files and changes based on the contents of a bug report.

Fig. 2 from GitHub shows the difference[4] between two versions of the ZXing project; this difference provides analyzable data used to trace the suspicious files and methods. If candidates for bug localization are traced, developers then perform tests to focus on the exceptional result from a bug report.

3. Approach

In the previous section, we discussed the process of bug or issue management, the information from a bug report, IR-based bug localization and code change history information from the SCM system. Table 2 shows a comparison of BLUiR [20], AmaLgam [21], BRTracer [23], Lobster [24] and BLIA, which have all been proposed to improve the performance of IR-based bug localization after the original proposal of BugLocator [19]. We build our technique based on analyzable factors from the process of handling bug reports as well as previous approaches. We first describe the analyzable inputs, analysis flow and architecture of the BLIA approach. We then present each of the six main components of BLIA.

3.1. Analyzable inputs

As was mentioned regarding the bug resolution process in Section 2, developers generally trace the location of buggy source files based on the contents of bug reports, which include the report date, scenario in which the error occurs, product version, reporter, status and other fields. When an exception occurs, the user submits a bug report with stack traces. Developers first attempt to understand the bug scenario and then find the suspicious source files, which include the main words in the bug report. Stack traces are important for localizing buggy source files and lines. The code line can be traced by finding the method names or class names in the bug report. Appended comments in bug reports include additional information that was not included when submitting bug reports. They also include the stack traces or crash logs of the software program.

In addition, similar bug reports that have already been fixed also assist in defect localization. If developers have found similar bug reports in a bug/issue management system, the fixed files of similar bug reports are candidates for the new bug report. Developers search commit logs as the source code change history in an SCM system to find related recent changes that may have produced new bugs. They analyze the file, method or line differences between two versions to accurately determine the bug location.

[4]https://goo.gl/yfuY0U

Table 2
Comparison of IR-based bug localization techniques.

Approach	BugLocator [19]	BLUiR [20]	AmaLgam [21]	BRTracer [23]	Lobster [24]	BLIA [1]
Published	ICSE 2012	ASE 2013	ICPC 2014	ICSME 2014	ICSME 2014	APSEC 2015
Full name	BugLocator	Bug Localization Using information Retrieval	Automated Localization of Bug using Various Information	BRTracer	LOcating Bugs using Stack Traces and tExt Retrieval	Bug Localization using Integration Analysis
IR methods	rVSM (revised VSM)	Indris built-in TF.IDF formulation	VSM	rVSM	Lucene	rVSM
Similarity between bug report and source files	O	O	O	O	O	O
Structured information of source file	X	O	O	X	X	O
Analyzing past similar bugs	O	X	O	O	X	O
Analyzing stack trace	X	X	X	O	O (Dependency graph analysis)	O
Version history	X	X	O	X	X	O
Evaluation metrics	Top N Rank MAP(Mean Average Precision) MRR(Mean Reciprocal Rank					

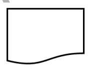
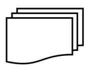
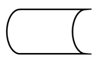
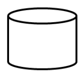
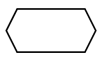
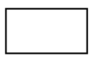
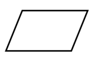
To summarize, analyzable inputs used to improve the accuracy of IR-based bug localization are as follows:

- Bug report (report date, scenario, stack traces, comments, etc.)
- Source files
- Similar fixed bug reports
- Source code change history (commit messages and differences among changes)

3.2. Analysis flow of BLIA

Fig. 3 and Fig. 4 present the overall analysis flow of BLIA for bug localization. The flowchart symbols used in the figures are described in Table 3. BLIA v1.0, which is a preliminary version, focused on the file level of bug localization (Fig. 3). We build our techniques based on rVSM as an IR model and the bug report analysis of BugLocator. The current version, BLIA v1.5, improves two aspects of BLIA v1.0. One is implementation at the method level of bug localization. For the other, we extend the analysis to include comments in the bug reports. The bug repository that was used in BLIA v1.0 does not include user comments and fixed method information. Comments are usually appended or updated to provide more detailed descriptions of the bug report. Fixed methods for each bug report are extracted from fixed files and commit logs. They are used to verify the results at the method level of BLIA v1.5.

Table 3
Flowchart symbols defined.

Symbol	**Name**	**Description**
	Document	Indicates a single document. It is for a process step that handle a document.
	Multi-documents	Indicates multiple documents. It is for a process step that handle one or more documents.
	Stored data	A general stored data for any process steps.
	Database	Database storage as data repository.
	Preparation	Indicates a preparation process flow step, such as a data preprocessing operation.
	Process	Shows an action step or process.
	Data(I/O)	Indicates inputs to and outputs from a process.
	Flow line	Shows the direction that the process flows.

3.2.1. Analysis flow of BLIA at the file level

The first phase of BLIA v1.5 (Fig. 3) is similar to that of BLIA v1.0. BLIA v1.0 performs integrated analysis of the structured information in the source files, analysis of stack traces in the bug reports and analysis of commit log messages at the file level. The difference between BLIA v1.5 and BLIA v1.0 during this phase is that comments in the bug reports are analyzed in the more recent version to

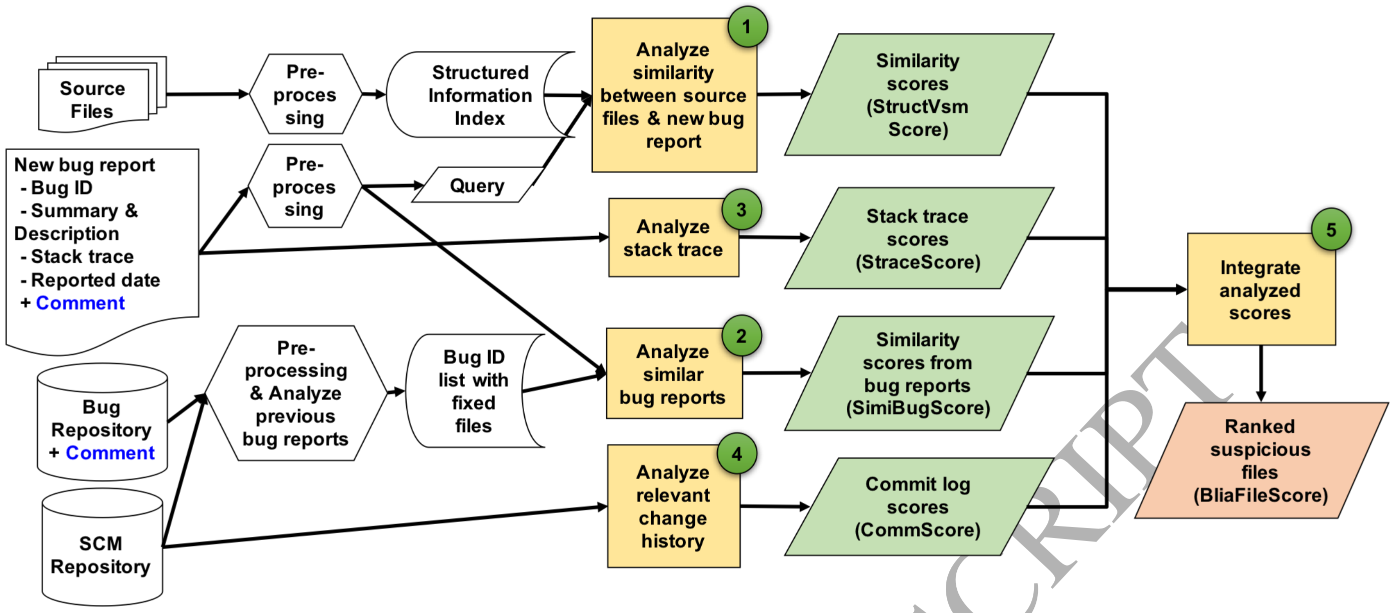

Source Files
Pre-processing
Structured Information Index
Analyze similarity between source files & new bug report
1
Similarity scores (StructVsm Score)
New bug report
- Bug ID
- Summary & Description
- Stack trace
- Reported date
+ Comment
Pre-processing
Query
Analyze stack trace
3
Stack trace scores (StraceScore)
Integrate analyzed scores
5
Bug Repository
+ Comment
Pre-processing & Analyze previous bug reports
Bug ID list with fixed files
Analyze similar bug reports
2
Similarity scores from bug reports (SimiBugScore)
Ranked suspicious files (BliaFileScore)
SCM Repository
Analyze relevant change history
4
Commit log scores (CommScore)

Fig. 3. File level analysis flow of BLIA.

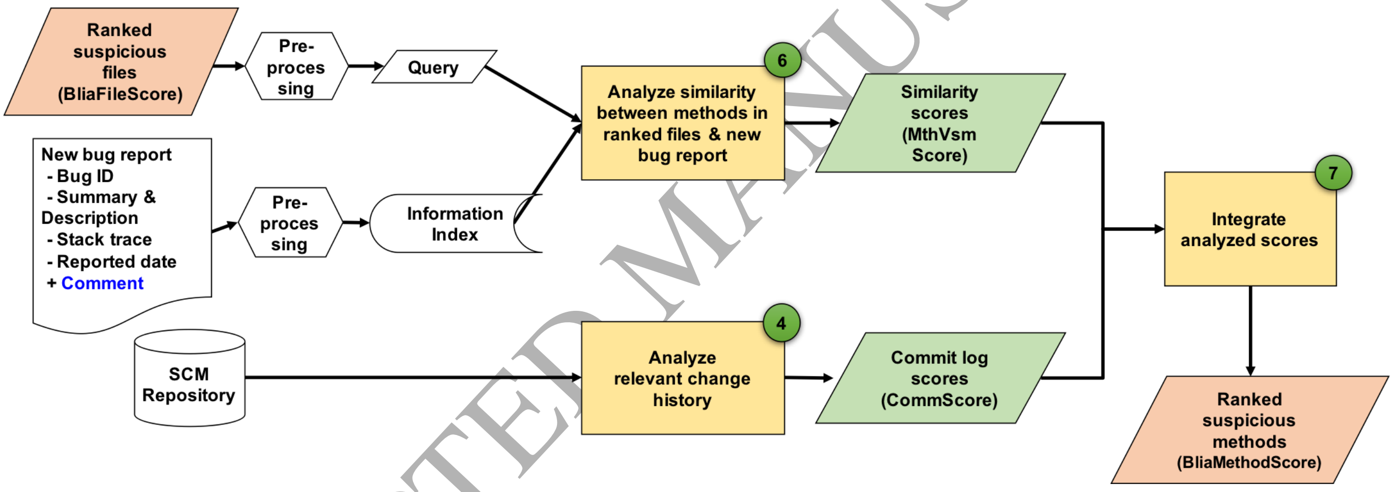

Ranked suspicious files (BliaFileScore)
Pre-processing
Query
Analyze similarity between methods in ranked files & new bug report
6
Similarity scores (MthVsm Score)
New bug report
- Bug ID
- Summary & Description
- Stack trace
- Reported date
+ Comment
Pre-processing
Information Index
Integrate analyzed scores
7
SCM Repository
Analyze relevant change history
4
Commit log scores (CommScore)
Ranked suspicious methods (BliaMethodScore)

Fig. 4. Method level analysis flow of BLIA.

improve the bug localization accuracy. The green circles in Fig. 3, Fig. 4 and Fig. 5 indicate the main processes or steps of BLIA. To explain in order, step 1 analyzes the similarity between source files and the new bug report. Structured information for the source files is generated through preprocessing. The new bug report is also preprocessed, and then a query is generated from it. We send this query to the structured information of the source files to analyze the relevance score (StructVsmScore) between the bug report and each source file. During step 2, we analyze similar fixed bug reports for the bug report. All bug reports with comments in the bug repository are preprocessed. The fixed files for each bug report are first stored. If similar bug reports are found, the similarity score (SimiBugScore) of each fixed file is calculated. Next, if stack traces are in the bug report, we extract the source file information from stack traces in the bug report. We then analyze the suspicious score (StraceScore) of the extracted source files, which comprises step 3. Additionally, we search related commit log messages and committed files that have been recently submitted before reporting the new bug report in an SCM. In step 4, we produce a calculated score (CommScore) for the committed files. BLIA v1.0 only calculates the commit log score for relevant source files. BLIA v1.5 extends the calculation of the commit log score for relevant methods. The commit log scores of the relevant methods are used during method level analysis. In step 5, the four above analyzed scores are integrated with three control parameters to calculate the final ranked suspicious score (BliaFileScore) of each source file in order to localize defects of the bug report at the file level.

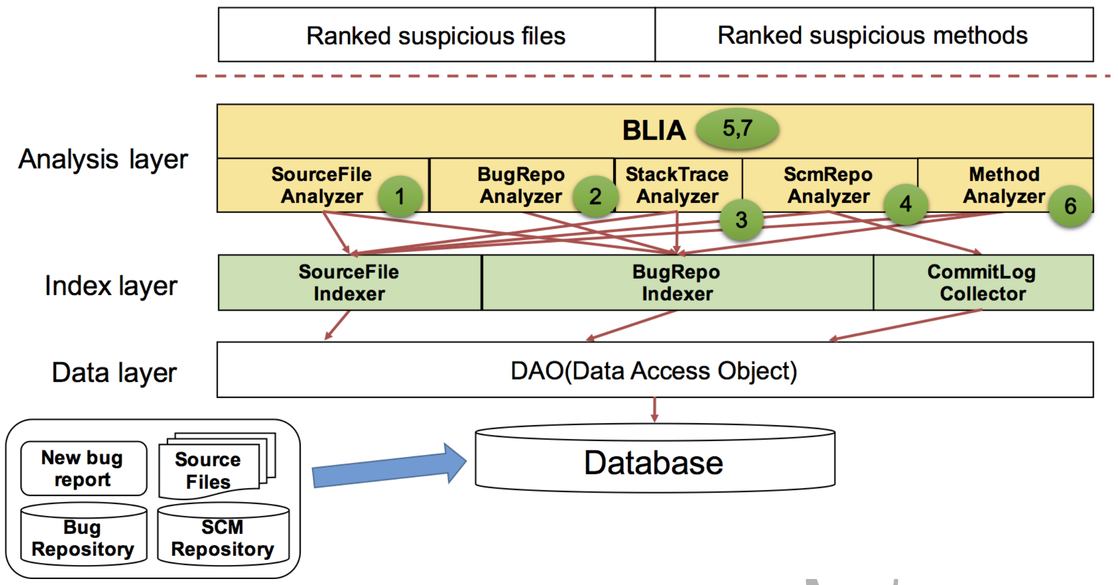


Fig. 5. BLIA architectural design.

3.2.2. Analysis flow of BLIA at the method level

For the method level of bug localization, steps 6 and 7 in Fig. 4 are extended as follows. During step 6, the similarities between the methods and the new bug report are analyzed using the ranked suspicious source files from step 5 as the input. The ranked suspicious files are preprocessed, and then the method information is used for queries. The bug report is also preprocessed. The information index for these objects is generated instead of the query from step 1. This query is sent to the information index to analyze the relevance score (MthVsmScore) between each method and the bug report. Finally, in step 7, the commit log scores for each method from step 4 and the similarity score between the methods and the bug report are integrated (with a control parameter) to calculate the final ranked suspicious score (BliaMethodScore) for each method in order to localize defects of the bug report at the method level.

3.3. Architecture of BLIA

Fig. 5 shows the layers of architectural design that portray the relationships among the analysis modules of BLIA. Using SourceFileIndex, BugRepoIndexer and CommitLogCollector in the index layer, we parse the terms of the source files, bug reports and commit log messages. We create vectors for each source file and bug report including comments. We then save the data to the internal database through the data layer. The H2 database[5], which is embedded, lightweight and does not require installation, is used by BLIA to manage the indexed data as well as to analyze the data. Five modules (SourceFileAnalyzer, BugRepoAnalyzer, StackTraceAnalyzer, ScmRepoAnalyzer, and MethodAnalyzer) in the analysis layer are used to calculate each suspicious score during each step (indicated by the green circles) by calling the indexer modules. In other words, each step mentioned in the previous section uses these analyzer modules to calculate each score. The top module, termed BLIA, integrates all of the analyzed scores and calculates the final scores for suspicious files and suspicious methods. We present in consecutive order a detailed algorithm for the BLIA analysis module from Section 3.5.

3.4. Extension of bug repositories

As described in Section 1, bug repositories for our experiments were extended.[6] Three types of information, i.e., the comments appended to a bug report, fixed methods and fixed commits, were added to the information for each bug report. For BLIA v1.0, these types of information are not included, but BLIA v1.5 analyzes this information for bug localization.

First, the comments in bug reports are used to calculate the similarity between the new bug report and previously fixed bug reports. The comments in a bug report provide more detailed descriptions that were not mentioned when the bug report was first created. Additionally, stack traces are sometimes included in comments if they are obtained after submitting the bug report. Thus, comments can also be considered as analyzable input for descriptions in a bug report.

Second, fixed methods for each bug report are included for bug localization at the method level. The bug repositories of BLIA v1.0 include only fixed files for each bug report. This is a limitation for evaluation of bug localization results at the method level. Thus, we used the JGit

[5]http://www.h2database.com/html/main.html

[6]https://goo.gl/nIdu3t

library to extract the fixed methods of each bug report by using the source code change history in SCM. JGit[7] is an open-source project and a pure Java library that implements the Git version control system. From the change history of the source files in SCM and the status history of the bug reports, we found fixed commits from a Git repository. Then, we extracted the added, deleted or modified lines from the fixed commits by using the 'git diff' function of JGit. Based on the changed lines, we calculated the fixed methods, including the lines using an AST. The method name, return type and parameters of the fixed methods are extracted and then included in the bug repositories.

Third, fixed commits for each bug report provide additional information, as already mentioned above. This information is not used directly for BLIA now. However, the fixed commits are available and can be used to improve the accuracy of bug localization in other research. For example, they can be used to localize suspicious changes or commits.

3.5. Analyzing structured source file information

As mentioned previously, source files comprise the document collection to be searched, with each bug report being a search query in IR-based bug localization. By treating each source file as one document, terms in files are extracted and a vector of terms is then created. Instead of treating source files as simple text documents, SourceFileIndexer parses source files to split them into class names, method names, variable names and comments by utilizing an AST for each source file. Compared with BLIA v1.0, SourceFileIndexer is extended to analyze the method information to calculate the similarity between methods in ranked suspicious files and bug reports. This structured information of the source files improves the accuracy of ranking for suspicious files. In actual bug reports, this is reasonable because class names or method names are usually included in either messages or the log of error cases. All identifiers from the AST are split into their constituent words using Camel Case splitting. We index full identifiers as well as split terms. Full identifiers are the original identifiers before they are split. Although it is a simple extension, Saha et al. have shown it to be effective [20]. During the removal of stopwords in preprocessing, we also remove project keywords, which are frequently used but are meaningless in the source files (e.g., "args", "param", "string" and "java"). Package names can be included in project keywords. For example, "aspectj" is frequently used in the source files of the AspectJ project. However, this word is not meaningful in the context of bug localization. Thus, it can be added to project keywords. The comments differ from other structured information because they are written in a natural language and contain javadoc tags or html tags to describe the functionality of classes or methods. We remove the tag information in comments to improve the accuracy. After preprocessing the source files, the length score of each source file is calculated by the split terms in the indexed data [19]. Similarly, we perform preprocessing of the bug report and then make it a query using BugRepoIndexer.

[7]https://eclipse.org/jgit/

SourceFileAnalyzer analyzes the TF and IDF of each term, and then calculates the vector values of each source file and bug report. Assume that a source file (f) and a bug report (b) are represented by the weighted term frequency vectors $\vec{f}$ and $\vec{b}$, respectively. Then $\{x_1,\cdots,x_m\}$ and $\{y_1,\cdots,y_n\}$ are all terms extracted from a source file and a bug report, respectively, where m and n are the total number of terms for each.

$$\vec{f} = (tf_f(x_1)idf(x_1), tf_f(x_2)idf(x_2), \cdots, tf_f(x_m)idf(x_m)) \tag{1}$$

$$\vec{b} = (tf_b(y_1)idf(y_1), tf_b(y_2)idf(y_2), \cdots, tf_b(y_n)idf(y_n)) \tag{2}$$

TF and IDF in Eq. (3) are the same as in the equation for BugLocator [19]. Here, f_{td} refers to the frequency of a term t in document d, and d_t refers to the number of documents that contain the term t.

$$tf_d(t) = log(f_{td}) + 1, \quad idf(t) = log(\frac{d}{d_t}) \tag{3}$$

BugLocator combines a classic VSM model and document length score. In our approach, we propose a modified rVSM model (Eq. (8)) to calculate the similarity between source files and a bug report by utilizing the structured information of the source files. We adapted the approach of Saha et al. [20], because utilizing the structured information of source files improves the accuracy of bug localization at the file level. BLUiR breaks a source file into four documents (class, method, variable and comments) and breaks a bug report into the summary and description, as two queries. Each of these parts can be converted into a vector following Eq. (1) through Eq. (3).

$$cos(\vec{f_p}, \vec{b_p}) = \frac{\vec{f_p} \bullet \vec{b_p}}{|\vec{f_p}||\vec{b_p}|} \tag{4}$$

$$s(\vec{f}, \vec{b}) = \textstyle\sum_{b_p \in b} \sum_{f_p \in f} (cos(\vec{f_p}, \vec{b_p}) \times w_{f_p}) \tag{5}$$

$$len(f_c) = \frac{1}{1 + e^{-N(f_c)}} \tag{6}$$

$$N(X) = \frac{x - x_{min}}{x_{max} - x_{min}} \tag{7}$$

$$StructVsmScore(\vec{f}, \vec{b}) = len(f_c) \times s(\vec{f}, \vec{b}) \tag{8}$$

$\vec{f_p}$ is a vector that represents an individual structured information part of a source file, and $\vec{b_p}$ is a vector in which represents each part of a bug report. The relevance score between $\vec{f_p}$ and $\vec{b_p}$ is computed as the cosine similarity (Eq. (4)). Saha et al. performed a separate search for each combination of the source file and bug report parts [20]. They then combined the relevance scores with the sum. However, comments contain many more terms than other parts. This decreases the performance according to our empirical experiments. Thus, we propose a weighted combination of the relevance scores, as given in Eq. (5). w_{f_p} is a weight value for a part of the source file. We assign a 50% weight to the comments part heuristically and a 100% weight to the other source file parts. These weight values are required in order to investigate the influence of each structured part later. Eq. (6) is the function used to calculate the length score of a source file. f_c is the number of terms in a source file. We normalize the value of f_c, and then the output value is used as the input for the exponential function, e^{-x}. The function ensures that larger documents are given higher scores during ranking. The normalization function is defined as in Eq. (7). X is a set of data, and x_{max} and x_{min} are the maximum and minimum data in X, respectively. This is the same procedure used by BugLocator [19].

3.6. Analyzing similar bug reports

BugRepoIndexer parses fixed bug reports and then creates an index in the internal database. Compared with BLIA v1.0, BugRepoIndexer is extended to analyze comments in bug reports. BugRepoAnalyzer analyzes the similarity between the new bug report and fixed reports. It calculates the similarity score of previously fixed bug reports. Only the summary and description of the new bug report are used as queries, while the summary, description and comments of previous bug reports are all included; this is because comments for new bug reports are not included when the bug report is submitted. This approach is based on the assumption that fixed files of similar bug reports can be used to fix the new bug report.

$$SimiBugScore(\vec{f},\vec{b}) = \textstyle\sum_{b' \in \{b' | b' \in B \land f \in b'_{fix}\}} \frac{cos(\vec{b},\vec{b'})}{|b'_{fix}|} \tag{9}$$

B is the set of fixed bug reports before a new bug report b, and b' is the fixed bug report in B. b'_{fix} represents the fixed files of b'. We also follow Eq. (9) of BugLocator [19].

3.7. Analyzing stack trace information in the bug report

Although not all bug reports contain stack traces, they could be the key to resolving a reported bug due to the information regarding the exception location. The files that appear in the stack traces or the imported files of the files that appear directly may be buggy. Wong et al. [23] proposed BRTracer, based on the analysis of stack traces. Moreno et al. [24] proposed Lobster for analyzing the structural similarity between stack trace and code elements. We apply Eq. (10) of BRTracer due to its simplicity and low computation cost.

In Eq. (10), b_{strace} is the set of files in the stack traces and $b_{strace.import}$ is the set of imported files of b_{strace}. BugRepoIndexer extracts all file names in the stack traces using a regular expression. StackTraceAnalyzer directly analyzes the files that appear and the imported files, and then calculates their StraceScores.

$$StraceScore(f,b) = \begin{cases} \frac{1}{rank} & f \in b_{strace} \land rank \leq 10 \\ 0.1 & f \in b_{strace} \land rank > 10 \\ 0.1 & f \in b_{strace.import} \\ 0 & otherwise \end{cases} \tag{10}$$

3.8. Analyzing code change history information

We make use of various bug prediction studies [33–35] as well as Eq. (11) and Eq. (12), from Wang et al. [21]. Wang et al. show that this algorithm is fast and simple, and they adapted this algorithm based on experiments performed by Google developers [35]. Thus, we decided to adapt this well-tested algorithm from Google into CommitLogCollector. Basically, CommitLogCollector collects commit log messages from the SCM repository, and then extracts not only committed files but also modified methods using the difference in information from each commit. ScmRepoAnalyzer analyzes relevant commit logs before a bug report is reported. Instead of all commit logs, filtered commit logs are analyzed after matching the following regular expression: (.*fix.*)|(.*bug.*)|(.*issue.*)|(.*fail.*)|(.*error.*). This regular expression filters commit logs containing "fix", "bug", "issue", "fail" or "error". We form a hypothesis that the commit logs used to fix prior bugs are suspicious and can actually cause new bugs. BLIA v1.0 only filters "fix" or "bug". The filtered words of the current BLIA were extended as described above. With the k parameter, we configure an analysis range of the commit logs in the past k days. Eq. (11) and Eq. (12) calculate the suspicious score of related commit logs; c is a commit, C refers to the set of relevant commits in the past k days, and d_c is the number of elapsed days between a commit, c, and a new bug report.

$$CommScore(f,b,k) = \textstyle\sum_{c \in C \land f \in c} \frac{1}{1+e^{12(1-(\frac{k-d_c}{k}))}} \tag{11}$$

$$CommScore(m,b,k) = \textstyle\sum_{c \in C \land m \in c} \frac{1}{1+e^{12(1-(\frac{k-d_c}{k}))}} \tag{12}$$

3.9. Combining analyzed scores at the file level

In Fig. 5, the BLIA module at the top layer integrates the four analyzed scores in previous subsections, and then computes the suspicious score of each source file for a new bug report using Eq. (13) and Eq. (14). Basically, we adapt both Eq. (13) and Eq. (14) from BugLocator [19], BRTracer [24] and AmaLgam [21]. Using α and β, we configure the influence rate of each analyzed score. Four analyzed scores are not simply combined as a weighted sum, because each score is not always valid and meaningful for every bug report. Thus, we use two control parameters, α and β, to configure the influence rate for scores that are always valid. The influence rates of StructVsmScore, SimiBugScore, StraceScore and CommScore are changed while α and β are controlled. Here, k is the time period for searching related commit logs for the CommScore analysis.

First of all, we combine StructVsmScore and SimiBugScore with α after normalization of the scores. We then add StraceScore to boost the source files in the stack traces without a specific control parameter, as in Eq. (13), because stack traces are not included in all bug reports. Thus, StraceScore is a valid value for when a bug report includes stack traces. Finally, we combine the intermediate score, $c(f,b,\alpha)$, and CommScore with β only if $c(f,b,\alpha)$ is greater than 0 in Eq. (14). Otherwise, the final score is 0 because of the lack of relevance between the source file and the bug report [21].

$$c(f,b,\alpha) = \big(1-\alpha\big) \times N(StructVsmScore(f,b)\big) \\ +\alpha \times N\big(SimiBugScore(f,b)\big) + StraceScore(f,b) \tag{13}$$

$$BliaFileScore(f,b,\alpha,\beta,k) = \\ \begin{cases} (1-\beta) \times c(f,b,\alpha) + \beta \times CommScore(f,b,k) & , c(f,b,\alpha) > 0 \\ 0 & , otherwise \end{cases} \tag{14}$$

3.10. Analyzing method information

For bug localization at the method level, MethodAnalyzer analyzes the TF and IDF of each term, and then calculates the vector values for each method in the ranked suspicious files from Section 3.9 and the bug reports. We analyze only the method names extracted from the top 10 ranked files to reduce the computation time. We also assume that the developers do not want to check suspicious files outside the top 10 ranks. Then, during preprocessing of the methods as described in Section 3.5, the words are treated as the query, and the bug reports to be searched are the document collection.

$$\vec{m} = (tf_f(z_1)idf(z_1), tf_f(z_2)idf(z_2), \cdots, tf_f(z_q)idf(z_q)) \tag{15}$$

$$\vec{b} = (tf_b(y_1)idf(y_1), tf_b(y_2)idf(y_2) \cdots, tf_b(y_r)idf(y_r)) \tag{16}$$

$$MthVsmScore(\vec{m},\vec{b}) = cos(\vec{m},\vec{b}) = \frac{\vec{m} \bullet \vec{b}}{|\vec{m}||\vec{b}|} \tag{17}$$

Table 4
Details of subject projects.

Project	Description	Period of registered bug (Used source version)	#Bugs	#Source Files
AspectJ	Aspect-oriented extension to Java	12/2002∼07/2007 (org.aspectj-1_5_3_final)	284	5188
SWT	Widget toolkit for Java	04/2002∼12/2005 (swt-3.659)	98	738
ZXing	Barcode image processing library	03/2010∼09/2010 (Zxing-1.6)	20	391

Assume that a method (m) and a bug report (b) are represented by weighted term frequency vectors $\vec{m}$ and $\vec{b}$ with q and r as the total number of terms, respectively. The TF and IDF in Eq. (15) and Eq. (16) are calculated using Eq. (3). The relevance score between $\vec{m}$ and $\vec{b}$, MthVsmScore, is computed as the cosine similarity according to Eq. (17).

3.11. Combining analyzed scores at the method level

Finally, the BLIA module integrates the analyzed scores in Section 3.8 and Section 3.10. Then, it computes the final suspicious score of each method for a new bug report, as in Eq. (18). Using γ, we configure the influence rate of MthVsmScore and CommScore. As mentioned in Section 3.8, k controls the time period for searching for related commit logs for the CommScore analysis at the method level. After normalization of MthVsmScore, we then combine the normalized MthVsmScore and CommScore for the method level of bug localization.

$$BliaMethodScore(m,b,\gamma,k) = \\ \big(1-\gamma\big) \times N(MthVsmScore(m,b)\big) \\ + \gamma \times CommScore(m,b,k) \tag{18}$$

4. Experimental setup

4.1. Subject projects

Table 4 presents detailed information for the projects used in our experiments. We used three open-source projects (AspectJ, SWT, and ZXing) together with the information on the fixed files for those bugs, used in previous approaches, for a better comparison. We applied the dataset used by Zhou et al. [19] in BugLocator and by Wong et al. [23] in BRTracer. Because not all datasets for the subject projects are available on the online site of Zhou et al., we used another available dataset from Wong et al. We checked the validity of each bug report status and fixed date. To extend the bug repository data, we checked all bug reports and searched the fixed commits for them. Then, we implemented a utility program to extract modified methods from each commit using the 'git diff' command, as mentioned in Section 3.4. We downloaded the source files that included the fixed files of the

Table 5
Comparison between including comments and excluding comments of new bug reports at the file level.

Project	Case	Top1 (%)	Top5 (%)	Top10 (%)	MAP	MRR
AspectJ (α=0.3, β=0.2, k=120)	Including new bug's comments	**41.5**	**71.1**	**80.6**	**0.39**	**0.55**
	Excluding new bug's comments	40.0	67.0	75.4	0.35	0.52
SWT (α=0.0, β=0.0, k=120)	Including new bug's comments	**67.3**	**86.7**	**89.8**	**0.65**	**0.75**
	Excluding new bug's comments	65.3	83.7	86.7	0.62	0.73
ZXing (α=0.2, β=0.0, k=120)	Including new bug's comments	**55.0**	**75.0**	**80.0**	**0.62**	**0.64**
	Excluding new bug's comments	50.0	65.0	70.0	0.54	0.58

Table 6
Comparison with other approaches at the file level.

Project	Approach	Top1 (%)	Top5 (%)	Top10 (%)	MAP	MRR
AspectJ	**BLIA v1.5** (α=0.3, β=0.2, k=120)	41.5	**71.1**	**80.6**	**0.39**	**0.55**
	BLIA v1.0	37.7	64.4	73.2	0.32	0.49
	BugLocator	30.8	51.1	59.4	0.22	0.41
	BLUiR	33.9	52.4	61.5	0.25	0.43
	BRTracer	39.5	60.5	68.9	0.29	0.49
	AmaLgam	**44.4**	65.4	73.1	0.33	0.54
SWT	**BLIA v1.5** (α=0.0, β=0.0, k=120)	67.3	**86.7**	**89.8**	**0.65**	**0.75**
	BLIA v1.0	**68.4**	82.7	89.8	0.64	0.75
	BugLocator	39.8	67.4	81.6	0.45	0.53
	BLUiR	56.1	76.5	87.8	0.58	0.66
	BRTracer	46.9	79.6	88.8	0.53	0.60
	AmaLgam	62.2	81.6	89.8	0.62	0.71
ZXing	**BLIA v1.5** (α=0.2, β=0.0, k=120)	**55.0**	**75.0**	**80.0**	**0.62**	**0.64**
	BLIA v1.0	50.0	60.0	80.0	0.51	0.57
	BugLocator	40.0	60.0	70.0	0.44	0.50
	BLUiR	40.0	60.0	70.0	0.39	0.49
	BRTracer	N/A	N/A	N/A	N/A	N/A
	AmaLgam	40.0	65.0	70.0	0.41	0.51

last fixed bug and commit logs from the Git repositories of the three projects. While checking the validity of the AspectJ dataset, we excluded two bugs because of their invalidity; these are described in detail in Section 6.

4.2. Evaluation metrics

To evaluate the effectiveness of the proposed bug localization method, we considered the following three metrics: top N rank, mean average precision (MAP) and mean reciprocal rank (MRR). MAP and MRR are popular metrics for evaluating IR techniques. The top N rank is widely used to evaluate IR-based bug localization.

Top N rank: This metric calculates the number of bug reports in which at least one buggy source file or method was found and ranked in the top $N(=1,5,10)$ returned results. For example, given a bug report, if at least one file or method in which the bug is fixed is found in the top N results, we consider the bug to be localized successfully in the top N rank. This metric emphasizes early precision over the total ranked count.

Mean Average Precision (MAP): MAP is the most commonly used IR metric for evaluating ranking approaches. All ranked files or methods are considered. Therefore, MAP emphasizes all of the ranked suspicious files or methods instead of only the first. MAP is the mean of the *average precision* values for all queries. The average precision of a single query is computed as follows:

$$AP = \sum_{i=1}^{M} \frac{P(i) \times pos(i)}{number\ of\ positive\ instances} \tag{19}$$

$$P(i) = \frac{\#buggy\ files}{i} \tag{20}$$

where i is a rank of the ranked suspicious objects, M is the number of ranked objects, $pos(i)$ is a binary indicator of whether the i^{th} ranked object is a buggy object, and $P(i)$ is the precision of the buggy files or methods at the given top i rank.

Mean Reciprocal Rank (MRR): The reciprocal rank of a query is the reciprocal of the position for the first buggy file or method in the results that is ranked as suspicious. MRR is the mean of the reciprocal ranks of the results of a set of queries, Q, and it can be calculated as follows:

$$MRR = \frac{1}{|Q|} \sum_{|Q|}^{i=1} \frac{1}{rank_i} \tag{21}$$

The top N rank metric does not consider the overall quality, while MAP and MRR cover the overall quality of ranked suspicious results. Larger values of all metrics indicate better accuracy.

In addition, we performed some statistical tests to confirm that the improvement is statistically significant rather than by chance. We used the Wilcoxon Rank-Sum test [36] to test the following null hypothesis:

1. There is no statistically significant difference between the MAP for BLIA v1.5 and each of the previous approaches.

2. There is no statistically significant difference between the MRR for BLIA v1.5 and each of the previous approaches.

Basically, we use a one-tailed test as we are interested in testing whether BLIA v1.5 introduces significant improvement in the MAP or MRR. We then perform the Bonferroni correction to account for performing multiple comparisons of the approaches. There are five comparisons (BLIA v1.5 vs. BugLocator, BLUiR, BRTracer, AmaLgam, and BLIA v1.0) for each hypothesis. The Bonferroni correction compensates for the large number of comparisons by testing each individual hypothesis at a significance level of α/n, where α is the desired overall alpha level and n is the number of testings or comparisons. Thus, if n is 5 with a desired $\alpha = 0.05$, then the Bonferroni correction would test each individual comparison at $\alpha = 0.05/5 = 0.01$.

We also estimated the magnitude of the difference between means. We used Cliff's delta (or d) [37], a non-parametric effect size measure for ordinal data. The effect size is considered small for $|d| < 0.33$, medium for $|d| < 0.474$, and large for all other values. We chose Cliff's delta, d, as the effect size because it is appropriate for our variables and given the different levels (small, medium, large) defined for it, it is quite easy to interpret.

5. Experimental results

This section describes the evaluation of BLIA at the file and method levels while performing bug localization on the subject projects shown in Table 4. We answer four research questions, which assess the effectiveness of BLIA in comparison with other IR-based bug localization techniques.

1. ***RQ1***: At the file level, can BLIA outperform other bug localization tools?
2. ***RQ2***: What is the best parameter value for each score at the file level?
3. ***RQ3***: Which score has a greater effect on bug localization at the file level?
4. ***RQ4***: At the method level, can BLIA outperform other bug localization tools?

5.1. Bug localization at the file level

Prior to comparison with other approaches, we compared two cases of new bug reports in Table 5. The first case is a new bug report without appended comments. It is an initial snapshot of a new bug report. The second case is a new bug report with appended comments. It is the last snapshot of a new bug report. Comments are included to the description section of the bug report for this case. When a new bug report is submitted, it may not include enough information for developers to resolve it. Thus, the developers can ask the reporter for the details on the reported scenario, environment or other information by using the comments in the bug report. In addition, the reporter can also append additional information to the bug report using comments. Some comments include stack traces or a more detailed scenario for bug reports. Table 5 clearly shows the results in which a new bug report that included comments improved the accuracy of bug localization. On average, including comments improves the MAP and MRR scores by 10% and 6%, respectively, compared to bug reports in which comments are excluded. Based on these comparison results, we included comments in new bug reports for all of the following experiments.

For ***RQ1***, Table 6 shows the results of BLIA and other IR-based bug localization models, i.e., BugLocator [19], BLUiR [20], BRTracer [23], and AmaLgam [21]. In the case of BRTracer, the results of ZXing are not included because BRTracer did not use the project for their experiments. The optimized α, β, and k values for BLIA v1.5 are found from the results in Section 5.2 and depend on the data in each project. More specifically, β and k for CommScore depend on the commit description quality and release cycle of the product. Thus, the accuracy results of IR-based bug localization models depend on the quality of bug reports and commit comments. This is the main limitation of IR-based bug localization methods, in that they are based on information retrieval. On average, BLIA v1.5 improves the MAP scores of BugLocator, BLUiR, BRTracer, AmaLgam and BLIA v1.0 by 54%, 42%, 30%, 25% and 15%, respectively. In terms of MRR, BLIA v1.5 outperforms BugLocator, BLUiR, BRTracer, AmaLgam and BLIA v1.0 by 34%, 24%, 19%, 11% and 8%, respectively. We note that BLIA consistently outperforms other techniques in terms of MAP. Compared with BLIA v1.0, BLIA v1.5 analyzes extended inputs, i.e., comments in bug reports and modified methods in the code change history. Therefore, an additional comment analysis in bug reports improves the accuracy of bug localization. This is reason-

Table 7
Statistical analysis results of Wilcoxon Rank-Sum test (p-value) and Cliff's delta (d).

Data set	Compared approach	p-value	d	Effect size
MAP	vs. BugLocator	0.2	0.56	large
	vs. BLUiR	0.13	0.67	large
	vs. BRTracer	0.2	0.25	small
	vs. AmaLgam	0.25	0.44	medium
	vs. BLIA v1.0	0.35	0.33	medium
MRR	vs. BugLocator	0.05	0.67	large
	vs. BLUiR	0.2	0.44	medium
	vs. BRTracer	0.2	0.25	small
	vs. AmaLgam	0.2	0.44	medium
	vs. BLIA v1.0	0.35	0.33	medium

able because developers usually search not only the summary and description, but also the comments of previous bug reports to resolve a new bug report.

Table 7 reports the results of the Wilcoxon Rank-Sum test (p-value) and Cliff's delta. After the Bonferroni correction, the results are interpreted as statistically significant at an α of 0.01. However, from the result table, all p-values of the results cannot reject the null hypothesis. We could not show that our improvement is statistically significant. To alleviate this, we will extend our proposed approach to other projects in the future. Despite this result, the effect sizes, d, are medium or large for all comparisons except BRTracer. The effect sizes of MAP and MRR between BLIA v1.5 and BugLocator are particularly all large.

5.2. Optimized combination of analyzable inputs

We performed numerous experiments while varying parameters to find the optimized combination and impact of each score. To find the optimum α for BLIA, α was varied (with β fixed at 0.0 and k fixed at 120 to protect against other impacts); the results of MAP and MRR are presented in Fig. 6. The MAP and MRR scores remain stable when α is between 0.0 and 0.4. Next, we investigate the impact of various values of β with an α that is fixed at the abovementioned optimized values. Fig. 7 shows the results in terms of MAP and MRR. The pattern of each project is slightly different. The highest MAP and MRR scores are recorded when β is between 0.0 and 0.2.

On the basis of the aforementioned first analysis, we analyze the relationship between α and β in terms of MAP and MRR while varying α and β from 0.0 to 0.9. Fig. 8 shows 3D surface graphs of MAP and MRR across subject projects. Each graph contains three axes, x, y and z. The x-axis represents α, the y-axis is β and the z-axis is MAP or MRR. From these graphs, we can determine the pattern and distribution of MAP and MRR visually while α and β are varied. In the case of AspectJ with 284 bug reports, we can find the highest MAP and MRR score areas for when α ranges from 0.1 to 0.5 and β ranges from 0.1 to 0.3. Additionally, varying β has a larger impact on MAP and MRR scores than varying α. For SWT, the highest MAP and MRR score area occurs when α ranges from 0.0 to 0.2 and β ranges from 0.0 to 0.2. For ZXing, there is little change in the MAP and MRR scores when α ranges from 0.0 to 0.4 and β ranges from 0.0 to 0.1. Moreover, across the three subject projects, the highest MAP and MRR score areas become narrow and distinct as the number of bug reports and the project size increase. In the future, we plan to perform experiments using different project sizes to extend the generality of the relationship between α and β in terms of MAP and MRR scores.

Fig. 9 shows the MAP and MRR scores with varying k values. The MAP and MRR scores remain stable at k values of 30, 60, 90, 120, 150 and 180. For all subject projects, the MAP and MRR scores are high when k is 120.

5.3. Influence rate of each type of analyzed data

For ***RQ2*** in the introduction of Section 5, we investigate the impact of each analyzed score with the optimized α and β values in Fig. 10. The influence rate is the portion of each analyzed score in the integrated score, BliaFileScore. This rate value depends on α and β. We calculate the influence rate of the analyzed scores to obtain the highest BliaFileScore for each project using the optimized α and β. Fig. 10 presents the average influence rate of each analyzed score from the experimental results. As introduced previously in Section 3, StraceScore is the sum of StructVsmScore and SimiBugScore. The source files in the stack trace are boosted only when a bug report contains stack traces [23]. We analyze the influence rate for two cases, *excluding and including the stack trace*, in Fig. 10. In the first case where the stack trace is excluded, the similarity between the bug report and source files (StructVsmScore) is 76%, the highest score. The similarity of the previously fixed bug reports (SimiBugScore) is 7% and that of the code change history (CommScore) is 17%. In the second case, where the stack trace is included, the stack trace analysis data is 47%, which is the highest value, StructVsmScore is 39%, SimiBugScore and CommScore are 7%. Through this investigation, stack trace accounts for the biggest portion of the integrated suspicious score for bug localization. Thus, if stack trace information is provided to submit a new bug report, it is mandatory data needed to improve the accuracy of IR-based bug localization approaches. Additionally, the summary and description in a bug report need to be written in detail using module names (classes, methods or variables). Furthermore, class, method, and variable names in source files should be named using human-readable words to improve the calculation of similarity between the new bug report and source files.

5.4. Bug localization at the method level

To answer ***RQ4***, Table 8 shows the results of BLIA v1.5 at the method level. We attempted to compare these with Le et al. [38] at the method level; however, Le et al. used a different data set that is not publicly accessible, and requires program execution for spectrum-based bug localization. Thus, an exact comparison is not possible. BLIA v1.5 statically analyzes source files, bug reports and code change histories without program execution. This reduces the computational resources required to localize bugs.

Furthermore, not all bug reports are resolved by modification of practical code lines in source files. For example, javadoc is fixed for documentation errors, and related member variables are fixed for property defects. Fig. 11 shows MAP and MRR scores with varying γ values. When γ is changed from 0.0 to 0.9, the MAP and MRR scores only slightly change. Thus, it is difficult to assess the highest value of γ. However, we discovered the potential of our bug localization approach at the method level. In

Table 8
Result of BLIA v1.5 at the method level.

Project	Approach	Top1 (%)	Top5 (%)	Top10 (%)	MAP	MRR
AspectJ	**BLIA v1.5** (α=0.3, β=0.2, γ=0.4, k=120)	6.0	14.8	21.5	0.08	0.11
SWT	**BLIA v1.5** (α=0.0, β=0.0, γ=0.5, k=120)	10.2	20.4	32.7	0.10	0.17
ZXing	**BLIA v1.5** (α=0.2, β=0.0, γ=0.3, k=120)	15.0	25.0	30.0	0.20	0.20

the future, we aim to further improve and generalize our approach.

6. Threats to validity

In this section, we discuss the limitations, validity and generalizability of our BLIA findings. In particular, we cover internal validity, external validity, construct validity, and reliability.

6.1. Internal validity

We used the same dataset from Zhou et al. [19] and Wong et al. [23]. The dataset contains bug reports from three subject projects. While the data may contain errors, we reviewed all bug reports for all subject projects; two invalid bugs were found (e.g., bugs in AspectJ: #64069, #104218). The status of the first is "REOPENED" and that of the other is "NEW"[8]. While checking information about bugs in the dataset, we also found numerous bugs whose fixed dates do not match the bug repository. The original fixed date of the bug reports in the dataset matches the last modified date of the bug reports. We modified the fixed date of these erroneous bug reports in the dataset. We then downloaded the source files and commit logs from the Git repositories of those three projects. We verified the extended comments and fixed files/methods for all bug reports in the extended bug repository data.

6.2. External validity

All subject projects for our experiments are open-source projects that were used in previous research. Although they are popular open-source projects, our work may not be generalizable to other open-source projects or industrial projects. Moreover, we performed statistical tests (Wilcoxon Rank-Sum test) to assess the improvement of our approach, but we could not show that our improvement is statistically significant as mentioned in Section 5.1. To alleviate this, in the future we will investigate our proposed approach and expand to other open-source projects (e.g., Eclipse) and industrial projects and evaluate them at the file and method levels. Then we will perform statistical tests to show that the improvement of our approach is statistically significant.

[8]The last checked date of the status is at 01/Mar/2016.

6.3. Construct validity

We use three evaluation metrics, i.e., Top N rank, MAP and MRR, in our experiments. These metrics have been widely used before to evaluate previous approaches [19–21, 23] and are well-known IR metrics. Thus, we argue that our research has strong construct validity.

6.4. Reliability

In Section 5.2, we performed numerous experiments using various combinations of control parameters to find the optimum parameters and the best accuracy of bug localization for each project. The optimized α, β, and k values are based on the experiments and are only for BLIA. However, different parameters will result in different accuracy, not only for BLIA but also for other tools. The other tools also found the best parameters for the best accuracy in a similar manner. In addition, they do not use the same control parameters. For example, BugLocator uses α to combine rVSMScore and SimiScore. rVSMScore and SimiScore of BugLocator are mapped with StructVsmScore and SimiBugScore of BLIA, respectively. The α value is 0.2 or 0.3 for the experiment projects. BLUiR uses two parameters, but they are not for combining scores. BRTracer uses α to combine rVSM$_{seg}$ and SimiScore. rVSM$_{seg}$ and SimiScore of BRTracer are mapped with StructVsmScore and SimiBugScore of BLIA, respectively. The value of α is 0.2 for the experiment projects. AmaLgam uses α and β to combine internal scores. The α value is set to 0.2 to combine $\text{Susp}^S(f)$ and $\text{Susp}^R(f)$. The β value is 0.3 to combine $\text{Susp}^{S,R}(f)$ and $\text{Susp}^H(f)$. $\text{Susp}^S(f)$ and $\text{Susp}^R(f)$ of AmaLgam are mapped with StructVsmScore and SimiBugScore of BLIA, respectively. Additionally, $\text{Susp}^H(f)$ is mapped with CommScore of BLIA. To alleviate this and automatically optimize control parameters for target projects, in the future we will expand our proposed approach using machine learning methods or generic algorithms.

7. Related work

7.1. Change impact analysis

Software change impact analysis (CIA) is a technique used to identify the effects of changes or to estimate what needs to be modified [5]. Several approaches of CIA were proposed and they can be categorized as a static analysis or dynamic analysis [7–11]. A representative CIA approach from Chianti [6] is a CIA tool for Java that analyzes two versions of an application and decomposes their differences. Uncovered source code changes can cause various defects. Thus, CIA techniques can be applied to bug localization. FaultTracer [12] is a change impact and regression fault analysis tool for evolving programs. It adapts spectrum-based ranking techniques to CIA to reduce developers' efforts.

7.2. Spectrum-based bug localization

In other fault localization approaches, researchers have proposed various spectrum-based fault localization methods by examining a small portion of the source code. These techniques require program runtime execution traces and usually analyze the program spectra information (such as program execution statistics) between passed and failed executions to calculate the suspiciousness score of the fault at various granularity levels (e.g., statements and branches) [13–16]. Gopinath et al. [39] proposed a technique that combines a specification-based analysis with a spectrum-based fault localization technique to overcome the limitation of spectrum-based fault localization methods. Laghari et al. [40] proposed a variant of spectrum-based fault localization with patterns of method calls by means of frequent itemset mining. Le et al. [41] proposed Savant, a new fault localization approach that uses a learning-to-rank machine learning approach to identify buggy methods from failures by analyzing both classic suspiciousness scores and inferred likely invariants observed in passing and failing test cases. Sun et al. [42] algebraically and probabilistically analyzed the metrics of spectrum-based fault localization, and then they suggested the most effective metrics based on their analysis results.

7.3. IR-based bug localization

Bug reports are the starting point and key to localizing bugs. Researchers have proposed IR-based bug localization to reduce the time and costs of fixing bugs in recent years [19–24, 26, 38, 43–45]. IR-based bug localization techniques combine available information to improve the accuracy of suspicious ranking. Examples include the analysis of stack traces in a bug report, structured information of source files, segmenting source files, and program dependency graphs. Data mining of a bug/issue management system and analysis of the source code change history are integrated with previous techniques [21, 31, 32, 46–48]. Moreover, there is a practical application program to show ranked suspicious files when a new bug report is registered, applying an IR-based bug localization technique to a bug/issue management system [49]. To compare the performance of IR-based bug localization, an experimental platform for researchers, BOAT, is available [50]. In the near future, various comparison studies of IR-based bug localization techniques are expected. In addition, datasets of open-source projects are available online [51, 52].

Recently, integration of spectrum-based approaches and IR-based approaches to improve the granularity of bug localization has been proposed [38]. Using deep learning techniques, Lam et al. [53] suggested an approach that uses a deep neural network (DNN) in combination with rVSM, an IR technique. rVSM collects the feature on the textual similarity between bug reports and source files. A DNN is used to learn and relate the terms in bug reports to potentially different code tokens and terms in source files and documentation if they appear frequently enough in the pairs of reports and buggy files. Ye et al. [54] proposed bridging the lexical gap by projecting natural language statements and code snippets to improve IR-based bug localization. Word embedding techniques are also applied in this approach. Word embeddings are trained on related software documents (API documents, tutorials, and reference documents), and then aggregated in order to estimate the semantic similarities between documents. Sisman et al. [55] proposed an IR framework using spatial code proximity and term ordering relationships in a code base for improved IR accuracy.

8. Conclusion

During the software maintenance phase, the occurrence of a suspicious defect in a bug report is delivered to the software development team. A large amount of maintenance time and cost are required to resolve reported bugs. Approaches on modeling the quality of bug reports as well as research on how to create better bug reports have been proposed [56–58]. However, it is difficult to standardize the style of bug reports written in natural languages to improve the accuracy of bug localization. Recently, researchers have proposed various approaches for automatic bug localization by using IR and data mining.

We proposed a new approach, BLIA, which is a statically integrated analysis approach for IR-based bug localization. We analyzed texts, stack traces and comments in bug reports, structured information of source files and the source code change history, which were all used differently in previously proposed approaches. After analyzing each data source, we combined all analyzed scores and displayed files that are localized and ranked as suspicious. From the ranked suspicious files, we analyzed the similarity of the various methods in terms of the files, bug reports, and method change histories from the SCM repository. Finally, we combined the analyzed scores at the method level, and then suspicious and ranked methods were used as outputs to localize bugs. We determined BLIA's potential for improving the accuracy of bug localization at the method level. By investigating various combinations of information, we analyzed the influence rate of each score on BLIA. For our experiments, bug repositories were extended. The comments, fixed methods and fixed commits for each bug report were inserted and are made available for improving the accuracy of bug localization research.

In the future, we aim to explore the following areas and further reduce the threats to the external validity.

1. Various studies regarding reviewer recommendations [59] or similar bug report recommendations [60] that are being used to resolve bug reports have been published. We plan to generalize and extend BLIA as a bug localization platform for application to other projects, which

will provide related information from analyzed BLIA data to support bug resolution activities.

2. BugLocalizer [49] is integrated into a bug/issue management system (e.g., Bugzilla) to support bug localization. Basically, the BugLocator approach was applied to Bugzilla. It automatically shows suspiciously ranked files and suspicious scores when a new bug is reported. We will integrate BLIA into bug/issue management systems and then apply not only other open-source projects but also industrial software projects to ensure the generality of the BLIA method.

Researchers interested in our approach for improving IR-based bug localization can access all datasets, all results and our proposed tools, i.e., BLIA v1.0 and BLIA v1.5, at the following link:

https://github.com/klausyoum/BLIA

Acknowledgment

This research was supported by the Next-Generation Information Computing Development Program through the National Research Foundation of Korea(NRF) funded by the Ministry of Education, Science and Technology(No. 2012M3C4A7033347), and the MISP(Ministry of Science, ICT & Future Planning) of Korea, under the National program for Excellence in Software supervised by the IITP (Institute for Information & Communications Technology Promotion). We are extremely grateful to Zhou et al., Saha et al. and Wong et al. for access to their datasets and results on their web sites, and anonymous reviewers for their feedback and suggestions for improvement.

References

[1] K. C. Youm, J. Ahn, J. Kim, E. Lee, Bug localization based on code change histories and bug reports, in: 2015 Asia-Pacific Software Engineering Conference (APSEC), IEEE, 2015, pp. 190–197.

[2] R. D. Banker, S. M. Datar, C. F. Kemerer, D. Zweig, Software complexity and maintenance costs, Communications of the ACM 36 (11) (1993) 81–95.

[3] M. J. C. Sousa, H. M. Moreira, A survey on the software maintenance process, in: Software Maintenance, 1998. Proceedings., International Conference on, IEEE, 1998, pp. 265–274.

[4] M. K. Davidsen, J. Krogstie, A longitudinal study of development and maintenance, Information and Software Technology 52 (7) (2010) 707–719.

[5] B. Li, X. Sun, H. Leung, S. Zhang, A survey of code-based change impact analysis techniques, Software Testing, Verification and Reliability 23 (8) (2013) 613–646.

[6] X. Ren, F. Shah, F. Tip, B. G. Ryder, O. Chesley, Chianti: a tool for change impact analysis of java programs, in: ACM Sigplan Notices, Vol. 39, ACM, 2004, pp. 432–448.

[7] X. Ren, O. C. Chesley, B. G. Ryder, Identifying failure causes in java programs: An application of change impact analysis, Software Engineering, IEEE Transactions on 32 (9) (2006) 718–732.

[8] J. Wloka, B. G. Ryder, F. Tip, Junitmx-a change-aware unit testing tool, in: Proceedings of the 31st International Conference on Software Engineering, IEEE Computer Society, 2009, pp. 567–570.

[9] M. Acharya, B. Robinson, Practical change impact analysis based on static program slicing for industrial software systems, in: Proceedings of the 33rd international conference on software engineering, ACM, 2011, pp. 746–755.

[10] N. Rungta, S. Person, J. Branchaud, A change impact analysis to characterize evolving program behaviors, in: Software Maintenance (ICSM), 2012 28th IEEE International Conference on, IEEE, 2012, pp. 109–118.

[11] B. Li, X. Sun, J. Keung, Fca–cia: An approach of using fca to support cross-level change impact analysis for object oriented java programs, Information and Software Technology 55 (8) (2013) 1437–1449.

[12] L. Zhang, M. Kim, S. Khurshid, Faulttracer: a spectrum-based approach to localizing failure-inducing program edits, Journal of Software: Evolution and Process 25 (12) (2013) 1357–1383.

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

[14] L. Naish, H. J. Lee, K. Ramamohanarao, A model for spectra-based software diagnosis, ACM Transactions on software engineering and methodology (TOSEM) 20 (3) (2011) 11.

[15] J. A. Jones, M. J. Harrold, Empirical evaluation of the tarantula automatic fault-localization technique, in: Proceedings of the 20th IEEE/ACM international Conference on Automated software engineering, ACM, 2005, pp. 273–282.

[16] W. E. Wong, V. Debroy, R. Gao, Y. Li, The dstar method for effective software fault localization, Reliability, IEEE Transactions on 63 (1) (2014) 290–308.

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

[18] D. Binkley, D. Lawrie, Information retrieval applications in software maintenance and evolution, Encyclopedia of Software Engineering (2010) 454–463.

[19] J. Zhou, H. Zhang, 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, IEEE, 2012, pp. 14–24.

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

[21] S. Wang, 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, ACM, 2014, pp. 53–63.

[22] S. Davies, M. Roper, Bug localisation through diverse sources of information, in: Software Reliability Engineering Workshops (ISSREW), 2013 IEEE International Symposium on, IEEE, 2013, pp. 126–131.

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

[24] L. Moreno, J. J. Treadway, A. Marcus, W. Shen, On the use of stack traces to improve text retrieval-based bug localization, in: 2014 IEEE International Conference on Software Maintenance and Evolution (ICSME), IEEE, 2014, pp. 151–160.

[25] A. Schroter, N. Bettenburg, R. Premraj, Do stack traces help developers fix bugs?, in: Mining Software Repositories (MSR), 2010 7th IEEE Working Conference on, IEEE, 2010, pp. 118–121.

[26] S. Davies, M. Roper, M. Wood, Using bug report similarity to enhance bug localisation, in: Reverse Engineering (WCRE), 2012 19th Working Conference on, IEEE, 2012, pp. 125–134.

[27] G. Antoniol, G. Canfora, G. Casazza, A. De Lucia, E. Merlo, Recovering traceability links between code and documentation, Software Engineering, IEEE Transactions on 28 (10) (2002) 970–983.

[28] C. D. Manning, H. Schütze, Foundations of statistical natural

language processing, Vol. 999, MIT Press, 1999.

[29] D. M. Blei, A. Y. Ng, M. I. Jordan, Latent dirichlet allocation, the Journal of machine Learning research 3 (2003) 993–1022.

[30] S. Deerwester, Improving information retrieval with latent semantic indexing.

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

[32] C. Tantithamthavorn, R. Teekavanich, A. Ihara, K.-i. Matsumoto, Mining a change history to quickly identify bug locations: A case study of the eclipse project, in: Software Reliability Engineering Workshops (ISSREW), 2013 IEEE International Symposium on, IEEE, 2013, pp. 108–113.

[33] S. Kim, T. Zimmermann, E. J. Whitehead Jr, A. Zeller, Predicting faults from cached history, in: Proceedings of the 29th international conference on Software Engineering, IEEE Computer Society, 2007, pp. 489–498.

[34] F. Rahman, D. Posnett, A. Hindle, E. Barr, P. Devanbu, Bugcache for inspections: hit or miss?, in: Proceedings of the 19th ACM SIGSOFT symposium and the 13th European conference on Foundations of software engineering, ACM, 2011, pp. 322–331.

[35] C. Lewis, R. Ou, Bug prediction at google (2011). URL `http://google-engtools.blogspot.sg/2011/12/bug-prediction-at-google.html`

[36] W. Conover, Practical nonparametric statistics, 3rd Edition, Wiley series in probability and statistics, Wiley, 1999.

[37] R. J. Grissom, J. J. Kim, Effect sizes for research: a broad practical approach, Lawrence Erlbaum Associates, 2005.

[38] T.-D. B. Le, R. J. Oentaryo, D. Lo, Information retrieval and spectrum based bug localization: better together, in: Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering, ACM, 2015, pp. 579–590.

[39] D. Gopinath, R. N. Zaeem, S. Khurshid, Improving the effectiveness of spectra-based fault localization using specifications, in: Automated Software Engineering (ASE), 2012 Proceedings of the 27th IEEE/ACM International Conference on, IEEE, 2012, pp. 40–49.

[40] G. Laghari, A. Murgia, S. Demeyer, Fine-tuning spectrum based fault localisation with frequent method item sets, in: Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering, ASE 2016, ACM, 2016, pp. 274–285.

[41] T.-D. B. Le, D. Lo, C. Le Goues, L. Grunske, A learning-to-rank based fault localization approach using likely invariants, in: Proceedings of the 25th International Symposium on Software Testing and Analysis, ISSTA 2016, ACM, 2016, pp. 177–188.

[42] S.-F. Sun, A. Podgurski, Properties of effective metrics for coverage-based statistical fault localization, in: 2016 IEEE International Conference on Software Testing, Verification and Validation (ICST), IEEE, 2016, pp. 124–134.

[43] R. K. Saha, J. Lawall, S. Khurshid, D. E. Perry, On the effectiveness of information retrieval based bug localization for c programs, in: 2014 IEEE International Conference on Software Maintenance and Evolution (ICSME), IEEE, 2014, pp. 161–170.

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

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

[46] S. Wang, D. Lo, J. Lawall, Compositional vector space models for improved bug localization, in: 2014 IEEE International Conference on Software Maintenance and Evolution (ICSME), IEEE, 2014, pp. 171–180.

[47] S. Wang, F. Khomh, Y. Zou, Improving bug localization using correlations in crash reports, in: Mining Software Repositories (MSR), 2013 10th IEEE Working Conference on, IEEE, 2013, pp. 247–256.

[48] M. Gethers, B. Dit, H. Kagdi, D. Poshyvanyk, Integrated impact analysis for managing software changes, in: 2012 34th International Conference on Software Engineering (ICSE), IEEE, 2012, pp. 430–440.

[49] F. Thung, T.-D. B. Le, P. S. Kochhar, D. Lo, Buglocalizer: integrated tool support for bug localization, in: Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering, ACM, 2014, pp. 767–770.

[50] X. Wang, D. Lo, X. Xia, X. Wang, P. S. Kochhar, Y. Tian, X. Yang, S. Li, J. Sun, B. Zhou, Boat: an experimental platform for researchers to comparatively and reproducibly evaluate bug localization techniques, in: Companion Proceedings of the 36th International Conference on Software Engineering, ACM, 2014, pp. 572–575.

[51] S. Rao, A. Kak, morebugs: A new dataset for benchmarking algorithms for information retrieval from software repositories (trece-13-07), Purdue University, School of Electrical and Computer Engineering, Tech. Rep 4.

[52] V. Dallmeier, T. Zimmermann, Extraction of bug localization benchmarks from history, in: Proceedings of the twenty-second IEEE/ACM international conference on Automated software engineering, ACM, 2007, pp. 433–436.

[53] A. N. Lam, A. T. Nguyen, H. A. Nguyen, T. N. Nguyen, Combining deep learning with information retrieval to localize buggy files for bug reports (n), in: Automated Software Engineering (ASE), 2015 30th IEEE/ACM International Conference on, IEEE, 2015, pp. 476–481.

[54] X. Ye, H. Shen, X. Ma, R. Bunescu, C. Liu, From word embeddings to document similarities for improved information retrieval in software engineering, in: Proceedings of the 38th International Conference on Software Engineering, ACM, 2016, pp. 404–415.

[55] B. Sisman, S. A. Akbar, A. C. Kak, Exploiting spatial code proximity and order for improved source code retrieval for bug localization, Journal of Software: Evolution and Process (2016) –.

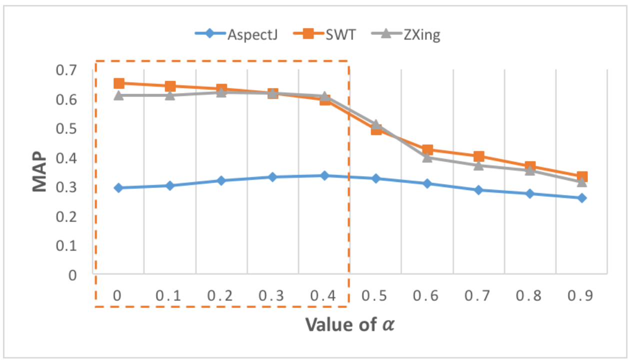
[56] N. Bettenburg, S. Just, A. Schröter, C. Weiß, R. Premraj, T. Zimmermann, Quality of bug reports in eclipse, in: Proceedings of the 2007 OOPSLA workshop on eclipse technology eXchange, ACM, 2007, pp. 21–25.

[57] P. Hooimeijer, W. Weimer, Modeling bug report quality, in: Proceedings of the twenty-second IEEE/ACM international conference on Automated software engineering, ACM, 2007, pp. 34–43.

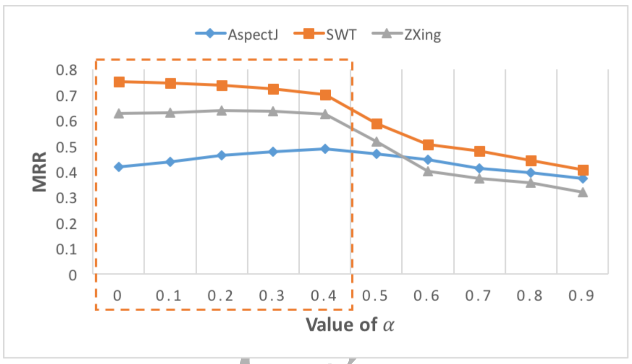
[58] T. Zimmermann, R. Premraj, N. Bettenburg, S. Just, A. Schröter, C. Weiss, What makes a good bug report?, Software Engineering, IEEE Transactions on 36 (5) (2010) 618–643.

[59] Y. Yu, H. Wang, G. Yin, C. X. Ling, Reviewer recommender of pull-requests in github, in: 2014 IEEE International Conference on Software Maintenance and Evolution (ICSME), IEEE, 2014, pp. 609–612.

[60] H. Rocha, G. De Oliveira, H. Marques-Neto, M. T. Valente, Nextbug: a bugzilla extension for recommending similar bugs, Journal of Software Engineering Research and Development 3 (1) (2015) 1–14.

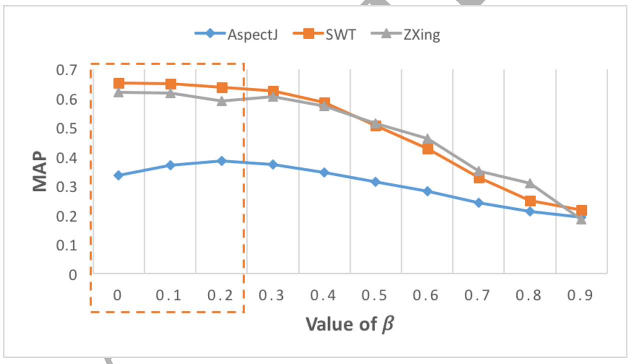


(a) MAP

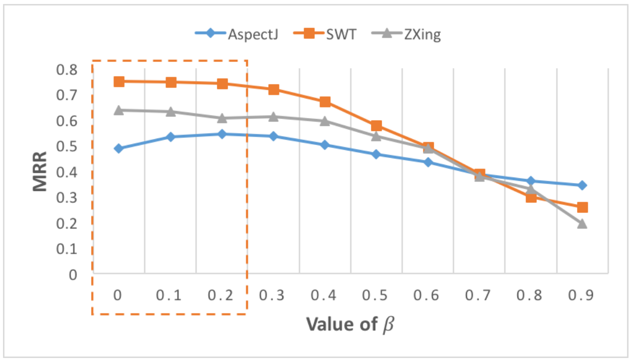


(b) MRR

Fig. 6. Impact of the value of α on BLIA in terms of MAP and MRR.

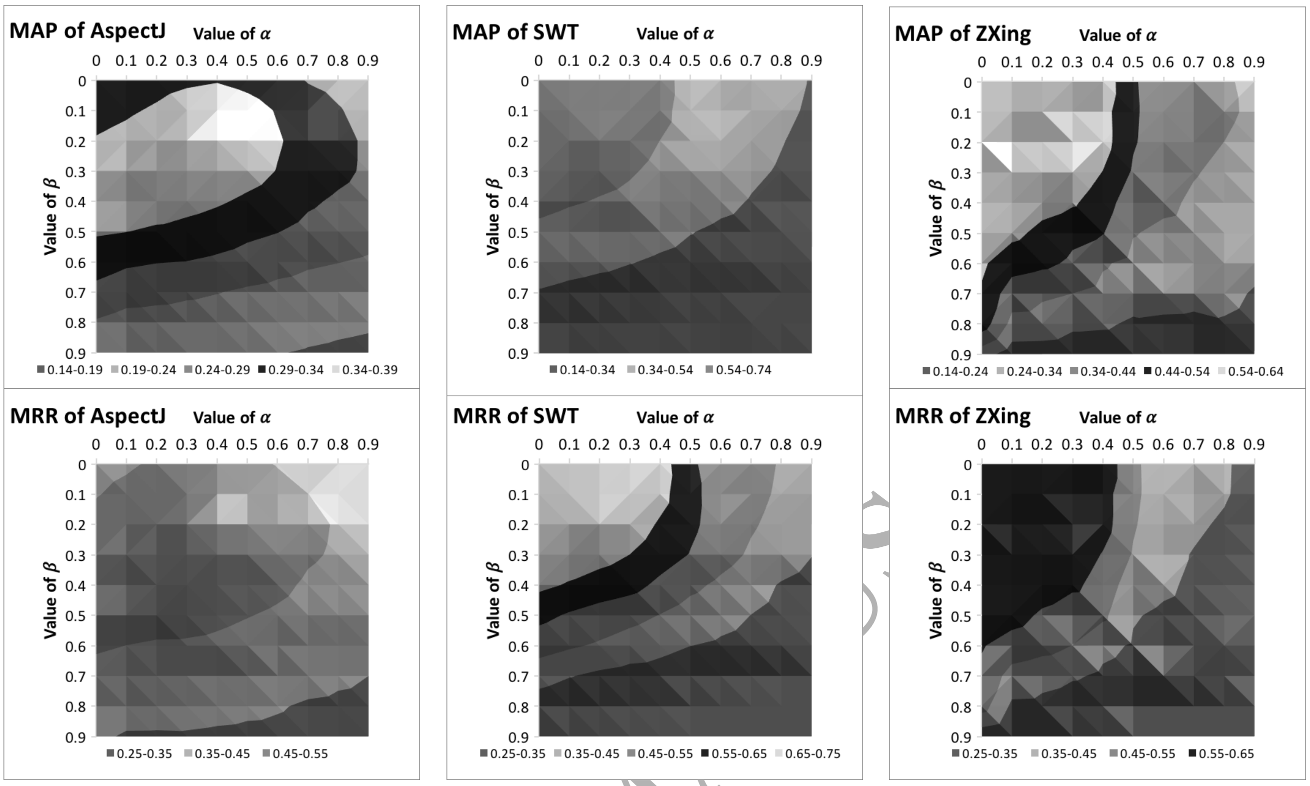


(a) MAP



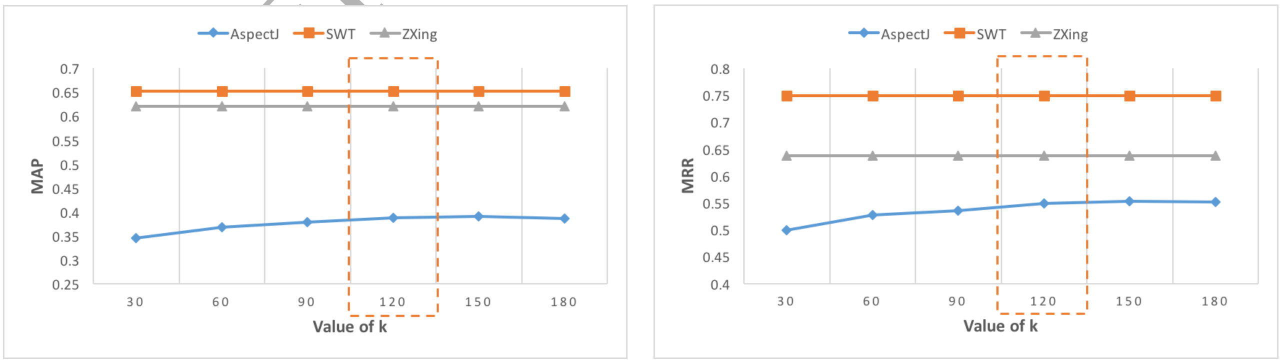
(b) MRR

Fig. 7. Impact of the value of β on BLIA in terms of MAP and MRR.



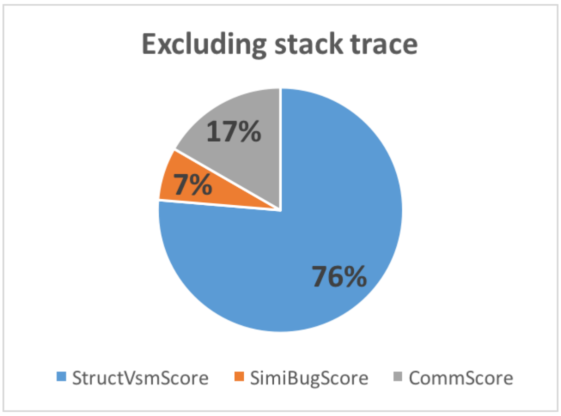
(a) AspectJ (b) SWT (c) ZXing

Fig. 8. Relation analysis between α and β on BLIA.

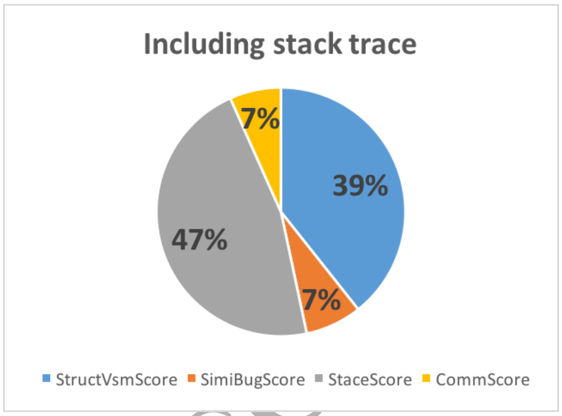


(a) MAP (b) MRR

Fig. 9. Impact of the value of k on BLIA in terms of MAP and MRR.

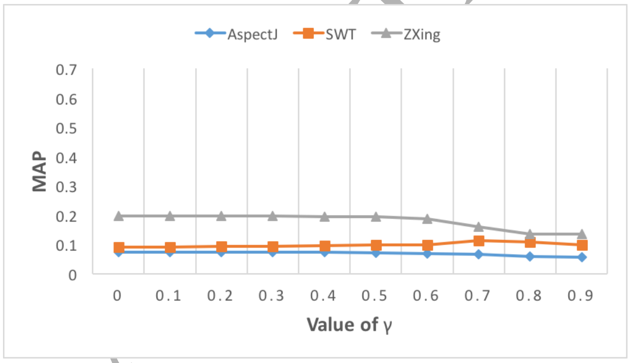


(a) Excluding stack trace

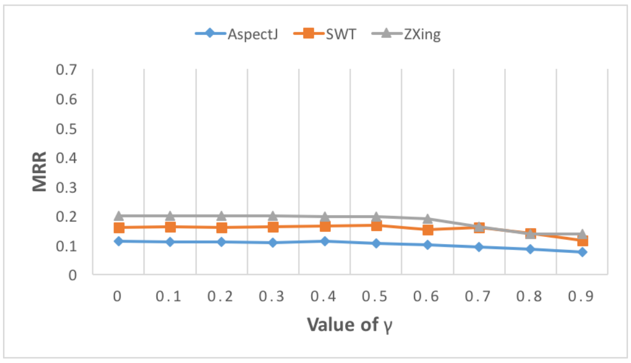


(b) Including stack trace

Fig. 10. Effects of each analyzed score.



(a) MAP



(b) MRR

Fig. 11. Impact of the value of γ on BLIA in terms of MAP and MRR.