Version History, Similar Report, and Structure: Putting Them Together for Improved Bug Localization

Shaowei Wang and David Lo
School of Information Systems
Singapore Management University, Singapore
{shaoweiwang.2010,davidlo}@smu.edu.sg

ABSTRACT

During the evolution of a software system, a large number of bug reports are submitted. Locating the source code files that need to be fixed to resolve the bugs is a challenging problem. Thus, there is a need for a technique that can automatically figure out these buggy files. A number of bug localization solutions that take in a bug report and output a ranked list of files sorted based on their likelihood to be buggy have been proposed in the literature. However, the accuracy of these tools still need to be improved.

In this paper, to address this need, we propose AmaLgam, a new method for locating relevant buggy files that puts together version history, similar reports, and structure. To do this, AmaLgam integrates a bug prediction technique used in Google which analyzes version history, with a bug localization technique named BugLocator which analyzes similar reports from bug report system, and the state-of-the-art bug localization technique BLUiR which considers structure. We perform a large-scale experiment on four open source projects, namely AspectJ, Eclipse, SWT and ZXing to localize more than 3,000 bugs. Compared with a history-aware bug localization solution of Sisman and Kak, our approach achieves a 46.1% improvement in terms of mean average precision (MAP). Compared with BugLocator, our approach achieves a 24.4% improvement in terms of MAP. Compared with BLUiR, our approach achieves a 16.4% improvement in terms of MAP.

Categories and Subject Descriptors

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

General Terms

Algorithms, Experimentation

ICPC'14, June 2–3, 2014, Hyderabad, India
ACM 978-1-4503-2879-1/14/06
http://dx.doi.org/10.1145/2597008.2597148

Keywords

Version History, Similar Report, Structure, Bug Localization

1. INTRODUCTION

Software systems are often plagued with bugs. To improve the reliability of systems, developers often allow users to submit bug reports to bug tracking systems. Unfortunately the number of these reports is often too large for the developers to handle manually in a timely manner. Anvik et al. cited a Mozilla triager that mentioned "Everyday, almost 300 bugs appear that need triaging. This is far too much for Mozilla programmers to handle" [4]. One of the most time consuming task to resolve a bug report is to find the buggy files that are responsible for a reported bug. A system may contains thousands or more files and often only one or a few of these files need to be changed to fix a bug. Lucia et al. analyze 374 bugs from Rhino, AspectJ and Lucene and find that 84-93% of the bugs reside in 1-2 source code files [19]. Thus, localizing these buggy files is like finding one or two needles in a big haystack.

To address the above mentioned challenge a number of studies have proposed ways to identify buggy program files given a bug report. Many of these approaches are information retrieval-based and they work by computing similarities between a reported bug and source code files [28, 30, 31, 35]. The source code files are then ranked based on their similarities to a reported bug. In this work, we are particularly interested in three recent approaches and we highlight each of them below.

Sisman and Kak leverage version history data for bug localization [31]. Their approach makes use of history data to compute a probability score of a file to be buggy. They also compute a similarity score between a file and a bug report. The probability and the similarity scores are then added up, and the resultant score is used to rank source code files (see [31], equation 9). They propose various variants and the best performing one is named TFIDF-DHbPd.

Zhou et al. leverage similarities among bug reports for bug localization [35]. Given a new bug report, their approach, named BugLocator, finds files that are fixed to resolve similar older bug reports. Based on the similarity of the new and older bug reports, and the number of files that are fixed in each of the older bug reports, their proposed approach computes a bugginess score (referred to as SimiScore) for each source code file. They also compute a similarity score between a file and a bug report (referred to as rVSMScore). Weighted SimiScore and rVSMScore are then added up, and the resultant score is used to rank source code files.

Table 1: Comparison of Our Approach to State-of-The-Art Bug Localization Techniques

Approach	Version History	Similar Report	Structure
TFIDF-DHbPd [31]	Yes	No	No
BugLocator [35]	No	Yes	No
BLUiR [30]	No	No	Yes
BLUiR+ [30]	No	Yes	Yes
Our Approach	Yes	Yes	Yes

To measure similarity between a bug report and a file, Saha et al. propose an approach named BLUiR which leverages the structure of a bug report and a source code file [30]. A bug report has many fields (description and summary) and a file has many parts (class names, method names, variable names, and comments). They thus transform a bug report and a file to structured documents and employ structured information retrieval. In structured information retrieval, textual contents of each field in a bug report and each part of a source code file are considered separately. Each field of a bug report is compared to each part of a source code file, and a similarity score is computed for each comparison. The summation of these scores is then used to rank source code files. Saha et al. also creates an extension of BLUiR, we refer to as BLUiR+, which integrates similar report information into BLUiR, in the same manner as BugLocator integrates SimiScore to rVSMScore.

The three approaches presented in the preceding paragraphs leverage different sources of information – see Table 1. Sisman and Kak's approach uses version history but neither related reports nor structure. Zhou et al.'s approach uses related reports but neither version history nor structure. Saha et al. approach uses both related reports and structure (i.e., for BLUiR+) but does not use version history. Thus, none of these approaches combine version history, related reports, and structure together. Based on this observation, in this work, we combine these three together, and investigate if the resultant approach works better. To do so, we combine a bug prediction technique used in Google, with the works of Zhou et al. and Saha et al. The bug prediction technique computes the probability of a file to be buggy based on historical data in a version control system. We then have a solution that puts together version history, similar report, and structure. We name our approach AmaLgam, which stands for Automated Localization of Bug using Various Information. The way AmaLgam combines historical information is different from that of Sisman and Kak in the following respects:

1. Our approach uses a well-tested bug prediction formula that is used in Google and it takes into consideration the effect of change burst [17].

2. Sisman and Kak consider the complete version history to compute a probability. Our approach only considers very recent version history and totally discards historical information that are more than k days away from the time a new bug report is submitted.

3. Sisman and Kak simply sums up the probability of a file to be buggy and the similarity of a bug report to the file. Our approach assigns weights that govern the contribution of the probability of a file to be buggy (computed by the bug prediction technique) and the similarity of a bug report to a file (computed by integrating BugLocator and BLUiR).

We have experimented our approach on a dataset of more than 3,000 bug reports from AspectJ, Eclipse, SWT, and ZXing. The AspectJ bug reports are taken from the iBugs benchmark [7] which have been used to evaluate Sisman and Kak's, Zhou et al.'s, and Saha et al's approaches. The Eclipse, SWT, and ZXing bug reports have been used before to evaluate Zhou et al.'s and Saha et al.'s approaches. The experiment results show that our approach can achieve an MAP score of 0.33, 0.35, 0.62, and 0.41 respectively. MAP score is a standard score used in information retrieval and it has been used as a yardstick to evaluate Sisman and Kak's, Zhou et al.'s, and Saha et al's approaches before. Comparing our approach's MAP scores to those of Saha et al.'s, on average, we can improve their approach's MAP score by 12.5%. Comparing our approach's MAP scores to those of Zhou et al.'s, on average, we can improve their approach's MAP score by 24.4%. Comparing our approach's MAP scores to those of Sisman and Kak's, we can improve their approach's MAP score on the AspectJ bug reports by 46.1%.

The contributions of our work are as follows:

1. We are the first to put together version history, similar reports and structure for bug localization. Past bug localization studies have only used one or two of these 3 sources of information. We have experimented our approach on more than 3,000 bug reports from 4 open source programs: AspectJ, Eclipse, SWT, and ZXing. Our experiments show that we can achieve an MAP of 0.33, 0.35, 0.62, and 0.41 for each of the 4 programs respectively.

2. The MAP scores of our proposed approach improves those of Saha et al. (i.e., BLUiR+) that put together similar reports and structure by an average of 12.5%. The MAP scores of our proposed approach improves those of BLUiR and Zhou et al. (BugLocator) by an average of 16.4% and 24.4%, respectively. Comparing with the reported results of the history-aware bug localization solution of Sisman and Kak (TFIDF-DHbPd), which was also evaluated on AspectJ bug reports from the iBugs dataset, our approach improves the MAP score by 46.1%.

The structure of the remainder of the paper is as follows. In Section 2, we first present preliminary information on bug reports and a motivating example. We elaborate the details of our approach in Section 3. We describe our experimental setup and results in Section 4. We describe related work in Section 5. We finally conclude and mention future work in Section 6.

2. PRELIMINARIES AND EXAMPLE

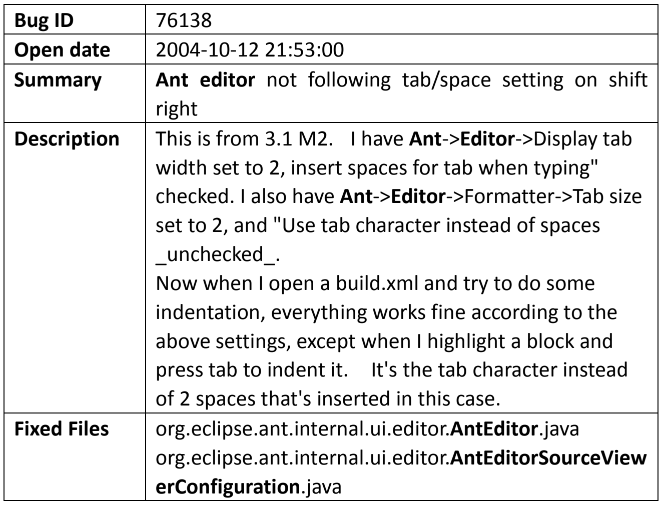
In this section, we first describe some preliminary information on bug reports. We then outline some text preprocessing steps that are applied to the bug reports. Finally, we show an example to illustrate why it is useful to consider version history, similar report, and structure.

2.1 Bug Reports

A bug report is a document submitted by users to describe an error that they experience when they use a system. A bug report contains a number of fields; we are particularly interested on 4 of them, namely bug identifier (id), the date a bug report was submitted (open date), summary of the error (summary), and more detailed description of the error (description).

We present a bug report from Eclipse in Figure 1 which can be downloaded from Eclipse's Bugzilla[1]. The identifier of this bug report is 76138 and it describes a problem with the ant editor which does not follow a display setting. The bug id provides a reference number that can be used to identify commits in version control systems that fix it, c.f. [35]. The open date helps us to identify bug reports that are submitted a number of days prior to bug 76138. The summary and description fields help us to understand the error that the user experienced.

A bug localization tool takes as input a bug report and returns the potential buggy files. The corresponding buggy Java files for the bug report shown in Figure 1, which are identified by checking the corresponding bug fixing commits, are AntEditor.java and AntEditorSourceViewerConfiguration.java.



Bug ID	76138
Open date	2004-10-12 21:53:00
Summary	**Ant editor** not following tab/space setting on shift right
Description	This is from 3.1 M2. I have **Ant**->**Editor**->Display tab width set to 2, insert spaces for tab when typing" checked. I also have **Ant**->**Editor**->Formatter->Tab size set to 2, and "Use tab character instead of spaces _unchecked_. Now when I open a build.xml and try to do some indentation, everything works fine according to the above settings, except when I highlight a block and press tab to indent it. It's the tab character instead of 2 spaces that's inserted in this case.
Fixed Files	org.eclipse.ant.internal.ui.editor.**AntEditor**.java org.eclipse.ant.internal.ui.editor.**AntEditorSourceViewerConfiguration**.java

Figure 1: An Eclipse's Bug Report and the Buggy Source Code Files Corresponding to It

2.2 Text Pre-processing

An information-retrieval based bug localization technique usually performs three pre-processing steps: text normalization, stopword removal, and stemming. The goal of the text pre-processing steps is to break a bug report or a source code file into terms that can then be analyzed by an information retrieval technique. In the pre-processing step, some compound words (e.g., program identifiers) are broken into parts, and some related words are mapped to the same term. We briefly describe these three pre-processing steps below.

First, text normalization would be performed which involves the removal of punctuation marks, tokenization (i.e., extraction of words from paragraphs or identifiers from source code), and identifier splitting. During this step, when a source code is processed, it would be converted into an Abstract Syntax Tree (AST) and using this tree, identifiers would be identified. These identifiers are split into its constituent words following Camel Case splitting [3]. For example the identifier "getMethodName" is split to "get", "Method", and "name". In this study, both the split words and the full identifier name are kept. For example, for the class name "AntEditorSourceViewerConfiguration" which is one of the buggy files corresponding to the bug report shown in Figure 1, we convert it to 6 words: "ant", "editor", "source", "viewer", "configuration" and the full identifier name "AntEditorSourceViewerConfiguration".

Second, we remove stopwords such as "on", "the", "are", "is", and so on. These stopwords carry little meaning and thus we remove them. Finally, we perform stemming which reduces inflected or derived words into a common root form. For example the word "reading" and "reads" are reduced to the root form "read". By doing this, similar words would be represented using the same term. We use the standard Porter Stemmer [24] to perform this stemming step.[2]

2.3 Motivating Example

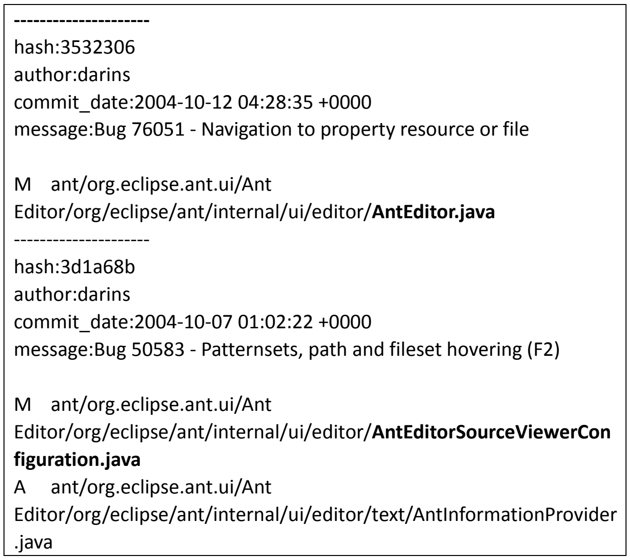
A traditional IR-based bug localization approach usually first performs text pre-precessing on a query (a bug report) and the documents in a corpus (source code files). Then, a similarity score between the query and each of the documents would be computed based on a particular information retrieval technique (e.g., TFIDF, LDA, LSI, etc), e.g., [28]. From Figure 1, we note that the buggy source code file names share a number of common words with the summary and description of the bug report, i.e., "ant" and "editor". Based on these common words a traditional IR-based bug localization approach would try to link the bug report with the source code files. In this sub-section, we highlight how version history, similar reports, and structure can be used to improve the accuracy of traditional IR-based bug localization techniques.

Version History. There are lots of historical data of changes to source code files that are stored in a version control system during program evolution. This historical data can be used to improve bug localization performance. Kim et al. found that bugs happen in bursts, and not in isolation [15]. The files responsible for a bug recently are more likely to be responsible for other bugs in the near future. Figure 2 presents the commit logs of Eclipse before bug 76138 occurred. We could see that the class files "AntEditor.java" and "AntEditorSourceViewerConfiguration.java", which were responsible for bug 76138, were also responsible for other bugs that happen prior to the reporting of bug 76138 (they are highlighted in bold). 'AntEditor.java" is fixed just one day prior to the reporting of bug 76138 and "AntEditorSourceViewerConfiguration.java" is fixed just 7 days prior to the reporting of bug 76138. Thus, we could see that historical data can be used to better locate bug.

Similar Reports. User often submit many similar bug reports that correspond to different errors that affect the

[1] `https://bugs.eclipse.org/bugs/show_bug.cgi?id=76138`

[2] `http://tartarus.org/martin/PorterStemmer/`



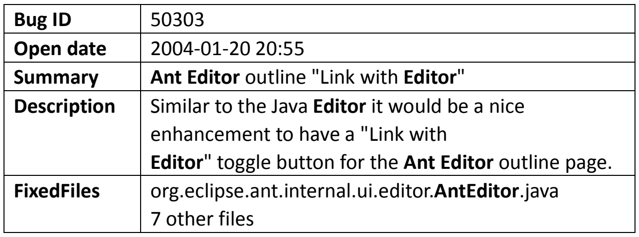
```
---------------------
hash:3532306
author:darins
commit_date:2004-10-12 04:28:35 +0000
message:Bug 76051 - Navigation to property resource or file

M    ant/org.eclipse.ant.ui/Ant
Editor/org/eclipse/ant/internal/ui/editor/AntEditor.java
---------------------
hash:3d1a68b
author:darins
commit_date:2004-10-07 01:02:22 +0000
message:Bug 50583 - Patternsets, path and fileset hovering (F2)

M    ant/org.eclipse.ant.ui/Ant
Editor/org/eclipse/ant/internal/ui/editor/AntEditorSourceViewerCon
figuration.java
A     ant/org.eclipse.ant.ui/Ant
Editor/org/eclipse/ant/internal/ui/editor/text/AntInformationProvider
.java
```

Figure 2: Recent Commit Logs Prior to the Reporting of Bug Report 76138

same buggy program elements. For example, Figure 3 shows an older report with identifier 50303[3], which were reported 9 months before bug report 76138. Note that this report shares the common words "ant" and "editor" with bug report 76138. Bug report 50303 was fixed on March 17, 2004 and was re-fixed on March 18, 2004 and "AntEditor.java" was modified on both fix instances. By analyzing bug report 50303, we can get a hint on files that need to be changed to fix 76138. From the example, we could see that similar reports can be used to better locate bug.



Bug ID	50303
Open date	2004-01-20 20:55
Summary	**Ant Editor** outline "Link with **Editor**"
Description	Similar to the Java **Editor** it would be a nice enhancement to have a "Link with **Editor**" toggle button for the **Ant Editor** outline page.
FixedFiles	org.eclipse.ant.internal.ui.editor.**AntEditor**.java 7 other files

Figure 3: An Older Eclipse's Bug Report and the Buggy Source Code Files Corresponding to It

Structure. Bug reports and source code files have structures. Bug reports have several fields including summary and description. Source code files can be split into class names, method names, variable names, and comments. This structural information can be leveraged for bug localization. Traditional IR-based bug localization approaches compute the similarity between a bug report and the entire content of a source code file (which contains a class name, many variable names, and many comments). For localizing the bug report in Figure 1, the class names contain the most important terms. Unfortunately, the impact of the terms "ant" and "editor" in the class names would be weaken by other tokens, which would make the performance poor. Structural information could be used to overcome this problem by computing the similarities of a query against different fields (e.g., class, method, variable, comment) in a source code file separately and summing up those similarities. In this way, the tokens "ant" and "editor" would have stronger impact to the overall similarity. Thus, we could see that structure can be used to better locate bug.

3. APPROACH

In this section, we first describe our overall framework named AmaLgam. We then present each of the four main components of AmaLgam.

3.1 Overall Framework of AmaLgam

Figure 4 presents the overall framework of AmaLgam. AmaLgam takes as input a bug report to be localized (new bug report), a set of source code files of the system for which the bug report is submitted (source code files), a history of commits made to the system as stored in a version control system (version history data), and a set of older bug reports stored in a bug tracking system (bug repository).

The inputs would be processed by three components of AmaLgam namely: version history component, similar report component, and structure component. Version history component makes use of version history information to rank files. Similar report component makes use of older reports in bug repository to rank files. Structure component makes use of the structure of bug reports and source code files to rank files. The three components would each outputs a suspiciousness score for each source code file. These three sets of suspiciousness scores would be input to the composer component which would produce the final ranked files.

3.2 Version History Component

For the version history component, we make use of studies on bug prediction whose goal is to predict which files are likely to be buggy in the future, e.g., [15, 27]. Kim et al. propose BugCache which predicts future bugs by maintaining a relatively short list of most fault-prone program entities [15]. Rahman et al. propose a cheaper algorithm which only sorts files based on the number of bug fixing commits that touch each of them [27]. Rahman et al. show that this simple and cheap approach achieves almost the same performance as BugCache. Google's developers adapt the simple algorithm proposed by Rahman et al to predict bugs on their large systems [17]. The resultant algorithm is simple and fast. Thus, we decide to adapt this well-tested bug prediction algorithm of Google as our version history component. We briefly describe how we adapt this algorithm in the following paragraphs.

The algorithm takes as input commit logs and outputs a list of files with their suspiciousness scores. It first identifies relevant bug-fixing commits. The relevant bug fixing commits are identified by following two rules:

1. The commit log must match the following regular expression $regex$: $(.*fix.*)|(.*bug.*)$. This regular expression specifies that all commit logs containing the word "fix" or "bug" would be matched.

2. The commit must be made in the past k days

[3] https://bugs.eclipse.org/bugs/show_bug.cgi?id=50303

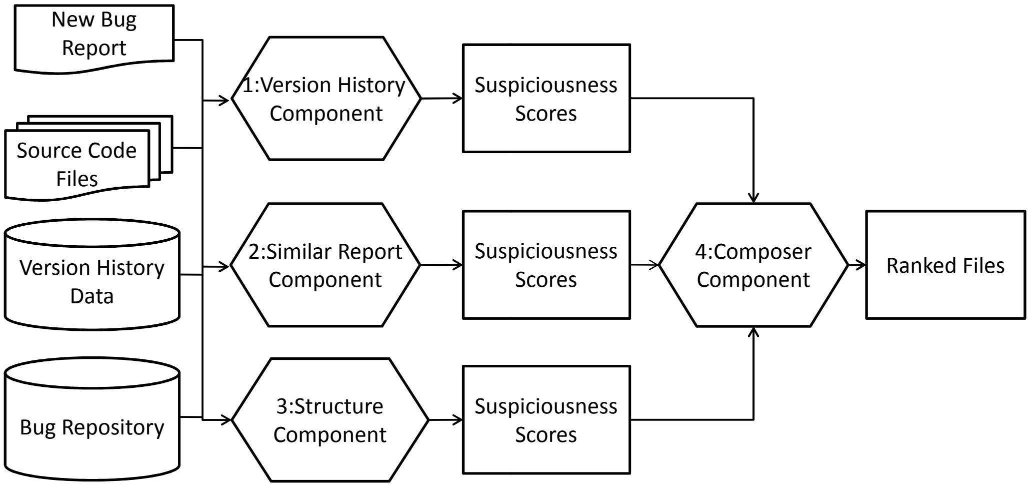


Figure 4: The Overall Framework of AmaLgam

We modify Google's approach by including the second requirement. Our experience shows that including older bug-fixing commits do not affect performance much and even can slightly decrease performance. Also, it is computationally cheaper to only consider recent commits. Parameter k could be set empirically. By default, we set k to 15. The algorithm analyzes these relevant commits and assigns a suspiciousness score to each source file f using the following equation:

$$score^{H}(f,k,R) = \sum_{c \in R \wedge f \in c} \frac{1}{1+e^{12(1-((k-t_c)/k))}} \quad (1)$$

In the above equation, R refers to the set of relevant commits and t_c is the number of days that has elapsed between a commit c and the input bug report. The output of this algorithm is a set of suspiciousness scores, one for each file. We denote the suspiciousness score of file f assigned by the version history component as $Susp^{H}(f)$.

Example: Consider the input bug report in Figure 1 and the two commit logs in Figure 2. For simplicity sake, let's assume that there are no other commit logs. We would like to illustrate how Equation 1 is used to compute the suspiciousness scores of files AntEditor.java and AntEditorSourceViewerConfiguration.java. As both commits with identifiers 3532306 and 3d1a68b contain the word "bug" and they are committed within 15 days before the time bug report 76138 was submitted (i.e., Oct 12, 2004, at 21:53:00), they are considered relevant bug fixing commits. The value of (k-t_c)/k for commit 3532306 is 0.95 (since the commit was made around 17 hours, i.e., 0.7 day, before the time bug report 76138 was submitted). Thus, the suspiciousness score for AntEditor.java is 1.82. The suspiciousness score of AntEditorSourceViewerConfiguration.java can be computed in a similar way and it is 0.009.

3.3 Similar Report Component

For our similar report component, we adapt BugLocator [35], in particular the algorithm that computes SimiRank scores. We describe briefly how we use this algorithm in the following paragraphs.

The algorithm takes in an input bug report and older bug reports that have been fixed in the bug repository. It then measures the similarity of the input bug report to the older fixed bug reports. Based on the similarity scores of the bug reports and the number of files that are modified to fix each bug report, we compute a suspiciousness score for each source code file.

To measure the similarity of two bug reports, the following steps are followed. First, each bug report is represented by their constituent pre-processed terms. Considering the universe of all terms as $\{t_1, \ldots, t_n\}$, we can compute for a bug report b, a vector $\overrightarrow{b}$:

$$\overrightarrow{b} = tf_b(t_1)idf(t_1), tf_b(t_2)idf(t_2), \ldots, tf_d(t_n)idf(t_n)$$

In the above formula, $tf_b(t_i)$ corresponds to the number of times term t_i appears in bug report b, $idf(t_i)$ corresponds to the reciprocal of the number of documents that contain term t_i. Given vector representations of two bug reports $\overrightarrow{b1}$ and $\overrightarrow{b2}$, their similarity can be measured by computing the standard cosine similarity [5] of their vector representations.

To compute a suspiciousness score for source code file f, we use the following equation:

$$score_R(f,b,B) = \sum_{b' \in \{b' | b' \in B \wedge f \in b'.Fix\}} \frac{sim(b,b')}{|b'.Fix|} \quad (2)$$

In the above equation, b is the input bug report, B is the set of older fixed bug reports, $sim(b,b')$ is the similarity of bug report b and b', $b'.Fix$ is the set of files that are modified to fix bug report b', and $|b'.Fix|$ is the size of set $b'.Fix$. The output of this algorithm is a set of suspiciousness scores, one for each file. In this component, we do not enforce a similarity threshold following what Zhou et al. did in their work [35]. We take all older bug reports to compute the suspiciousness score. We denote the suspiciousness score of file f assigned by the similar report component as $Susp^{R}(f)$.

Example: Consider the input bug report shown in Figure 1 and the older bug report in Figure 3. For simplicity sake, let us assume that there are no other bug reports in the bug repository. We would like to illustrate how Equation 3.3 is used to compute the suspiciousness scores of files AntEditor.java. Let us assume for simplicity sake that the similarity of the two bug reports is 0.15. The suspiciousness score of AntEditor.java can then be computed as: 0.15*8 = 1.2.

3.4 Structure Component

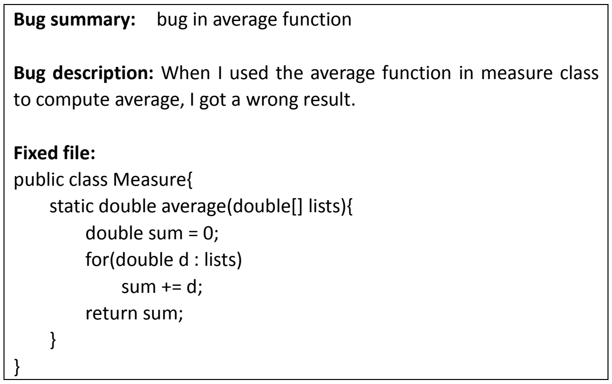
For the structure component we use BLUiR [30] which performs structured retrieval for bug localization. For completeness-sake, we briefly describe BLUiR in the following paragraphs.

BLUiR breaks a bug report into 2 parts: summary and description. It breaks a source code file into 4 parts: class names, method names, variable names, and comments. Each of these parts can be converted into a vector following a similar procedure described in Section 3.3. The suspiciousness score of a source code file f given an input bug report b can then be computed as:

$$score_S(f,b) = \sum_{fp \in f} \sum_{bp \in b} sim(fp, bp)$$

where fp is a part of file f, bp is a field in bug report b, and $sim(fp, bp)$ is the cosine similarity of the vector representations of fp and bp. The output of the structure component is a set of suspiciousness scores, one for each file. We denote the suspiciousness score of file f assigned by the structure component as $Susp^S(f)$.

Example: Consider a bug report and a file shown in Figure 5. After pre-processing, the terms in the summary field of the bug report are: "bug", "average", and "function". The terms in the description field of the bug report are: "us", "average", "function", "measure", "class", "comput", "got", "wrong", and "result". The term in the class name of the file is "measure". The term in the method name of the file is "average". The terms in the variable names of the file are "list", "sum", and "d". The set of terms in the comments of the file is Ø. Based on these fields of the bug report and these parts of the source code file, we can compute a suspiciousness score which would be a summation of 8 similarity scores.



Bug summary: bug in average function

Bug description: When I used the average function in measure class to compute average, I got a wrong result.

Fixed file:

```
public class Measure{
    static double average(double[] lists){
        double sum = 0;
        for(double d : lists)
            sum += d;
        return sum;
    }
}
```

Figure 5: Example Bug Report and Source Code File

3.5 Composer Component

This component processes the 3 sets of suspiciousness scores output by the first 3 components of AmaLgam and computes a set of final suspiciousness scores. The composer component first combines the scores output by the structure component and the similar report component for a file f, as follows:

$$Susp^{S,R}(f) = (1-a) \times Susp^S(f) + a \times Susp^R(f)$$

The value of a can be empirically determined. Saha et al. has also tried to combine standard BLUiR with similar report data and they set the value of a to be 0.2. We use the same value of a. The impact of various a on the effectiveness of incorporating similar report information has been shown in [35]. Based on $Susp^{SR}$ we compute the final suspiciousness score of f as follows:

$$Susp^{S,R,H}(f) =$$

$$\begin{array}{rr} (1-b) \times Susp^{S,R}(f) + b \times Susp^H(f), & if Susp^{S,R}(f) > 0 \\ 0, & otherwise \end{array}$$

In the above equation, we set the final suspiciousness score to 0, if $Susp^{S,R}(f)$ is 0. We do this since we observe that the likelihood of a file to be relevant to a bug report is very small if $Susp^{S,R}(f)$ is 0. The value of b can be empirically determined. By default, we set this value to 0.3.

In the end, the composer component would sort all source code files based on their final suspiciousness scores and this ranked list of files would be the output of AmaLgam.

Example: Suppose, $Susp^{S,R}$ scores of 3 files are $\{f_1 = 0.2, f_2 = 0.1, f_3 = 0\ \}$ and the $Susp^H$ scores of the 3 files are $\{f_1 = 0.1, f_2 = 0.5, f_3 = 0.9\ \}$. Since the $Susp^{S,R}$ score of f_3 is 0, then the final suspiciousness score of f_3 would still be 0 even though its $Susp^H$ is large. Thus, the final suspiciousness scores are $f_1 = 0.17$, $f_2 = 0.22$, $f_3 = 0$.

4. EXPERIMENTS

In this section, we first describe the dataset that we use to evaluate our approach. Next, we describe our evaluation metrics, followed by our research questions. Finally, we describe our experiment results which answer the research questions.

4.1 Dataset

We use the same dataset used by Zhou et.al and Saha et.al to evaluate BugLocator and BLUiR respectively [30, 35]. This dataset contains a total of 3,379 bug reports from four popular open source projects, AspectJ, Eclipse, SWT, and ZXing. For each bug report, information on files that were modified to fix the bug is also provided in the dataset. The AspectJ bug reports originate from the iBugs benchmark [7] which was also used by Sisman and Kak to evaluate their proposed approach [31]. Table 2 describes the dataset in more detail. For our version history component, we collect commit logs from Git repositories of those four projects.

4.2 Evaluation Metrics

To measure the effectiveness of the proposed bug localization approach, we use the following metrics:

- **Top-N Rank (Hit@N)**: This metric calculates the number of bug reports where one of the buggy files appears in the top N (i.e., 1, 5, 10) ranked files. Given a bug report, if at least one of its buggy files is in the top N results, we consider the bug is successfully

Table 2: Dataset Details

Project	Description	Period	#Fixed Bugs	#Source Files
AspectJ	Aspect-oriented extension of Java	07/2002-10/2010	286	6485
Eclipse	Open source IDE	10/2004-03/2011	3075	12863
SWT	Open source widget toolkit	10/2004-04/2010	98	484
ZXing	Barcode image processing library for Android platform	03/2010-09/2010	20	391

located. The higher the value of this metric is, the better the performance of an approach is.

- **Mean Average Precision (MAP)**: MAP is the most commonly used IR metric to evaluate ranking approaches. It considers the ranks of all buggy files into consideration. Therefore, MAP emphasizes all of the buggy files instead of only the first one. MAP is computed by taking the mean of the *average precision* scores across all queries. The average precision of a single query is computed as:

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

  where k is a rank in the returned ranked files, M is the number of ranked files and $pos(k)$ indicates whether the k^{th} file is a buggy file or not. $P(k)$ is the precision at a given top k files and is computed as follows:

$$P(k) = \frac{\#buggy\ files}{k}.$$

- **Mean Reciprocal Rank (MRR)**: The reciprocal rank for a query is the reciprocal of the position of the first buggy file in the returned ranked files. MRR is the mean of the reciprocal ranks over a set of queries Q and it can be computed by following equation:

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

  where $rank_i$ is the position of the first buggy file in the returned ranked files for the first query in Q.

4.3 Research Questions

Research Question 1 How effectiveness is AmaLgam for bug localization?

To answer this research question, we apply AmaLgam to the 4 sets of bug reports in our dataset. We then evaluate the returned ranked lists and compute Hit@N, MAP, and MRR to characterize the effectiveness of AmaLgam.

Research Question 2 Does AmaLgam outperform other bug localization techniques?

AmaLgam combines 3 state-of-the-art approaches: TFIDF-DHbPd by Sisman and Kak [31], BugLocator by Zhou et al. [35], and BLUiR by Saha et al. [30]. TFIDF-DHbPd is the best performing variant of the approach proposed by Sisman and Kak. Saha et al. also tried to compose similar report information in the same way as Zhou et al. We refer to this variant of BLUiR as BLUiR+. We would like to investigate whether and to what extent AmaLgam outperforms these existing state-of-the-art approaches. We compare the results of AmaLgam with the results reported in past papers [30, 31, 35].

Research Question 3 How does the performance of AmaLgam vary for various settings of parameters k and b?

AmaLgam accepts 3 parameters: k, a, and b. The setting of parameter a follows the setting used by Saha et al. [30] which actually follows the setting used by Zhou et al. [35]. In this research question, we would like to investigate the effect of varying the other two parameters of k and b. k is a parameter of the version history component which determines the number of days for which we cache the history. b is a parameter of the composer component that determines the contribution of the history-based suspiciousness score ($Susp^H$).

4.4 Experiment Results

The following subsections describe our experimental results which answer the 3 research questions. We answer one research question at a time.

4.4.1 *RQ1: Effectiveness of AmaLgam*

To answer the RQ1, we measure the effectiveness of AmaLgam in terms of the metrics we listed in Section 4.2. Table 3 presents the results for all programs. For 127 (44.4%) AspectJ bug reports, AmaLgam successfully locates a buggy source code file in the top 1 ranked files. For 187 (65.4%) AspectJ bugs, at least one buggy source code file is among the top 5 ranked files. For 209 (73.1%) AspectJ bugs, at least one buggy source code file is among the top 10 ranked files. In terms of MAP and MRR, AmaLgam achieves a score of 0.33 and 0.45, respectively.

For Eclipse, 1060 (34.5%) and 1775 (57.7%) bugs could be localized by inspecting top 1 and 5 ranked files, respectively. Also, 2059 (67.0%) bugs could be localized when only the top 10 ranked files are inspected. The scores of MAP and MRR that AmaLgam gets for Eclipse are 0.35 and 0.45, respectively. For SWT, 61 (62.2%) bugs have a buggy file at the top 1 ranked file. Also, 80 (81.6%) and 88 (89.8%) bugs are successfully localized when only the top 5 and 10 ranked files are inspected, respectively. AmaLgam gets MAP and MRR scores of 0.62 and 0.71, respectively. For ZXing, AmaLgam is able to localize 8 (40.0%), 13 (65.0%), and 14 (70%) bugs when only the top 1, 5 and 10 ranked files are inspected, respectively. In terms of MAP and MRR, AmaLgam achieves a score of 0.41 and 0.51, respectively.

4.4.2 *RQ2: AmaLgam VS. Other Bug Localization Approaches*

Table 3 compares the results of AmaLgam with those of TFIDF-DHbPd, BugLocator, and two versions of BLUiR (BLUiR and BLUiR+) in terms of Hit@1, Hit@5, Hit@10, MAP and MRR. For TFIDF-DHbPd, Sisman and Kak only evaluates it using AspectJ bug reports from the iBugs benchmark. They also did not compute Hit@N or MRR. Thus, in the table, we only show the MAP score of TFIDF-DHbPd for AspectJ. Comparing our approach with TFIDF-

Table 3: Comparison among AmaLgam, TFIDF-DHbPd, BugLocator, BLUiR, and BLUiR+

Project	Approach	Hit@1	Hit@5	Hit@10	MAP	MRR
AspectJ	AmaLgam	**127 (44.4%)**	**187 (65.4%)**	**209 (73.1%)**	**0.33**	**0.54**
	TFIDF-DHbPd	N/A	N/A	N/A	0.23	N/A
	BugLocator	88 (30.8%)	146 (50.1%)	170 (59.4%)	0.22	0.41
	BLUiR	92 (32.2%)	146 (51.0%)	173 (60.5%)	0.24	0.41
	BLUiR+	97 (33.9%)	150 (52.4%)	176 (61.5%)	0.25	0.43
Eclipse	AmaLgam	**1060 (34.5%)**	**1775 (57.7%)**	**2059 (67.0%)**	**0.35**	**0.45**
	BugLocator	896 (29.1%)	1653 (53.8%)	1925 (62.6%)	0.30	0.41
	BLUiR	952 (31.0%)	1636 (53.2%)	1933 (62.9%)	0.32	0.42
	BLUiR+	1013 (32.9%)	1729 (56.2%)	2010 (65.4%)	0.33	0.44
SWT	AmaLgam	**61 (62.2%)**	**80 (81.6%)**	**88 (89.8%)**	**0.62**	**0.71**
	BugLocator	39 (39.8%)	66 (67.3%)	81 (62.6%)	0.45	0.53
	BLUiR	54 (55.1%)	75 (76.5%)	85 (86.7%)	0.56	0.65
	BLUiR+	55 (56.1%)	75 (76.5%)	86 (87.8%)	0.58	0.66
ZXing	AmaLgam	**8 (40.0%)**	**13 (65.0%)**	**14 (70.0%)**	0.41	**0.51**
	BugLocator	8 (40.0%)	12 (60.0%)	14 (70.0%)	**0.44**	0.50
	BLUiR	8 (40.0%)	13 (65.0%)	14 (70.0%)	0.38	0.49
	BLUiR+	8 (40.0%)	13 (65.0%)	14 (70.0%)	0.39	0.49

DHbPd, we can improve their approach's MAP score on AspectJ bug reports by 46.1%.

Comparing our approach with BugLocator, AmaLgam outperforms BugLocator with respect to all metrics for AspectJ, Eclipse, and SWT bug reports. Both techniques have the same performance in terms of Hit@1, Hit@5, and Hit@10 for ZXing. For ZXing, AmaLgam improves BugLocator in terms of MRR, bug marginally loses to BugLocator in terms of MAP. On average, AmaLgam improves the MAP and MRR scores of BugLocator by 24.4% and 19.3%, respectively.

Comparing AmaLgam with BLUiR and BLUiR+, we could note that AmaLgam consistently outperforms BLUiR and BLUiR+ in terms of MAP and MRR for all programs. The Hit@N scores of AmaLgam are better than those of BLUiR and BLUiR+ for all programs except ZXing. For ZXing, the Hit@N scores of AmaLgam are the same with those of BLUiR and BLUiR+. On average, AmaLgam improves the MAP and MRR scores of BLUiR+, which performs better than BLUiR, by 12.5% and 9.9% respectively. We perform Wilcoxon signed-rank test [34] to test whether the improvements obtained by AmaLgam over BLUiR+ are significant. We found that the improvements in terms of MAP and MRR are significant.

4.4.3 RQ3: Effect of Varying k and b

In this study, we select $k \in \{5,10,15,20,25,30,35,40,45,50\}$ and compute MAP and MRR for all bug reports in our dataset. The results are presented in Figures 6 and 7.

When k increases from 0 to 15, the MAP and MRR scores increase for all programs. Increasing the value of k further from 20 to 50 generally does not improve performance much. For SWT, the MAP and MRR slightly decrease when we increase k from 15 to 50. These results show that there is no need to consider old history. The most important part of the history is commits in the last 15-20 days.

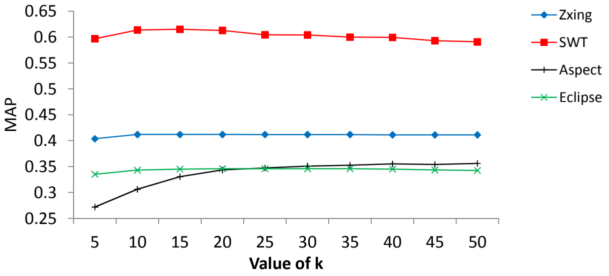


Figure 6: Impact of Varying the Value of k on AmaLgam in Terms of MAP

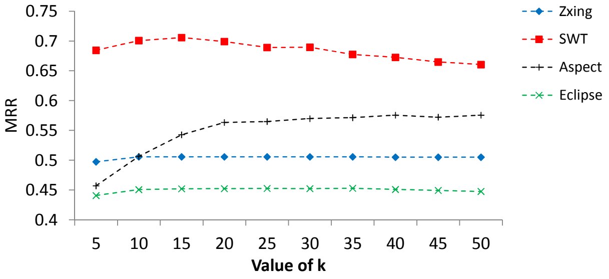


Figure 7: Impact of Varying the Value of k on AmaLgam in terms of MRR

Next, we investigate the impact of different values of b on the performance of AmaLgam. We vary the values of b from 0 to 1 with an interval of 0.1. Figures 8 and 9 show how the performance of AmaLgam varies due to different b values, in terms of MAP and MRR. For ZXing, Eclipse and SWT, both MAP and MRR slightly increase when the value of b is varied from 0 to 0.3; when the value of b is varied from 0.3 to 1, in general the performance goes down.[4] The impact of varying b values for AspectJ is different from the other programs: MAP value keeps increasing when the value of b is varied from 0 to 0.7, remains relatively stable when b is varied from 0.7 to 0.9, and decreases when b is increased from 0.9 to 1; In terms of MRR, the performance improves when b is varied from 0 to 0.5, remains stable when b is varied from 0.5 to 0.7, and decreases when b is varied from 0.7

[4] The performance slightly increases for ZXing when b is varied from 0.4 to 0.9.

to 1.0. For three of the four programs, AmaLgam achieves the best performance when b is 0.3. Averaging across the 4 programs, the performance of AmaLgam remains relatively stable when we vary b between 0.3-0.4.

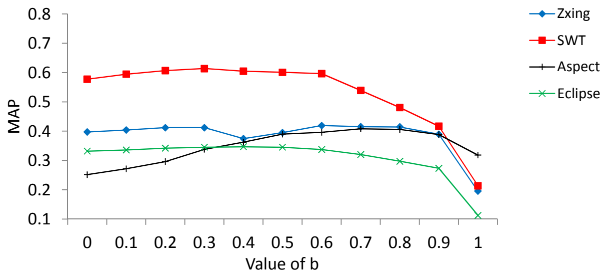


Figure 8: Impact of Varying the Value of b on AmaLgam in terms of MAP

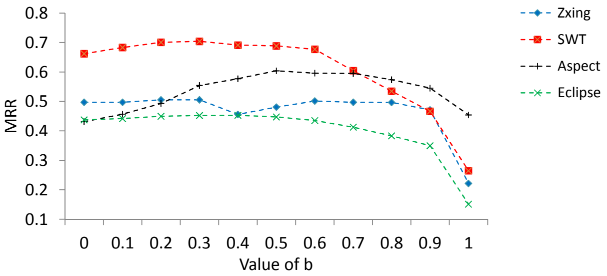


Figure 9: Impact of Varying the Value of b on AmaLgam in terms of MRR

4.5 Threats to Validity

Threats to internal validity includes experimenter bias. To reduce this threat, we reuse the bug reports dataset that has been used before to evaluate prior approaches. Thus, the evaluation is not biased to our approach.

Threats to external validity relates to the generalizability of our findings. To reduce this threat, we have analyzed more than 3,000 bug reports from 4 popular projects. Still in the future, we plan to reduce these threats further by analyzing more bug reports from more projects written in multiple programming languages.

Threats to construct validity refers to the suitability of the set of evaluation metrics that we use in this study. Three metrics are used namely Hit@N, MAP, and MRR. These metrics are well-known information retrieval metrics and have been used before to evaluate many past bug localization approaches, e.g., [28, 30, 31, 35]. Thus, we believe there is little threat to construct validity.

5. RELATED WORK

In this section, we first describe a number of bug localization works. We then describe some bug prediction and feature location works. The survey here is by no means complete.

5.1 Bug Localization

In recent years, many bug localization approaches have been proposed. These methods can be categorized into two: dynamic and static approaches.

Generally, dynamic approaches can localize a bug much more precisely than static approaches, e.g., pinpoint a buggy statement or basic block. However, they usually require a test suite to execute a program to collect passing and failure execution traces. Thus the effectiveness of a dynamic approach is often dependent on the quality of a test suite. Unfortunately, Kocchar et al. have shown that the adoption of software testing in many projects is often poor [16]. Spectrum-based fault localization, e.g., [2, 14, 18, 29] and model-based fault localization, e.g., [10, 22], are some of the well known dynamic approaches. Spectrum-based fault localization approaches often use program traces to correlate program elements at various granularity levels (e.g., statements, basic blocks, functions, and components) with program failures often with the help of a statistical analysis. Tarantula [14] and Ochiai [2] are two well known techniques and they are proposed to rank program elements according to their suspiciousness scores computed based on the executions of a program with a test suite. The basic idea of Tarantula and Ochiai is that a program element is considered to be more suspicious if it appears more frequently in failed executions than in correct ones. Saha et al. propose a customized automated fault localization technique for data-centric programs which interact with databases [29]. Model-based fault localization approaches, e.g., [10,22] are based on more expensive logic reasoning over formal models of programs which are often more accurate than spectrum-based fault localization approaches.

Static approaches do not require any test suite to be run to generate execution traces. They only need program source code files and bug reports to localize a bug. The static approaches usually can be categorized into two groups: program analysis based approaches and IR-based approaches. FindBugs is a program analysis based approach that locates a bug based on some predefined bug patterns [13]. However, it often detects too many false positives and misses many real bugs [32]. IR-based approaches use information retrieval techniques (such as, TFIDF, LSA, LDA, etc.) to calculate the similarity between a bug report and a source code file. Rao and Kak investigates many standard information retrieval techniques for bug localization and find that simpler techniques, e.g., TFIDF and SUM, perform the best [28]. Lukins et al. use Latent Dirichlet Allocation (LDA), which is a well-known topic modeling approach, to localize bug [20]. Sisman and Kak propose a history-aware IR-based bug localization solution to achieve a better result [31]. Zhou et al. propose BugLocator which leverages similarities among bug reports and uses a refined vector space model (rVSM) to perform bug localization [35]. Saha et al. consider the structure of bug reports and source code files and employ structured retrieval to achieve a better result [30]. Different from the existing IR-based bug localization approaches, we put together version history, similar report, and structure, to achieve better performance.

5.2 Bug Prediction

There are many approaches proposed for bug prediction. One family of bug prediction approaches uses change logs to predict buggy files. Change log-based approaches extract historical information from a version control system and assume that recently or frequently changed files have the most potential to be buggy. Hassan measures the complexity of a code change and proposes several code change models which are based on the concept of entropy and shows that the code change models can be used to predict future faults [12]. Kim et al propose BugCache which stores a list of recent buggy files in a cache and use it to predict future buggy files [15]. BugCache is based on an assumption that similar bugs happen in bursts and not in isolation. Rahman et al. perform an empirical study to evaluate BugCache and shows that it is not substantially better than a basic prediction model which computes the suspiciousness of a file based on the number of bug-fixing commits that touch the file [27].

Another family of approaches do not require historical data but only analyzes the current version of a system using various metrics. One well-known set of metrics is the Chidamber and Kemerer (CK) metrics [6]. These metrics and several coupling metrics have been used by El Emam et al. to predict faults on commercial Java application [1]. Nagappan et al. use a number of source code metrics (including CK metrics) to predict module-level defects on five Microsoft systems [23]. They find that no predictor could perform well on all the projects. Marcus et al. propose the notion of conceptual cohesion of classes (C3) which is based on the analysis of unstructured text (e.g., comments and identifiers) in a code base, and use C3 for defect prediction [21].

There are also other approaches that do not belong to the two families described above. For example, Zimmermann and Nagappan use network analysis to analyze the dependencies between binaries in Windows server 2003 and predict defects based on that analysis [36].

5.3 Feature/Concept/Concern Location

Feature/concept/concern location is a task that is closely related to bug localization. Its goal is to map a description of a feature or concept or concern to the program units (e.g., package, file, method) that implement it. Many approaches have been proposed to perform feature/concept/concern localization with information retrieval techniques.

Poshyvanyk et al. make use of Latent Semantic Indexing (LSI) to map a software feature to its relevant program units, and then apply Formal Concept Analysis (FCA) to cluster the results [25]. In another work, they also make use of execution traces in addition to textual description of a feature to locate relevant program units [26]. Dit et al. combine information retrieval, execution and link analysis algorithms to improve feature location techniques that analyze textual description and execution traces by using data fusion model [9]. Gethers et al. combine information retrieval (IR), dynamic analysis, and software repository mining techniques to recommend relevant source code entities given a change request and its contextual information, i.e., execution trace and initial source code entity to be changed [11]. Wang et al. perform an empirical study on Linux kernel to evaluate the performance of 10 different IR models for feature location [33]. They shows that vector space model (VSM) outperforms other models. There are many other feature location approaches. For a comprehensive description of these studies, please refer to a recent survey paper by Dit et al. [8].

6. CONCLUSION AND FUTURE WORK

A large number of bug reports are submitted during the evolution of a software system. For a large system, locating the source code files responsible for a bug is a tedious and expensive work. Thus, there is a need to develop a technique that can automatically figure out these buggy files given a bug report. A number of bug localization tools have been proposed in recent years. However, the accuracy of these tools still need to be improved. In this paper, we propose AmaLgam, a new method for locating relevant buggy files that combines historical data, similar report, and structural information to achieve a higher accuracy. We perform a large-scale experiments on four projects, namely AspectJ, Eclipse, SWT and ZXing to localize more than 3,000 bugs. Compared with the state-of-the-art bug localization approaches, BLUiR+ and BLUiR, our approach, on average, achieves 12.5% and 16.4% improvements in terms of mean average precision (MAP). Compared with BugLocator which considers similar reports, our approach, on average, achieves a 24.4% improvement in terms of MAP. Compared with a history-aware bug localization approach proposed by Sisman and Kak, our approach achieves a 46.1% improvement in terms of MAP.

In the future, we would like to reduce the threats to external validity further by applying our approach on more bug reports from various systems. We are also interested to integrate other bug prediction approaches to AmaLgam. Furthermore, we want to investigate different ways to combine the scores from the three components of AmaLgam. We are also interested to use Principal Component Analysis (PCA) to analyze which component (i.e., version history, similar report, or structure) contributes the most to the final results.

7. ACKNOWLEDGEMENT

We would like to thank Zhou et al. and Saha et al. for making their results and datasets available for public download.

8. REFERENCES

[1] The prediction of faulty classes using object-oriented design metrics. *Journal of Systems and Software*, 56(1):63 – 75, 2001.

[2] R. Abreu, P. Zoeteweij, R. Golsteijn, and A. J. C. van Gemund. A practical evaluation of spectrum-based fault localization. *J. Syst. Softw.*, 82(11).

[3] G. Antoniol, G. Canfora, G. Casazza, A. De Lucia, and E. Merlo. Recovering traceability links between code and documentation. *IEEE Trans. Softw. Eng.*, 28(10), Oct. 2002.

[4] J. Anvik, L. Hiew, and G. C. Murphy. Coping with an open bug repository. In *ETX*, pages 35–39, 2005.

[5] R. Baeza-Yates, B. Ribeiro-Neto, et al. *Modern information retrieval*, volume 463. ACM press New York, 1999.

[6] S. R. Chidamber and C. F. Kemerer. A metrics suite for object oriented design. *IEEE Trans. Softw. Eng.*, 20(6), June 1994.

[7] V. Dallmeier and T. Zimmermann. Extraction of bug localization benchmarks from history. In *ASE*, pages 433–436, 2007.

[8] B. Dit, M. Revelle, M. Gethers, and D. Poshyvanyk. Feature location in source code: a taxonomy and survey. *Journal of Software: Evolution and Process*, 25(1):53–95, 2013.

[9] B. Dit, M. Revelle, and D. Poshyvanyk. Integrating information retrieval, execution and link analysis algorithms to improve feature location in software. *Empirical Software Engineering*, 18(2):277–309, 2013.

[10] A. Feldman and A. van Gemund. A two-step hierarchical algorithm for model-based diagnosis. In *Proceedings of the 21st National Conference on Artificial Intelligence - Volume 1*, AAAI'06, pages 827–833, 2006.

[11] M. Gethers, B. Dit, H. H. Kagdi, and D. Poshyvanyk. Integrated impact analysis for managing software changes. In *ICSE*, pages 430–440, 2012.

[12] A. Hassan. Predicting faults using the complexity of code changes. In *Proceedings of the 31st International Conference on Software Engineering*, ICSE '09, pages 78–88, 2009.

[13] D. Hovemeyer and W. Pugh. Finding bugs is easy. *SIGPLAN Not.*, 39(12).

[14] J. A. Jones and 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*, ASE '05, 2005.

[15] S. Kim, T. Zimmermann, E. J. Whitehead Jr., and A. Zeller. Predicting faults from cached history. In *Proceedings of the 29th International Conference on Software Engineering*, ICSE '07, pages 489–498, 2007.

[16] P. S. Kochhar, T. F. Bissyandé, D. Lo, and L. Jiang. An empirical study of adoption of software testing in open source projects. In *QSIC*, pages 103–112, 2013.

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

[18] Lucia, D. Lo, L. Jiang, and A. Budi. Comprehensive evaluation of association measures for fault localization. In *ICSM*, pages 1–10, 2010.

[19] Lucia, F. Thung, D. Lo, and L. Jiang. Are faults localizable? In *MSR*, pages 74–77, 2012.

[20] S. K. Lukins, N. A. Kraft, and L. H. Etzkorn. Bug localization using latent dirichlet allocation. *Information & Software Technology*, 52(9):972–990, 2010.

[21] A. Marcus, D. Poshyvanyk, and R. Ferenc. Using the conceptual cohesion of classes for fault prediction in object-oriented systems. *IEEE Trans. Softw. Eng.*, 34(2), Mar. 2008.

[22] W. Mayer and M. Stumptner. Model-based debugging – state of the art and future challenges. *Electron. Notes Theor. Comput. Sci.*, 174(4), 2007.

[23] N. Nagappan, T. Ball, and A. Zeller. Mining metrics to predict component failures. In *Proceedings of the 28th International Conference on Software Engineering*, ICSE '06, 2006.

[24] M. Porter. An algorithm for suffix stripping. *Program*, 14(3):130–137.

[25] D. Poshyvanyk, M. Gethers, and A. Marcus. Concept location using formal concept analysis and information retrieval. *ACM Trans. Softw. Eng. Methodol.*, 21(4), Feb. 2013.

[26] D. Poshyvanyk, Y.-G. Guéhéneuc, A. Marcus, G. Antoniol, and V. Rajlich. Feature location using probabilistic ranking of methods based on execution scenarios and information retrieval. *IEEE Trans. Software Eng.*, 33(6), 2007.

[27] F. Rahman, D. Posnett, A. Hindle, E. Barr, and 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*, ESEC/FSE '11, pages 322–331, New York, NY, USA, 2011. ACM.

[28] S. Rao and A. C. Kak. Retrieval from software libraries for bug localization: a comparative study of generic and composite text models. In *MSR*, pages 43–52, 2011.

[29] D. Saha, M. G. Nanda, P. Dhoolia, V. K. Nandivada, V. Sinha, and S. Chandra. Fault localization for data-centric programs. In *Proceedings of the 19th ACM SIGSOFT Symposium and the 13th European Conference on Foundations of Software Engineering*, ESEC/FSE '11, 2011.

[30] R. K. Saha, M. Lease, S. Khurshid, and D. E. Perry. Improving bug localization using structured information retrieval. In *ASE*, pages 345–355, 2013.

[31] B. Sisman and A. C. Kak. Incorporating version histories in information retrieval based bug localization. In *MSR*, pages 50–59, 2012.

[32] F. Thung, Lucia, D. Lo, L. Jiang, F. Rahman, and P. T. Devanbu. To what extent could we detect field defects? an empirical study of false negatives in static bug finding tools. In *ASE*, pages 50–59, 2012.

[33] S. Wang, D. Lo, Z. Xing, and L. Jiang. Concern localization using information retrieval: An empirical study on linux kernel. In *WCRE*, 2011.

[34] F. Wilcoxon. Individual comparisons by ranking methods. *Biometrics Bulletin*, 6(1):80âĂŞ83, 1945.

[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 *ICSE*, pages 14–24, 2012.

[36] T. Zimmermann and N. Nagappan. Predicting defects using network analysis on dependency graphs. In *Proceedings of the 30th International Conference on Software Engineering*, ICSE '08, pages 531–540, 2008.