BugLocalizer: Integrated Tool Support for Bug Localization

Ferdian Thung, Tien-Duy B. Le, Pavneet Singh Kochhar, and David Lo
School of Information Systems
Singapore Management University
{ferdiant.2013,btdle.2012,kochharps.2012,davidlo}@smu.edu.sg

ABSTRACT

To manage bugs that appear in a software, developers often make use of a bug tracking system such as Bugzilla. Users can report bugs that they encounter in such a system. Whenever a user reports a new bug report, developers need to read the summary and description of the bug report and manually locate the buggy files based on this information. This manual process is often time consuming and tedious. Thus, a number of past studies have proposed bug localization techniques to automatically recover potentially buggy files from bug reports.

Unfortunately, none of these techniques are integrated to bug tracking systems and thus it hinders their adoption by practitioners. To help disseminate research in bug localization to practitioners, we develop a tool named BugLocalizer, which is implemented as a Bugzilla extension and builds upon a recently proposed bug localization technique. Our tool extracts texts from summary and description fields of a bug report and source code files. It then computes similarities of the bug report with source code files to find the buggy files. Developers can use our tool online from a Bugzilla web interface by providing a link to a git source code repository and specifying the version of the repository to be analyzed. We have released our tool publicly in GitHub, which is available at: `https://github.com/smagsmu/buglocalizer`. We have also provided a demo video, which can be accessed at: `http://youtu.be/iWHaLNCUjBY`.

Categories and Subject Descriptors: D.2.7 [Software Engineering]: Distribution, Maintenance, and Enhancement
Keywords: Bug localization, Bugzilla, git.

1. INTRODUCTION

Bug tracking systems like Bugzilla are used by a large number of developers and organisations to manage bugs affecting their projects. The number of bug reports can overwhelm developers working on the project. A Mozilla developer reported that "Everyday, almost 300 bugs appear that need triaging. This is far too much for only the Mozilla programmers to handle" [1]. Furthermore, a software system may contain thousands of source code files and often a bug only affects one or a few files. Recently, Lucia et al. reported that 84-93% of bugs only affect 1-2 source code files [4]. It is not easy to identify these few files from the thousands of source code files. These highlight that finding buggy files for bug reports in a bug tracking system is an arduous task.

To overcome this issue, researchers have proposed automated bug localization techniques, which take as input bug reports and use textual information from the summary and description fields of these reports to find the buggy source code files. These techniques extract words from a bug report and words from identifiers present in source code files; it then compare these words to find source code files that are the most similar to the bug report using an information retrieval (IR) approach. One of the recently proposed techniques is BugLocator [14] which is invented by Zhou et al.; It uses a revised Vector Space Model (rVSM) that takes into account the fact that larger files tend to be more buggy, and uses information gleaned from past fixed similar bug reports based on the hypothesis that textually similar bug reports are likely to share similar buggy files. Zhou et al. have demonstrated the effectiveness of BugLocator and its performance is better than many other bug localization techniques based on VSM, LSI, SUM and LDA.

Many bug localization techniques including BugLocator have been evaluated on a large number of bug reports and shown to be effective. However, at the moment, they are not widely used by practitioners. One barrier towards the adoption of these techniques is the fact that they are not integrated to bug tracking systems like Bugzilla. Thus practitioners are less likely to use them as they need to manually download bug reports from Bugzilla, read the manual of these techniques (if their implementations are publicly available), configure them, and run them outside the practitioners' existing bug management environment (i.e., Bugzilla).

To address the above issue and make it easier for developers to adopt automated bug localization techniques, we develop a tool named BugLocalizer, which is implemented as a Bugzilla extension. BugLocalizer is integrated to Bugzilla and git version control system and makes use of BugLocator bug localization technique.

The structure of the remainder of this paper is as follows. In Section 2, we briefly introduce Bugzilla and BugLocator. In Section 3, we describe our tool BugLocalizer. Related work is presented in Section 4. Section 5 concludes and describes future work.

FSE'14, November 16–21, 2014, Hong Kong, China
Copyright 2014 ACM 978-1-4503-3056-5/14/11...$15.00
http://dx.doi.org/10.1145/2635868.2661678

2. PRELIMINARIES

In this section, we first briefly describe Bugzilla, a bug tracking system on which our tool is build upon. Next, we describe information retrieval (IR) based bug localization.

2.1 Bugzilla

Bugzilla[1] is a web-based bug tracker. Users can report new bugs and developers can track the status of bugs that are reported. It is used by well-known organisations such as Mozilla and Apache and open-source projects such as Eclipse and LibreOffice.

A bug report in Bugzilla contains a number of fields such as bug id, product, component, assignee, summary and description. Each of them carries a piece of information. In this work, we are interested in the following two textual fields: (1) summary, (2) description. Summary is a short synopsis of a bug, while description is a longer text that elaborates the bug.

2.2 IR Based Bug Localization

IR based bug localization takes as input a textual bug report and a set of program source code files. Its goal is to output a ranked list of files sorted by their likelihood to be buggy (i.e., contain the bug described in the report). The ranked list is then returned to developers who can then manually inspect files one-by-one from the beginning of the list until the buggy files are found. The earlier the buggy files appear in the ranked list, the more effective the bug localization technique is.

The intuition behind IR-based bug localization is that textual bug reports and their corresponding buggy source code files tend to share common words. Furthermore, if a file has greater textual similarity to a bug report than other source code files, the file is likely to be the buggy one. By utilizing a suitable text retrieval model, an IR-based bug localization technique calculates textual similarity scores between a bug report and source code files. Later, files are sorted in descending order of their textual similarity scores to create a ranked list of files for developer inspection.

IR-based bug localization considers both input bug reports and source code files as textual documents. It extracts textual contents from summary and description fields of bug reports and it extracts identifiers and comments from source code files. These documents go through text preprocessing procedure before the computation of textual similarity scores. The purpose of this procedure is to make the similarity of a bug report and a source code file more apparent. In the text preprocessing procedure, there are three main steps. They are text normalization, stopword removal, and stemming. The following is the description of the three steps:

- *Text Normalization.* In this step, punctuation marks and special symbols are deleted from documents (i.e., bug reports and source code files). Then, documents are split into constituent words. For source code files, identifiers are split into smaller words following the Camel casing convention (e.g., "bugLocalization" is split into "bug" and "localization").

- *Stopword Removal.* In this step, English stopwords are deleted from documents (i.e., bug reports and source code files). These words frequently appear in many documents. Thus, they are not too helpful to differentiate a document from the other ones.

- *Stemming.* In this step, we transform English words to their root form. For example, "connection", "connecting", and "connected" are all reduced by stemming to "connect". In our study, we apply Porter stemming algorithm to perform this step [8].

[1]`http://www.bugzilla.org/`

After text preprocessing, the input bug report is compared to source code files using a suitable text retrieval model. In general, this model estimates a weight for each word in the pre-processed documents. Next, weights of common words that appear in both the bug report and a source code file are used to compute their textual similarity score.

Recently, Zhou et al. propose a technique named BugLocator [14] which is based on Vector Space Model (VSM). The following paragraphs describe VSM and then BugLocator in more detail.

Vector Space Model. Vector Space Model (VSM) represents a document as a vector of weights, where each weight corresponds to a word in the document. Weights of words are usually estimated based on the standard *term frequency-inverse document frequency* (tf-idf) weighting scheme [6]. The tf-idf weight of word w in document d given a set of documents D (denoted as $\textit{tf-idf}(w, d, D)$) is calculated as:

$$\textit{tf-idf}(w, d, D) = \log(f(w, d)+1) \times \log \frac{\mid D \mid}{\mid \{d_i \mid w \in d_i \wedge d_i \in D\} \mid}$$

where $f(w, d)$ is the frequency of occurrences of word w in document d, and $w \in d_i$ represents that word w appears in document d_i. The textual similarity between two documents q and d is obtained by calculating the *cosine similarity* of the two vectors representing q and d [6]. We denote the textual similarity score between two documents q and d as computed by VSM as $VSM(q, d)$

BugLocator. BugLocator proposes revised Vector Space Model (rVSM) which is an extension of the standard Vector Space Model (VSM). The intuition behind rVSM is that larger files are more likely to have bugs [14]. Therefore, using rVSM, files with larger amount of source code are ranked higher than using standard VSM. The following is the formula to compute the textual similarity between bug report q and source code file d using rVSM (denoted as $rVSMScore(q, d)$)

$$rVSMScore(q, d) = g(\#word(d)) \times VSM(q, d)$$

where $g(\#word(d))$ is a function that takes as input the number of words that d has (i.e., $\#word(d)$), and $VSM(q, d)$ is the VSM textual similarity of q and d. According to Zhou et al., $g(x) = \frac{1}{1+e^{-x}}$ achieves the best performance [14].

BugLocator also utilizes information gathered from similar past fixed bug reports to improve bug localization performance based on the rationale that textually similar bug reports tend to have similar buggy files. BugLocator constructs a three-layer heterogeneous graph, which captures past fixed reports that are similar to an input bug report q and their buggy files, to calculate another similarity score $SimiScore(q, d)$, for each source code file d [14]. Then, the final textual similarity score between bug report q and source code file d (denoted as $FinalScore(q, d)$) is calculated as:

$$\begin{aligned} FinalScore(q, d) = {} & (1 - \alpha) \times rVSMScore(q, d) \\ & + \alpha \times SimiScore(q, d) \end{aligned}$$

where $0 \leq \alpha \leq 1$.

3. BUGLOCALIZER

In this section, we describe the architecture of BugLocalizer, the bug localization algorithm that we use to return potential buggy files, some implementation details, and usage scenarios on how BugLocalizer can be used.

System Architecture. BugLocalizer architecture consists of a client-side component and a server-side component. The client-side takes an input query from user (i.e., developer) and communicates with the server-side component to retrieve potential buggy files given a query. A user query is a combination of the summary and description of a bug report, a URL to a source code repository, and a version of the repository to be searched to find buggy files. The client-side component is added to the bug report view page (i.e., the page where a user sees and edits existing bug reports) of Bugzilla. The server-side component takes as input a user query and, based on summary and description information, localizes the described bug to files in the requested version of the source code repository.

Algorithm. Algorithm 1 shows the pseudocode of the procedure that we use to localize bug; it is implemented in the server-side component. This algorithm is based on Zhou et al.'s work [14]. It accepts as input a bug report (Br), a source code repository ($Repo$), a version of the repository where the buggy files are to be found ($TargetVersion$), a set of historical fixed bug reports ($FixedReports$), a similarity weighting factor (α), and a number of potential buggy files to return (k). The algorithm returns a ranked list of k files in $Repo$ that are most likely to contain the root cause of the bug described in Br. At line 1, it retrieves $Files$ from $TargetVersion$ of the $Repo$. It then constructs a three-layer heterogeneous bug-file graph BFG from Br, $FixedReports$, and files that were changed to fix them (line 2). At line 3, it performs text preprocessing on the textual contents in the bug report. Next, it iterates through every file in $Files$ (line 4). For each file, at line 5, it preprocesses its content. It then computes the $rVSMScore$ and $SimiScore$ for each file and Br (lines 6-7). Next, for each file $File$, it computes the final score $File.FinalScore$ by linearly combining the two similarity scores according to parameter α. At line 10, it then sorts $Files$ based on their final similarity scores. Finally, at line 11, top-k files with the highest scores are returned. By default, we set k and α to 10 and 0.2, respectively.

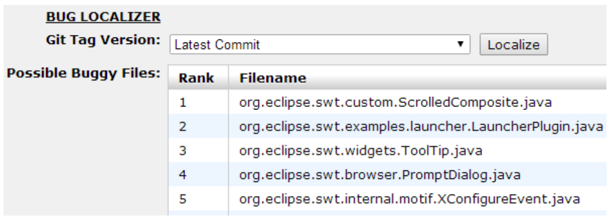
Implementation Details. Both client-side and server-side components of BugLocalizer are implemented by extending templates and codes of Bugzilla. It follows a system of hooks that is provided by Bugzilla for supporting extensions[2]. For the bug localization algorithm, we make use of the original implementation of BugLocator by Zhou et al. [14]. The current version of BugLocalizer supports git repository. Git is chosen due to performance consideration. Its decentralized design allows for faster query and retrieval of source code files as compared to centralized version control system, e.g.,

[2]`http://www.bugzilla.org/docs/tip/en/html/api/Bugzilla/Extension.html`

Algorithm 1 LocalizeBug

```
Input: $Br$: bug report
       $Repo$: source code repository
       $TargetVersion$: target source code version
       $FixedReports$: historical fixed bug reports
       $\alpha$: similarity weighting factor
       $k$: number of potential buggy files to return
Output: top-$k$ potential buggy files
1  $Files \leftarrow Repo.GetSourceCodeFiles(TargetVersion)$
2  $BFG \leftarrow ConstructBugFileGraph(Br, FixedReports, Files)$
3  $Br.Words \leftarrow Preprocess(Br.Summary, Br.Desc)$
4  foreach $File \in Files$ do
5      $File.Words \leftarrow Preprocess(File.Content)$
6      $File.rVSMScore \leftarrow rVSM(Br, File)$
7      $File.SimiScore \leftarrow BFG.Simi(Br, File)$
8      $File.FinalScore \leftarrow (1 - \alpha) \times File.rVSMScore + \alpha \times File.SimiScore$
9  end
10 Sort $Files$ by $File.FinalScore$
11 return top-$k$ potential buggy files
```

Subversion. Installation-wise, assuming Bugzilla and Git are installed, it only requires a few operations to set up BugLocalizer, e.g., copying files and setting up permission. Detailed installation instruction is provided in BugLocalizer's GitHub page. We have also performed a preliminary performance test by running BugLocalizer 10 times using various textual queries (to represent bug reports) on a sample software corpus and recording the average server response time, which is 11.3 seconds. Note that the current implementation is not optimized yet.

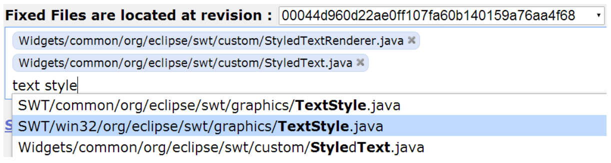


Git Tag Version: Latest Commit

Possible Buggy Files:

Rank	Filename
1	org.eclipse.swt.custom.ScrolledComposite.java
2	org.eclipse.swt.examples.launcher.LauncherPlugin.java
3	org.eclipse.swt.widgets.ToolTip.java
4	org.eclipse.swt.browser.PromptDialog.java
5	org.eclipse.swt.internal.motif.XConfigureEvent.java

Figure 1: BugLocalizer Main User Interface



Fixed Files are located at revision : 00044d960d22ae0ff107fa60b140159a76aa4f68

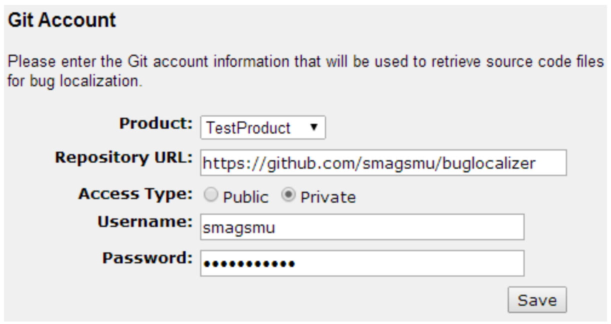
Widgets/common/org/eclipse/swt/custom/StyledTextRenderer.java
Widgets/common/org/eclipse/swt/custom/StyledText.java
text style
SWT/common/org/eclipse/swt/graphics/**TextStyle**.java
SWT/win32/org/eclipse/swt/graphics/**TextStyle**.java
Widgets/common/org/eclipse/swt/custom/**Styl**e**dText**.java

Figure 2: Recording Fixed Files Information

Usage Scenario. Figure 1 shows the interface of the bug report view page of Bugzilla when BugLocalizer is enabled and used. We show the interface when a bug report has been processed resulting in a list of top-k files that are deemed to be most likely to be buggy among files in the latest commit of the project's git repository. As shown in the figure, a user can pick a source code version whose files will be searched to identify buggy ones (by selecting a version from "Git Tag Version" drop down list). After choosing the version, the user can click "Localize" button to request for bug localization to be performed on the chosen source code version based on the summary and description fields of a bug report (these fields are not shown in the figure). A background script will then asynchronously send a request to the server (i.e., server-side component), inform the user that the request is currently being processed, and wait for the server response. The server accepts the request and runs Algorithm 1 to find top-k potentially buggy files. Right after the algorithm's completion, the server will send a list of top-k potential buggy files as the response to the client request. The background script in the client will then receive the server response and display a table containing the top-k potential buggy files as shown in Figure 1. The table lists the ranks and names of the potentially buggy files. A higher rank indicates that the corresponding file is more likely to be buggy. The user can then inspect the files starting from the top to the bottom of the list.

Aside from the above main usage scenario, BugLocalizer has two other usage scenarios that support the main one. The first usage scenario is a support for recording a list of buggy files for a fixed bug report. This is essential for computing *SimiScore* which is calculated based on buggy files of past similar fixed bug reports. Figure 2 shows the interface for recording these buggy files. This interface will only appear in the bug report view page when a user changes the bug resolution status to "FIXED". It requires the user to choose in which git revision the files were fixed. The user can then type the name of the fixed files in the given input area. Note that not all files that are changed in a bug fixing commit are buggy ones; some non-buggy files are changed since developers mix bug fixing with refactoring (c.f., [2, 11]). A list of matched file names that exist in the chosen git revision will be recommended to the user while the user types a file name. The fixed files information will be submitted to the server. The server will record it and use it when localizing future bugs.

The second usage scenario is to support git repository integration. Figure 3 shows the interface for linking BugLocalizer to a git repository. A user can input the repository URL for each product (i.e., a major sub-part of a software) and specify whether the repository is a public or private one. If it is not publicly accessible, the user needs to further specify a valid username and password for accessing the repository. After all the necessary information are input, the user can submit the git repository setting to the server.


Git Account

Please enter the Git account information that will be used to retrieve source code files for bug localization.

Product: TestProduct
Repository URL: https://github.com/smagsmu/buglocalizer
Access Type: Public [✓ Private]
Username: smagsmu
Password: •••••••••••

Figure 3: Configuring Git Setting for Bug Localization

4. RELATED WORK

Rao and Kak employed several popular IR techniques for bug localization and evaluated their performance [9]. Lukins et al. applied Latent Dirichlet Allocation (LDA) for bug localization [5]. Marcus and Maletic utilized Latent Semantic Indexing (LSI) for recovering document to source code traceability links [7]. Zhou et al. proposed BugLocator, a bug localization tool that considers source code file size and historical fixed reports to rank potentially buggy files [14]. Saha et al. utilized the structure of source code files and bug reports to build a structured retrieval model for bug localization [10]. Le et al. applied Latent Dirichlet Allocation (LDA) technique multiple times to create a hierarchy of different topic models to localize bugs [3]. Wang et al. use information in version history, similar report, and bug report structure to improve bug localization [12]; they also use genetic algorithm to compose a number of Vector Space Model (VSM) variants for improved bug localization [13].

In this work, we integrate one of the state-of-the-art bug localization techniques (i.e., BugLocator) to a popular bug tracking system (i.e., Bugzilla) and a popular version control system (i.e., git).

5. CONCLUSION AND FUTURE WORK

Bug localization is one of the active research areas in software engineering. Different kinds of bug localization techniques have been proposed in the last few years. However, they usually end up only as research prototypes; unreachable and inaccessible to practitioners. In an effort to bring recent advances in bug localization closer to the hand of practitioners, we have developed BugLocalizer that integrates a recently proposed bug localization technique, named BugLocator, to Bugzilla and git. In the future, we plan to incorporate other bug localization techniques to BugLocalizer. We also plan to perform a user study to understand how practitioners use BugLocalizer, and get better insight on its strengths and weaknesses, and how to improve it.

6. REFERENCES

[1] J. Anvik, L. Hiew, and G. C. Murphy. Coping with an open bug repository. In *ETX*, 2005.
[2] D. Kawrykow and M. P. Robillard. Non-essential changes in version histories. In *ICSE*, pages 351–360, 2011.
[3] T.-D. B. Le, S. Wang, and D. Lo. Multi-abstraction concern localization. In *ICSM*, 2013.
[4] Lucia, F. Thung, D. Lo, and L. Jiang. Are faults localizable? In *MSR*, pages 74–77, 2012.
[5] S. K. Lukins, N. A. Kraft, and L. H. Etzkorn. Bug localization using latent dirichlet allocation. *Information and Software Technology*, 52(9):972–990, 2010.
[6] C. Manning, P. Raghavan, and H. Schutze. *Introduction to Information Retrieval.* Cambridge, 2008.
[7] A. Marcus and J. I. Maletic. Recovering documentation-to-source-code traceability links using latent semantic indexing. In *ICSE*, 2003.
[8] M. Porter. An algorithm for suffix stripping. *Program*, 1980.
[9] S. Rao and A. Kak. Retrieval from software libraries for bug localization: a comparative study of generic and composite text models. In *MSR*, 2011.
[10] R. K. Saha, M. Lease, S. Khurshid, and D. E. Perry. Improving bug localization using structured information retrieval. In *ASE*, pages 345–355, 2013.
[11] F. Thung, D. Lo, and L. Jiang. Automatic recovery of root causes from bug-fixing changes. In *WCRE*, 2013.
[12] S. Wang and D. Lo. Version history, similar report, and structure: Putting them together for improved bug localization. In *ICPC*, 2014.
[13] S. Wang, D. Lo, and J. Lawall. Compositional vector space models for improved bug localization. In *ICSME*, 2014.
[14] 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*, 2012.