Abstract—Numerous efforts have been invested in improving the effectiveness of bug localization techniques, whereas little attention is paid to making these tools run more efficiently in continuously evolving software repositories. This paper first analyzes the information retrieval model behind a classic bug localization tool, BugLocator, and builds a mathematical foundation illustrating that the model can be updated incrementally when codebase or bug reports evolve. Then, we present IncBL, a tool for Incremental Bug Localization in evolving software repositories. IncBL is evaluated on the Bugzbook dataset, and the results show that IncBL can significantly reduce the running time by 77.79% on average compared with the re-computing the model, while maintaining the same level of accuracy. We also implement IncBL as a Github App that can be easily integrated into open-source projects on GitHub. Users can deploy and use IncBL locally as well. The demo video for IncBL can be viewed at https://youtu.be/G4gMuvlJSb0, and the source code can be found at https://github.com/soarsmu/IncBL.

Index Terms—Bug Localization, Information Retrieval, Mining Software Repository

I. INTRODUCTION

Information retrieval-based bug localization (IRBL) is a popular research topic in software engineering and has shown promising results in the last decade. The basic idea behind IRBL is to model the bug localization as an ad-hoc document search problem. Numerous research efforts have been invested in developing IRBL techniques [1]–[3]. However, such research mainly aims to achieve better retrieval accuracy rather than make tools more efficient in software repositories where codebases evolve and new bug reports emerge. As software repositories are evolving, these tools need to re-process all files and update the entire model representations (e.g., tf-idf weights) to ensure accuracy, even for some minor changes (e.g., adding or deleting several lines of code), which limits their usages in time-sensitive or compute-intensive scenarios.

To improve the efficiency of IRBL, Rao et al. [4] proposed an incremental update framework, but their approach cannot tackle some cases, e.g., when the number of documents changes1. Moreover, their method has only been applied to bug localization methods that purely rely on Bag-of-Words (BoW) models. Last but not least, there is no open-source artifact or tool that practitioners can adopt. The above facts motivate us to develop an incremental bug localization tool that addresses the limitations of Rao et al.'s work.

Akbar and Kak [1] divided IRBL tools that were published between 2004 and 2019 into three generations. The 1st generation tools that are solely based on BoW models perform the worst, while 3rd generation tools that utilize term order and semantics, e.g., SCOR [5], require much time and computing resources to train models. Moreover, none of the tools specified as 3rd generation in [2] is made publicly available. The 2nd generation tools use structural information in codebases and software evolution information (e.g., historical bug reports) to improve accuracy. Two representatives of the 2nd generation tools, BugLocator [2] and BLuir [3], are evaluated in [1], and the empirical study shows that BugLocator outperforms BLuir on Bugzbook [2], a dataset containing bug reports from 27 large open-source repositories. Thus, we decide to build our incremental bug localization solution (IncBL) on top of BugLocator for the following reasons: It does not require as much time and computing resources as the 3rd generation tools while performing better than many other 2nd generation tools and the 1st generation tools. Moreover, BugLocator is a popular tool and many other tools are built on top of it, e.g., the recently proposed Legion tool implemented in Adobe [6]. Although this work focuses on making BugLocator incremental, the incremental design used in IncBL can also potentially be translated into other solutions that are based on BugLocator or have similar features to it.

IncBL performs a complete computation and retains the computed information, e.g., term frequency, document frequency, and other model parameters when it is applied on the underlying repository for the first time. Once a new bug report is raised, IncBL only updates the corresponding information, e.g., tf-idf weights, incrementally rather than re-compute the entire model, which can significantly reduce redundant computation and retrieval latency. We evaluate IncBL on Bugzbook dataset. Our evaluation results show that the IncBL can run 4.5 times faster (i.e., reducing 77.79% of the running time on average) than the original BugLocator, without sacrificing the accuracy.

To promote the usage of IRBL in practices and alleviates debugging costs for developers, we implement IncBL as a Github App, which can be installed in GitHub to locate potential potential buggy files for issues tagged as ‘bug’ or be deployed locally for the same functionality.

The rest of this paper is organized as follows. Section II

---

1 More detailed discussion can be found in our online appendix, that we make available at https://github.com/soarsmu/IncBL/blob/main/discussion/appendix.pdf
introduces the retrieval model and workflow of BugLocator. Section III describes the design and implementation of main features, and usage scenario of IncBL. Section IV reports the evaluation results of IncBL on the Bugzbook dataset. After surveying the related work in Section V, we conclude the paper and present future work in Section VI.

II. BUGLOCATOR

IncBL extends BugLocator [2] with the support for incremental computing and integration into the GitHub platform. In this section, we briefly introduce the workflow of BugLocator and the information retrieval model used.

Step 1. Processing code files. BugLocator preprocesses each Java code file in the codebase as follows. First, it utilizes a Java parser to extract identifiers (e.g., package name, method name, etc.) and then appends them to the original code contents. After that, BugLocator performs stemming and stop-words removal to produce a code corpus. A Vector Space Model (VSM) (more details of the incremental version of VSM will be given in Section III-A) is then used to vectorize each document in the code corpus so that further steps (e.g., computing similarity) can be performed.

Step 2. Processing bug reports. BugLocator first combines bug report titles with descriptions. Then, a bug corpus is created after performing stemming and stop-words removal on the combined documents. An important feature of BugLocator is to leverage bug-fixing history information to help rank faulty files, by computing the similarity between a bug report and all the past fixed bug reports. The intuition is that the current new bug report and the similar past fixed bug reports are likely to share the same source code files to be modified. BugLocator uses the VSM to compute similarities between bug reports. For a bug report B and the mth file referred by at least one bug report, their SimiScore is computed by:

\[
\text{SimiScore} = \frac{\sum_{B' \in \ell(m)} \text{Similarity}(B, B')}{|\ell(m)|} \tag{1}
\]

where \(\ell(m)\) is all the bug reports linked to the mth file and \(\text{Similarity}(B, B')\) is the similarity between B and B' computed using VSM. For a bug report and a code file that is not linked to any past bug report, their SimiScore is 0.

Step 3. Localizing buggy files. BugLocator defines and computes the relevance score between bug reports and source code files. A VSM requires documents and queries to share the same vocabulary set, so BugLocator discards all the terms that appear in bug reports but not in the code corpus and then use the VSM produced in Step 1 to compute the VSMScore between a document (source code file) d and a query (bug report) q. Besides, BugLocator revises the VSM model by weighting VSM with a function g to favors long documents during ranking. BugLocator defines g as

\[
g = \frac{1}{1 + e^{-N(\text{terms})}}, \quad \text{where } N(x) = \frac{x - x_{\text{min}}}{x_{\text{max}} - x_{\text{min}}} \tag{2}
\]

The \(|\text{terms}|\) is the number of terms in preprocessed code files and \(N(x)\) is a max-min normalization function. The final relevance score between the bug reports and source code files is a linear combination of revised VSMScore and SimiScore:

\[
\text{relevance} = \alpha \times g \times \text{VSMScore} + (1 - \alpha) \times \text{SimiScore} \tag{3}
\]

where \(0 \leq \alpha \leq 1\), and the performance is best empirically [2] when \(\alpha\) is between 0.2 and 0.3.

III. DESIGN AND USAGE CASE OF INCBL

In this section, we introduce how the VSM used in BugLocator can be updated incrementally and how IncBL incorporates incremental computing in the tool. We also present how users can utilize IncBL as a GitHub App and a locally-deployed tool.

A. Incremental Design

VSM is used to represent a collection of documents (the code corpus and bug report corpus in this paper). First, we need to create \(V\), which is the vocabulary of terms appearing in documents, and a term-document matrix \(A\), whose size is \(M \times |V|\), where \(M\) is the number of documents. \(A_m(w)\) represents the occurrence count of the \(w\)th term in \(V\). In a VSM, each document is represented as a vector with the size of \(|V|\). The \(w\)th value of this vector is the tf-idf weight that is computed by \(t_f(w) \times \text{idf}(w)\). The term frequency \(t_f(w)\) has different definitions over \(A\), and BugLocator computes it by \(t_f(w) = \log(A_m(w) + 1)\).

The \(\text{idf}(w)\), called inverse document frequency, is computed by \(\log \left( \frac{M}{\text{df}(w)} + 1 \right)\) where \(\text{df}(w)\) refers to document frequency, total number of documents that have the \(w\)th term.

We concatenate representations for each source file into a big matrix, denoted by \(D\), in which \(D_m(w)\) presents the tf-idf weight of the \(w\)th term in the \(m\)th file. Note that an element \(D_m(w)\) is computed with \(\log(A_m(w) + 1) \times \frac{M}{\text{df}(w) + 1}\), meaning that \(D_m(w)\) needs to be updated only when at least one of the three values (i.e., \(A_m(w)\), \(\text{df}(w)\), and \(M\)) changes.

Next, we explain how to incrementally update \(D\) when documents change. At the document level, changes can be categorized into one or a combination of the following three atomic changes: (1) adding a new document, (2) deleting an existing document, (3) modifying an existing document, which corresponds to adding, deleting, and modifying a row in \(A\).

At the term level, the atomic changes are: (1) adding a new term, (2) deleting an existing term, and (3) changing the term frequency of an existing term in a document.

Let us consider two cases: 1) \(M\) does not change and 2) \(M\) changes. For the case in which \(M\) does not change, only \(\text{df}\) and \(A\) have an influence on \(D\). We first elaborate on how the term-document matrix \(A\) can be updated. The update is straightforward. We first update the columns (term-level) and then update the rows (document-level) in \(A\). If the \(w\)th term is added, we append a new column to \(A\) and set all the values as 0. If the term frequency value of the \(w\)th term in \(m\)th document...
changes, we just need to set the corresponding position \(A_m(w)\) to a new number. Then, we can update rows in a similar way. if the \(m^{th}\) document is deleted, we delete the \(m^{th}\) row in \(A\). If a new document is added, we append a new row to \(A\) and set this row as the term frequency of this newly added document. When the \(m^{th}\) document is changed, the \(m^{th}\) row in \(A\) should be updated according to the modified term frequency of the document. We next elaborate how document frequency vector should be updated. Assuming the \(m^{th}\) file is affected, the \(df(w)\) can be updated with the following formula:

\[
df^{new}(w) = df^{old}(w) + [\text{sign}(A^{new}_m(w)) - \text{sign}(A^{old}_m(w))]
\]

(4)

where \(\text{sign}(\cdot)\) returns +1 for positive inputs, −1 for negative inputs, and 0 for 0. \(A^{old}\) and \(A^{new}\) are term-document matrices before and after this change. If the file \(m\) or term \(w\) are not in \(A^{new}\) (or \(A^{old}\)), we set \(A^{new}_m(w)\) (or \(A^{old}_m(w)\)) as 0.

We illustrate how the term-document matrix \(A\), the document frequency vector \(df\), and document representation matrix \(D\) can be updated according to the changes of documents and terms in Fig. 1. We label the columns in \(A\) as green and red to represent term addition and deletion respectively. We highlight the rows in \(A\) as green and red to represent document addition and deletion. Some blocks are highlighted with orange, which means that term frequency values at the corresponding positions are updated. We label the changes in \(df\) using the same color scheme. \(D\) is generated by an element-wise production between \(df\) and \(A\) (i.e., \(df \odot A = D\)), therefore in \(D\), only the blocks that are produced by the changed portion in \(df\) and \(A\) need updates.

So far, we have explained how term-document matrix \(A\) and document frequency vector \(df\) can be updated incrementally. Considering the changes between two versions are small (several commits), most elements in \(A\) and \(df\) will remain the same. As a result, when \(M\) does not change, we can update \(D\) incrementally at a very low cost: only positions highlighted with a color need to be updated.

However, if \(M\) changes, the whole \(idf\) vector needs updates, meaning that we must update all the values in \(idf\), and all model parameters must be recomputed. We take this case in consideration as follows. We have \(idf(w) = \log\left(\frac{M}{df(w) + 1}\right)\), which can be transformed to \(\log(M) - \log(df(w) + 1)\). We can update \(idf(w)\) by

\[
idf^{new}(w) = idf^{old}(w) + \log\left(\frac{M + \Delta M}{M}\right)
\]

(5)

where \(\Delta M\) is the change on \(M\). In other words, we use the old \(idf\) that is not affected to update the value of \(idf\) incrementally rather than computing the \(idf\) value from scratch.

Such changes are widespread in software development as codebases are continuously evolving and new bug reports are raised. We use this incremental model to update the VSMs in Step 1 and 2 of the BugLocator as described in Section II. The updating method also indicates that IncBL only needs to preprocess the affected files rather than act like BugLocator who re-processes all files in the repositories each time it runs, which saves time even further.

B. Implementation and Usage

To promote the use of bug localization tools, we implement IncBL as a GitHub App. With some simple settings given in the tool homepage (https://github.com/apps/incbl), GitHub users can easily install IncBL in their public projects. Once IncBL is installed, it will automatically analyze the codebases and past bug reports. Each time when a new issue tagged with ‘bug’ is raised, IncBL updates models incrementally and locates relevant buggy files for this issue. After files are retrieved, IncBL posts the top 10 most relevant files for the issue so developers can get notified. Please refer to our video (https://youtu.be/G4gMuvlJSb0) to see IncBL in action. Users can also deploy IncBL locally on their own machines. Only a simple Python command is needed to run IncBL. Users just need to specify the path to the codebases under analysis, the path to bug reports, and the path for storage.

IV. Evaluation

A. Experimental setting

We use Bugzbook [1] as our benchmark to evaluate IncBL. Bugzbook contains releases of 27 software projects and bug reports that correspond to each release. Following Akbar and Kak [1], we focus on Java, Python, C, and C++ code files, and discard other files in the codebase as well as bug reports linked to other file types in this experiment. We end up with 43,017 bug reports.

We aim to examine if IncBL can run more efficiently without losing any accuracy. For this purpose, we measure and compare the running time of IncBL and re-computing from scratch. To measure the running time of IncBL, we consider the usage scenario mentioned in Section III-B: a user raises an issue in GitHub and then IncBL updates the VSMs incrementally to localize buggy files with lower latency.
Therefore, improvement of the efficiency of bug localization tools is needed. However, there is no other studies in the research community apart from Rao et al.’s framework in [4] and [7]. The limitations of [4] were discussed in Section I, and are addressed by IncBL. [7] only considers incremental LSA and LSA has been shown to perform worse than VSM in bug localization [8]. Moreover, IncBL is the first incremental bug localization tool integrated as a GitHub App. BugLocalizer [9] is a tool that implements BugLocator as a Bugzilla plugin. However, it does not support incremental updates.

VI. CONCLUSION AND FUTURE WORK

To help developers localize bugs, this paper presents IncBL, which can update the VSM incrementally and avoid repetitive computation, on top of BugLocator. Our evaluation shows that on average IncBL can run 4.5 times faster than re-computing the model from scratch while maintaining the same level of accuracy. We implement IncBL as a GitHub App that can be easily installed to analyze public repositories on Github. When a new bug report is raised, IncBL will update the VSM incrementally and notify developers about the potentially buggy files. Users can also deploy IncBL on their own machine to analyze repositories locally. In the future, we plan to create variants of IncBL built on top of other bug localization tools.

ACKNOWLEDGMENT

This research was supported by the Singapore Ministry of Education Academic Research Fund (AcRF) Tier 1 grant.

REFERENCES