IncBL: Incremental Bug Localization

Zhou Yang, Jieke Shi
Singapore Management University
Singapore
{zyang, jkshi}@smu.edu.sg

Shaowei Wang
University of Manitoba
Canada
shaowei@cs.umanitoba.ca

David Lo
Singapore Management University
Singapore
davidlo@smu.edu.sg

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 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, and users can also deploy and use *IncBL* locally. 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 scenario..

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 changes[1]. Moreover, this incremental 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 is made available so that practitioners can adopt them. 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-term order and semantics, e.g., SCOR [5], require much time and computing resource to train models, which makes it harder for users to deploy locally. 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 realize 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 performs better than many other 2nd generation 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 be 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 applied on the underlying repository for the first time. When 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 incremental updating feature, 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 buggy files for issues tagged as "bug" or be deployed locally for the same functionality.

---

[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

The rest of this paper is organized as follows. Section II 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 stopwords removal to produce 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 reports title with descriptions. Then, it performs stemming and stopwords removal on the combined document to create a bug corpus. 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 code file referred by at least one bug report, their $SimiScore$ is computed by:

$$SimiScore = \sum_{B' \in l(m)} \frac{Similarity(B, B')}{|l(m)|} \qquad (1)$$

where $l(m)$ is all the bug reports linked to the mth code file and $Similarity(B, B')$ 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. The VSM models require documents and queries to share the same vocabulary set, so BugLocator discards all the terms that appear in bug reports but not in 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 $VSMScore$ with a function g to favors long documents during ranking. BugLocator defines g as

$$g = \frac{1}{1 + e^{-N(|terms|)}}, \text{ where } N(x) = \frac{x - x_{min}}{x_{max} - x_{min}} \qquad (2)$$

The $|terms|$ is the number of terms in preprocessed code file 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 $SimiScore$:

$$relevance = \alpha \times g \times VSMScore + (1-\alpha) \times SimiScore \qquad (3)$$

where $0 \leq \alpha \leq 1$, and the performance is best empirically when α 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 local-deployed tool.

A. Incremental Design

VSM is used to represent a collection of documents (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 wth term in V of the mth file. In a VSM model, each document is represented as a vector with the size of $|V|$. The wth value of this vector is the tf-idf weight that is computed by $tf_m(w) \times idf(w)$. The *term frequency* $tf_m(w)$ has different definitions over $A_m(w)$, and BugLocator computes it by $tf_m(w) = log(A_m(w)) + 1$. The $idf(w)$, called *inverse document frequency*, is computed by $idf(w) = log(\frac{M}{df(w)+1})$ where $df(w)$ refers to *document frequency*, total number of documents that have the wth 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 wth term in the mth file. Note that an element $D_m(w)$ is computed with $(log(A_m(w)) + 1) \times \frac{M}{df(w)+1}$, meaning that $D_m(w)$ needs to be updated only when at least one of the three values (i.e., $A_m(w)$, $df(w)$, 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 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 wth term is deleted, we delete the wth column in A. If a new term is added, we append a new column to A and set all the values as 0. If the term frequency value of the wth term in m^{th} document
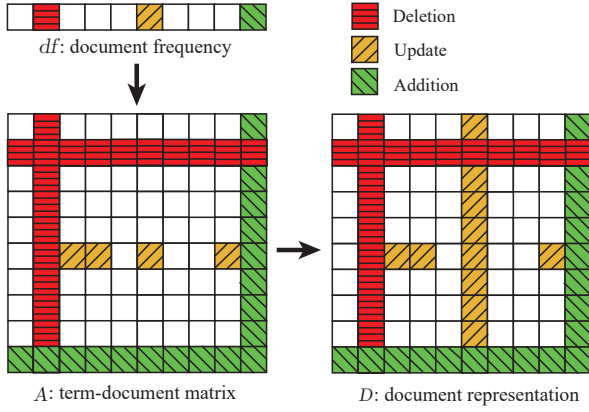
Fig. 1. This figure illustrates how term-document matrix (A), document frequency (df) and document representation (D) can be updated incrementally. Each row in D is the element-wise product of df and the corresponding row in A, i,e, $df \odot A = D$. Blocks being highlighted in red, green, and orange indicates that the corresponding positions in the matrices should be deleted, added, or modified, respectively. Colors in df and A will propagate to D, indicating that only the labeled blocks in D need to be updated.

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 mth document is deleted, we delete the mth 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 mth document is changed, the mth 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 mth file is affected, the $df(w)$ can be updated with the following formula:

$$df^{new}(w) = df^{old}(w) + [sign(A_m^{new}(w)) - sign(A_m^{old}(w))] \quad (4)$$

, where $sign(\cdot)$ 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 $A_m^{new}(w)$ (or $A_m^{old}(w)$) 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 portion of 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 $idf(w) = log(\frac{M}{df(w)+1})$, which can be transformed to $log(M) - log(df(w) + 1)$. We can update $idf(w)$ by

$$idf^{new}(w) = idf^{old}(w) + log(\frac{M + \Delta M}{M}) \quad (5)$$

where ΔM 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 being 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 acting 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* has been 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 report. 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 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 kbar 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 ended 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 VSM model of source code incrementally to localize buggy files
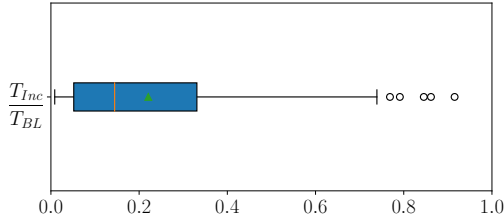
Fig. 2. Boxplot of the ratio of the running time to perform bug localization on all studied bug reports of *IncBL* over that of the original BugLocator ($\frac{T_{Inc}}{T_{BL}}$). The median and mean are 14.44% and 22.21%, corresponding to 6.30 and 4.50 times speed up.

with lower latency. We measure the running time that *IncBL* requires for locating potential buggy files for the raised bug report, denoted as T_{Inc}. We apply BugLocator on the usage scenario and measure its running time for each bug report, denoted as T_{BL}. Conducting the experiments on all collected bug reports is very time-consuming since BugLocator needs to compute everything from scratch. Therefore, we randomly sample 381 reports from all the studied bug reports, and this forms a statistically representative sample considering a 95% confidence level and 5% interval. We measure the running of *IncBL* and BugLocator on these sampled 318 reports. When measuring the accuracy, we keep the same setting as Bugzbook experiment: associating a group of bug reports with one software release. We run *IncBL* on all the 43,017 bug reports and incrementally updates VSM model representations for each new software release.

We run our experiments on a server with a 16-core CPU and 512 GB memory. The dataset, experimental results, and detailed instructions to run the experiments can be accessed via https://github.com/soarsmu/IncBL.

B. Results

IncBL can significantly reduce the running time for locating bugs in continuously evolving repositories by 77.79% (i.e., 4.5 times faster) on average compared with the original BugLocator while maintaining the same accuracy. Fig. 2 presents the boxplot of the ratio of running time between *IncBL* and BugLocator. The median and mean ratios are 14.44% and 22.21%. Wilcoxon signed-rank test shows that the reduction in running time is statistically significant (*p*-value < 0.01). The amount of time saved by the *IncBL* is mainly related to two factors: (1) the number of affected files and (2) the number of existing files. The more files change, the more updating operations are needed. Besides, if there are too many existing files, the VSM can be very large and *IncBL* needs more time to identify the correct locations in matrices to perform updates. On the Bugzbook dataset, *IncBL* achieves a Mean Average Precision (MAP) of 0.331 – on average, the correct files appear in the top 3 locations.

V. RELATED WORK

Although many efforts have been put into improving the effectiveness of bug localization techniques [1]–[3], the long running time of such tools can limit their real-world usages. 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 incremental 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 the 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, beyond BugLocator.

REFERENCES

[1] S. A. Akbar and A. C. Kak, "A Large-Scale Comparative Evaluation of IR-Based Tools for Bug Localization," in *Proceedings - 2020 IEEE/ACM 17th International Conference on Mining Software Repositories, MSR 2020*, 2020, pp. 21–31.

[2] 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 *Proceedings - International Conference on Software Engineering*, ser. ICSE '12. IEEE Press, 2012, pp. 14–24.

[3] R. K. Saha, M. Lease, S. Khurshid, and D. E. Perry, "Improving bug localization using structured information retrieval," in *2013 28th IEEE/ACM International Conference on Automated Software Engineering, ASE 2013 - Proceedings*, ser. ASE'13. IEEE Press, 2013, pp. 345–355. [Online]. Available: https://doi.org/10.1109/ASE.2013.6693093

[4] S. Rao, H. Medeiros, and A. Kak, "An incremental update framework for efficient retrieval from software libraries for bug localization," in *Proceedings - Working Conference on Reverse Engineering, WCRE*, 2013, pp. 62–71.

[5] S. A. Akbar and A. C. Kak, "SCOR: Source code retrieval with semantics and order," in *IEEE International Working Conference on Mining Software Repositories*, vol. 2019-May, 2019, pp. 1–12.

[6] D. Jarman, J. Berry, R. Smith, F. Thung, and D. Lo, "Legion: Massively composing rankers for improved bug localization at adobe," *IEEE Transactions on Software Engineering*, pp. 1–1, 2021.

[7] S. Rao, H. Medeiros, and A. Kak, "Comparing Incremental Latent Semantic Analysis Algorithms for Efficient Retrieval from Software Libraries for Bug Localization," *ACM SIGSOFT Software Engineering Notes*, vol. 40, no. 1, pp. 1–8, 2015.

[8] S. Rao and A. Kak, "Retrieval from software libraries for bug localization: a comparative study of generic and composite text models," in *Proceedings of the 8th Working Conference on Mining Software Repositories*, 2011, pp. 43–52.

[9] F. Thung, T. D. B. Le, P. S. Kochhar, and D. Lo, "BugLocalizer: Integrated tool support for bug localization," in *Proceedings of the ACM SIGSOFT Symposium on the Foundations of Software Engineering*, vol. 16-21-November-2014, 2014, pp. 767–770.