This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TSE.2021.3075215, IEEE Transactions on Software Engineering

Legion: Massively Composing Rankers for Improved Bug Localization at Adobe

Darryl Jarman, Jeffrey Berry, Riley Smith, Ferdian Thung, and David Lo

Abstract—Studies have estimated that, in industrial settings, developers spend between 30% and 90% of their time fixing bugs. As such, tools that assist in identifying the location of bugs provide value by reducing debugging costs. One such tool is BugLocator. This study initially aimed to determine if developers working on the Adobe Analytics product could use BugLocator. The initial results show that BugLocator achieves a similar accuracy on 5 of 7 Adobe Analytics repositories and on open-source projects. However, these results do not meet the minimum applicability requirement deemed necessary by Adobe Analytics developers prior to possible adoption. Thus, we consequently examine how BugLocator can achieve the targeted accuracy with two extensions: (1) adding more data corpora, and (2) massively composing individual rankers consisting of augmented BugLocator instances trained on various combinations of corpora and parameter configurations with a Random Forest model. We refer to our final extension as Legion. On average, applying Legion to Adobe Analytics repositories results in at least one buggy file ranked in the top-10 recommendations 76.8% of the time for customer-reported bugs across all 7 repositories. This represents a substantial improvement over BugLocator of 36.4%, and satisfies the minimum applicability requirement. Additionally, our extensions boost Mean Average Precision by 107.7%, Mean Reciprocal Rank by 86.1%, Top 1 by 143.4% and Top 5 by 58.1%.

Index Terms—bug localization; information retrieval; bug reports; data augmentation, ranker composition, industrial study

1 Introduction

Software testing and debugging are labor-intense activities, accounting for 30% to 90% of labor spent for a project [1]. Analysis suggests that sufficient testing and debugging infrastructure can reduce software errors that cost the U.S. economy billions of dollars [2]. Automatic bug localization is one of the research domains that aim to limit and reduce the cost of debugging software. It works by automatically identifying a ranked list of potentially buggy files to inspect given a bug report.

In this paper, we report our experience of applying a bug localization solution in industry, specifically on Adobe Analytics repositories. Originating from web analytics, Adobe Analytics[1] provides industry-leading solutions for applying real-time analytics and detailed segmentation across marketing channels. After collecting data across many digital platforms using various application programming interfaces, users act on the data by creating and pushing content to targeted segments using the full suite of Adobe Analytics products. In this way, Adobe Analytics provides users with the ability to action on valuable insights to improve key business processes. As the product has evolved, engineering processes have grown more complex resulting in efforts to eliminate labor-intense activities.

We investigated an information retrieval approach for bug localization proposed by Zhou et al. [3]: BugLocator. BugLocator is well-known, and easy to re-implement and potentially extend. For these reasons, we use BugLocator (instead of other bug localization tools) in our study. In order for BugLocator to be adopted by Adobe Analytics developers, we required a high level of reliability, as a developer will accept limited failures. To assess this, we choose the Top N (i.e., percentage of times a file containing a bug is found in the top N recommendations) as the most intuitive evaluation metric for developers.

We consulted 4 developers at Adobe Analytics during water cooler discussions to determine their minimum thresholds for applicability. These discussions suggest that developers who have been working on a code repository for a long time tend to expect the tool to run faster and be more accurate. This makes sense since the effectiveness of the tool would need to be higher than their ability to guess or discover the buggy files within a similar time frame. On the other hand, a developer less experienced with the same code repository may find a slower or less accurate tool to be useful as long as it still exceeds their own ability during the same time frame. In addition, it may help them gain more familiarity with the code as the tool's recommendations implicitly expose relationships between files that they have not yet noticed. These discussions lead us to identify the following thresholds: (1) for developers new to a repository, the tool should identify a buggy file in the top 10 recommendations at least 70% of the time (*Top 10* score $\geq$ 70%) and (2) for developers familiar with a repository, it should identify a buggy file in the top 5 recommendations at least 80% of the time (*Top 5* score $\geq$ 80%).

We implemented BugLocator as proposed in [3], which we refer to as BL. However, this initial implementation did not meet the aforementioned applicability requirements for

- *D. Jarman, J. Berry, and R. Smith are with Adobe, Lehi, Utah, United States of America.*
  E-mail: {djarman, berry, rilsmith}@adobe.com
- *F. Thung and D. Lo are with the School of Information Systems, Singapore Management University, Singapore.*
  E-mail: {ferdianthung, davidlo}@smu.edu.sg

1. https://www.adobe.com/analytics/adobe-analytics.html

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TSE.2021.3075215, IEEE Transactions on Software Engineering

Adobe Analytics developers. In this work, we consequently show how to meet the requirements by extending BL in two ways. The first extension adds additional corpora that can be matched against an incoming bug report. BL only matches the incoming bug report against source code files and summary and description of prior bug reports. Here, we consider additional corpora that exist in the issue tracking and source code management systems. For example, these systems provide additional text corpora such as commit messages, file change histories, and issue comments. These additional corpora act as intermediate information by having characteristics of both code and human language. We refer to BL with this extension as BL+. The second extension ranks files based on BL+ *features* of files (extracted by running BL+ considering different configuration options) using a Random Forest classifier. We refer to this extension as Legion. In effect, Legion composes a large number of rankers (i.e., 2,772 variants of BL+). Using Legion, the Top 10 results exceed the minimum applicability requirement for developers new to a repository, with a result of 76.8%. Moreover, Legion improves over BL by 143.4% and 58.1% for Top 1 and Top 5, respectively.

The contributions of our work are as follows:

1) To the best of our knowledge, our study is the first that applies a bug localization approach to industrial data.
2) We have applied BugLocator (BL), a well known information retrieval based bug localization approach, and found that it does not meet our applicability requirements.
3) We propose two extensions to BL, dubbed BL+ and Legion, to achieve the applicability requirements. BL+ adds additional corpora while Legion composes different configurations of BL+ using Random Forest.
4) Our experiments show that Legion exceeds the minimum applicability requirement, improving over BL by an average of up to 143.4% in terms of various metrics.

This paper is structured as follows. Section 2 provides some background materials on bug localization and BugLocator (BL). Section 3 describes the experiment settings and results on applying BL in Adobe Analytics repositories. Section 4 presents our proposed extensions to BL. Section 5 describes experiments, research questions, and results for BL extensions (BL+ and Legion). Section 6 presents additional findings. Section 7 discusses lessons learned and threats to validity. Section 8 presents related work. Last but not least, Section 9 concludes the paper and presents some future work.

2 Preliminaries

2.1 Bug Localization

Bug localization is a technique that aims to find the location of a bug in the program source code according to the content of a bug report. Many bug localization approaches aim to find the relevant source code files that must be modified to fix the bug. These relevant files might be among thousands or even millions of files in the program source code, depending on the size of the program.

Figure 1 shows an example of a bug report in Adobe Analytics. It contains information such as *summary*, *description*, and *components*. Some information, such as reporter name, are redacted due to sensitivity. Bug localization approaches use this information to help in better localizing bugs to relevant source code files. For example, in Figure 1, words in the bug description can tell us the topic of the bug.

Bug localization approaches use this information to generate a ranked list of relevant source code files. This ranked list contains all the files in the code-base which are ordered based on their relevancy to the bug described in the bug report. The relevancy scores can be computed by considering different types of information. Each type of information produces a relevancy score according to a particular strategy. The relevancy scores can be aggregated in various ways, e.g., summation [4], empirical weighting [5], [3], or weights learned from machine learning algorithms such as Support Vector Machine [6] and Deep Learning [7].

2.2 BugLocator (BL)

BL is a well-known information-retrieval based bug localization approach [3]. BL makes use of the following information:

1) *Source code files.* Files that contain the same or similar words as found in the bug report are more likely to be relevant.
2) *Similar bug reports.* Past bug reports that are similar with the current bug report may share the same set of relevant source code files (i.e., files that need to be modified to fix the bug).
3) *File size.* Large files are more likely to contain bugs than small files.

Based on the above information, BL creates a ranked list that combines the ranking based on source code files and the ranking based on similar bug reports.

BL first represents bug reports and source code files in a Vector Space Model (VSM). For bug reports, words from the bug report's summary and description are extracted. Stop words are removed and the remaining words are reduced to their roots, i.e. stemmed (e.g. the word *driving* is reduced to the word *drive*). The resulting words are then represented as vectors in a VSM. For source code files, tokens are extracted, and symbols and numbers are removed. Programming language keywords (e.g. `for`, `if`) are also removed. Depending on coding style, tokens are split using camel or snake case splitting. For example, in camel case splitting, "getString" token is split into "get" token and "string" token. The resulting tokens are then treated like words in bug reports' summaries and descriptions.

For the ranking based on similarities between an input bug report and source code files, BL uses a revised Vector Space Model (rVSM). rVSM revises the classical VSM by introducing a function that accounts for the source code file size, ensuring that large source code files are not poorly represented [8]. It also gives higher scores to large source code files (i.e., as compared to classical VSM) due to their higher probability of containing bugs [9], [10], [11]. rVSM similarity score $rVSMScore(b, f)$ between bug report b and source code file f is defined as follows:

$$rVSMScore(b, f) = \frac{cos(b, f)}{1 + e^{-N(\#terms)}} \quad (1)$$

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TSE.2021.3075215, IEEE Transactions on Software Engineering

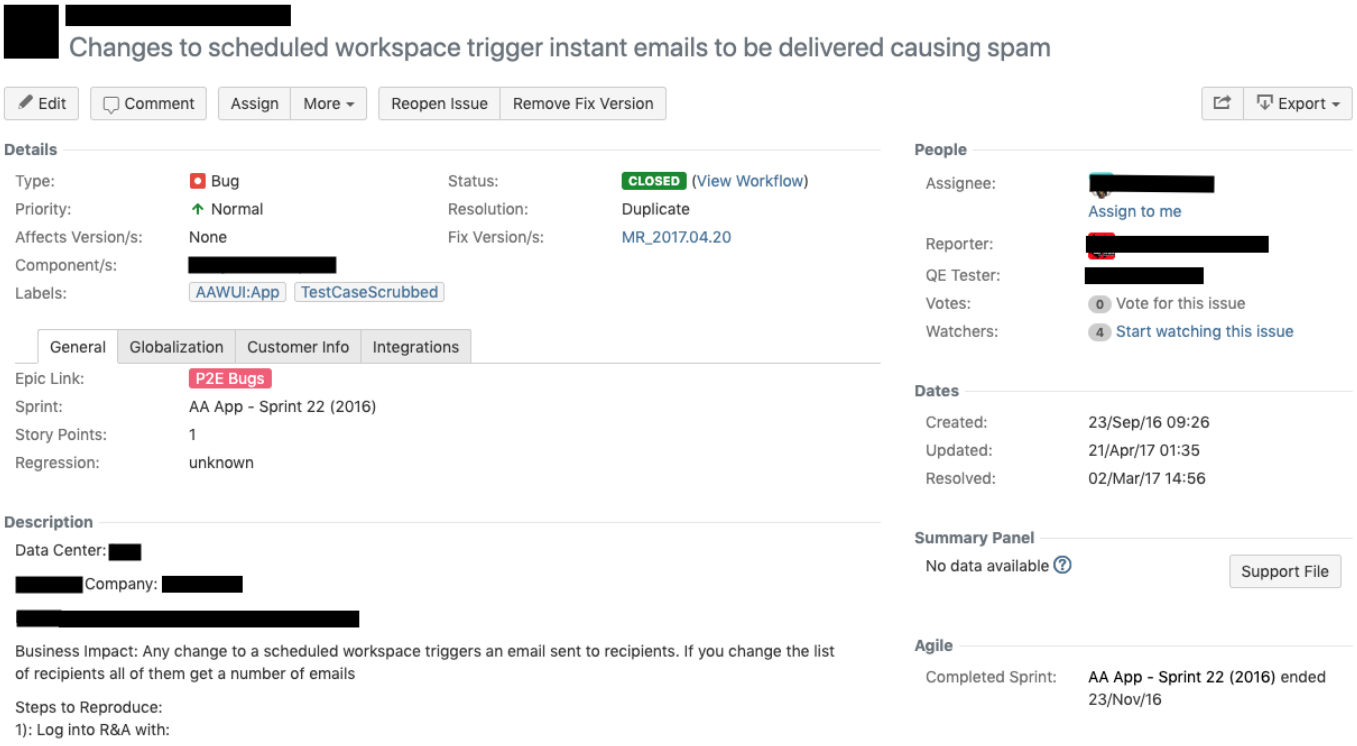

Fig. 1. An example of a bug report in Adobe Analytics (redacted due to sensitive information)

where $cos(b, f)$ is a classical VSM cosine similarity score between b and f, $\#terms$ is the total number of terms in f, and $N(\#terms)$ is the normalized value of $\#terms$.

For the ranking based on similar bug reports, BL creates a bug-file graph that links a new bug report to its similar past bug reports that are in turn linked to their relevant source code files (i.e., files that were modified to fix the bug reports). This graph is used to compute a similarity score between the new bug report and a source code file according to localized past bug reports. The similarity score $SimiScore(b, f)$ between bug report b and source code file f is defined as follows:

$$SimiScore(b, f) = \sum_{pb \in P(f)} \frac{cos(b, pb)}{n(pb)} \quad (2)$$

where $P(f)$ is a set of past bug reports that have been localized to source code file f, $cos(b, pb)$ is a classical VSM cosine similarity score between b and pb, and $n(pb)$ is the number of files that were modified to fix the bug reported in pb.

BL then combines the above two scores into a final score as follows:

$$FinalScore(b, f) = \alpha \times rVSMScore(b, f) + (1 - \alpha) \times SimiScore(b, f) \quad (3)$$

where α is a weighting factor and $0 \leq \alpha \leq 1$. The above combined score represents the estimated relevancy between bug report b and source code file f. This combined score is used to rank the source code files in order of their likelihood to be buggy.

3 Applying BugLocator (BL) to Adobe Analytics Repositories

In this section, we describe experiments that answer the following research question:

RQ1: *How effective is BugLocator (BL) on Adobe Analytics repositories?*

To answer this question, we ran BL on a dataset of historical bug reports in Adobe Analytics repositories. We then measure and report the accuracy of BL. We describe this dataset in Section 3.1, and our experiment settings, evaluation metrics, and results in Section 3.2.

3.1 Datasets

For this study, seven of the most functionally important repositories in the Adobe Analytics product were used. These repositories ranged in function across user interface components, micro services and backend collection and processing projects. The source code languages included C++, Java, JavaScript, and PHP. Only source code files are collected — config, build, and test files and folders are excluded. Six of the seven repositories had a total number of source code files ranging from 300 to 700. One repository had an order of magnitude more than the others ranging from 2,500 to 5,000 files. Figure 2 shows the summary of the dataset statistics.

Adobe Analytics uses Jira for issue tracking and Git for source code management. Adobe Analytics has no formal process mandating how the Jira and Git repositories are linked. Each development team has the freedom to establish the process that works best for them. For the purposes of this study, we needed to link these two data sources. To

establish this link, we used the following heuristic, which is similar to the procedure followed by Bissyandé et al. [12]. For every Git repository, all commit messages for every file in the history of the repository were searched for text patterns matching Jira issue IDs that represented issues of type bug. When a Jira issue ID was found, information was collected from both Git and Jira. For Jira, the following fields were collected: bug summary, bug description, bug ID, bug reporter, bug report creation date, customer-reported flag (true if customer-reported, false if internally reported), and all available comments, including the timestamp reflecting when the comment was added. This information, along with the linked Git commit message and commit ID, were collected.

When dealing with customer-reported bugs, within Adobe Analytics, it is standard practice for the repositories we examined to include the Jira issue ID in the commit message in order to link the commits to the issue. This is not strictly enforced, however, and therefore we were unable to link some Jira issues with their accompanying commits. Often an issue that is initially reported as a bug is judged by the developer to be a feature request or working as designed. In these cases there were no Git commits linked to the issue, and therefore they were excluded from our data as well.

Figure 2 shows the basic descriptive statistics for each of these data sources and the number of experiments for each repository. Each experiment represents an application of BL to localize buggy source code files for a bug report. For these bug reports, we have the ground truth (i.e., files that were modified to fix the corresponding bug report), and can therefore evaluate the accuracy of BL.

Prior studies, e.g., [13], [14], [15], [16], [17], reported that three data-set biases are of concern when using learning to rank methods for defect/bug localization. These include: (1) misclassified bugs, (2) using bloated oracles and (3) the presence of code snippets in bug reports. An additional general concern in machine learning is target leakage. Target leakage occurs when information for a given experiment includes data that was not available at the time of the initial bug report. This can happen if commits, comments, etc, that have been added after the bug report date, are present in the training data for any experiment.

The main impetus for adoption of a bug localization tool at Adobe Analytics is to obtain improvements in bug fixing accuracy and rates. Customer-reported issues should have the highest potential benefit from these improved rates. They are of greater value to Adobe than internal issues, as these are cases that are directly impacting customers, and are therefore likely to affect customer retention and up-sell prospects in the future. Developers understand that these bugs take the highest priority and work to resolve these as quickly as possible. Any tool that can aid in this process even slightly presents a high value to the company. As such, this study limits defects to customer-reported issues only. To the best of our knowledge, no previous defect localization study has been carried out using only customer issues. Using only customer-reported issues also results in the reduction of the previously stated biases. This reduction is possible due to differences in reporting and tracking of internal and customer-reported issues.

Important differences exist between customer-reported and internally reported issues at Adobe Analytics. Internal issues differ from customer-reported issues by being less formal in reporting format and therefore have a wider range of included details — being either more or less verbose depending on developer habits. Internal bug reports usually have some indication of where the issue might be addressed in comments, descriptions, or steps to reproduce. Also, they frequently include actual code and code related information such as debug traces or source file names. Additionally, internal issues are more likely to be used broadly and be linked to commits with code changes for a variety of issues, related or not. Issue type and resolution status are also less frequently monitored for accuracy. These features make internally reported issues prone to the biases mentioned above.

For customer-reported issues, Adobe Analytics implements several formal procedures. Some of these procedures include: standardized bug reporting formats, trained customer support representatives who create the initial reports, and issue and resolution type assignment guidelines and monitoring to accurately reflect the work performed to resolve a given customer issue. Because the customer is often informed of the steps taken to resolve an issue, adherence to these procedures are given a high priority. Since customer bugs with a resolution status of fixed are released into production on a regular basis, the potential need to roll back a faulty fix encourages developers to only include changes directly related to an issue in the source control commit. It is very bad practice to roll back fixes that are working, to change a single one that is not, as is the case for a commit that includes changes for multiple bugs. Another advantage of using customer-reported defects alone, is the avoidance of the bias of code related information in the bug report. Customer issues rarely have source code or code related words in them. Bug reports are most often reported in terms of functionality and violations of expected behaviors. These practices reduce the biases of misclassified bugs, bloated oracles and the presence of code snippets.

The issue of target leakage is addressed by carefully implementing checks during experiment data collection. For this study, timestamps are recorded for all collected corpora. By collecting timestamps, it is simple to exclude all data that occurred after the initial issue reporting date, for all text, across all experiments. This approach eliminates the issue of target leakage.

3.2 Experiment Settings, Evaluation Metrics and Results

Setting: We ran BL on customer-reported bug reports collected from each of the seven repositories. Each individual bug report corresponds to one experiment. In total, we have 933 experiments. For each pair of a customer-reported bug and its linked Git commit message, we collected the changed files for the commit. Only files that existed in the version before the commit were labeled as buggy files. Given the summary and description of each customer-reported bug report, BL was evaluated based on its ability to process these textual contents and rank the buggy files higher in its returned list of files.

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TSE.2021.3075215, IEEE Transactions on Software Engineering

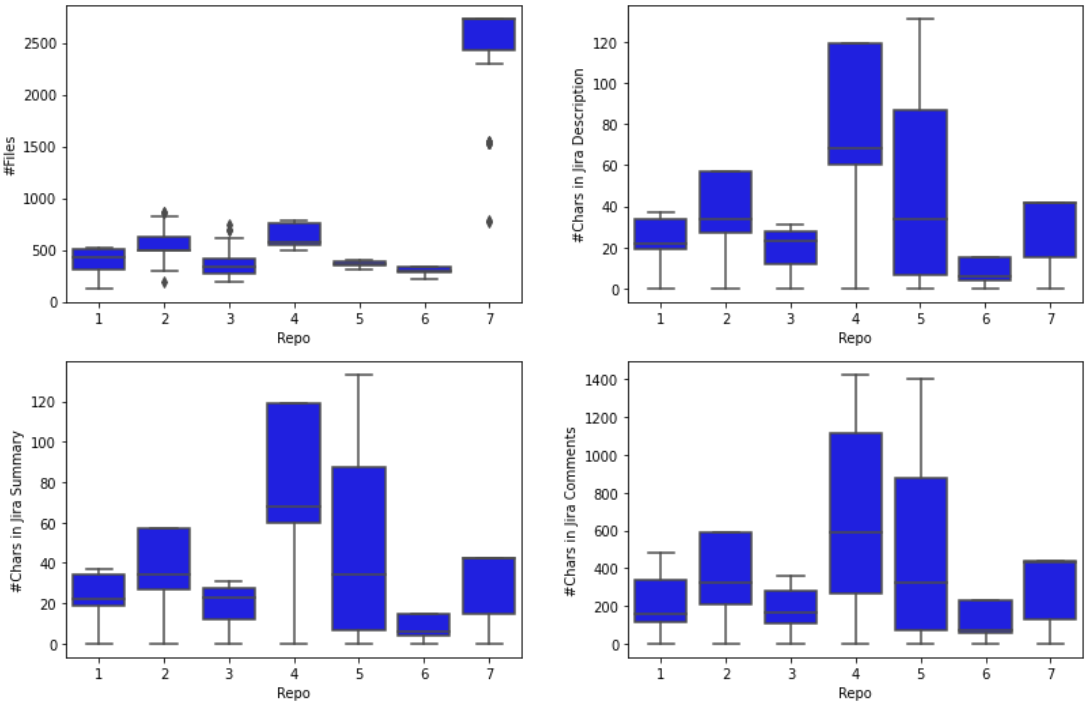

Fig. 2. Descriptive statistics of the data sources. See Table 8 in the Appendix for more details.

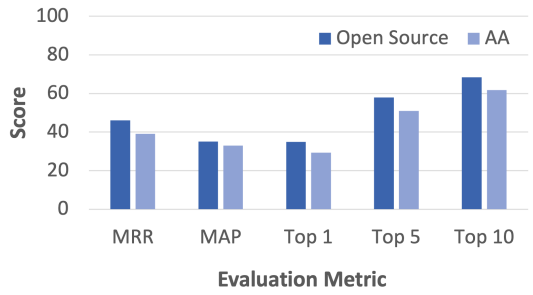

Fig. 3. Comparison of the accuracy of BugLocator on 4 open source repositories as reported in [3] vs. 7 Adobe Analytics repositories (in percentage). See Table 2 in the Appendix for more details.

Evaluation Metrics: We used several evaluation metrics that were also used by Zhou et al. [3] and many other bug localization studies in their evaluation. They include: Mean Average Precision (MAP), Mean Reciprocal Rank (MRR) and percentage of experiments where at least one buggy file is located in the top 1, 5, and 10 (i.e., Top N for N $\in \{1, 5, 10\}$). These are defined as follows:

$$AP(b) = \frac{\sum_{i=1}^{|F|} P(i) \times rel(i,b)}{\#buggy\ source\ code\ files}$$

$$MAP = \frac{\sum_{i=1}^{|B|} AP(b_i)}{|B|} \quad (4)$$

$$MRR = \frac{1}{|B|} \sum_{i=1}^{|B|} \frac{1}{rank_{b_i}} \quad (5)$$

$$Top\ N = \frac{\sum_{i=1}^{|B|} localized(b_i, N)}{|B|} \quad (6)$$

where B is the set of bug reports to localize to their linked source code files, F is the set of source code files in the repository, $P(i)$ is the proportion of buggy source code files in the top-i returned source code files, $rel(i,b)$ value is 1 if source code file at rank i is linked to bug report b (i.e., the file is one of those that need to be modified to resolve the bug report) and 0 otherwise, $rank_{b_i}$ is the rank of the first source code file that is linked to bug report b_i, and $localized(b_i, N)$ value is 1 if at least one file that is in the top-N list was present in the commit linked to the fix of bug report b_i, and 0 otherwise. For the evaluation, we define a returned file to be correct if it was present in the commit linked to the bug fix.

Results: The results for applying BL to Adobe Analytics repositories are shown in Figure 3. Comparing the results in the Open Source and AA in the figure, we can note that BL applied to the Adobe Analytics repositories performed worse than when applied to open source repositories as reported in [3]. This is due in large part to the very poor results for Adobe Analytics repository 7 and the somewhat poor results of Adobe Analytics repository 6. The results for the remaining 5 repositories are comparable to BL for open source repositories. Unfortunately, this result is insufficient in meeting the projected minimum applicability requirement for adoption of a defect localization tool by Adobe Analytics developers.

The poor accuracy for repository 7 is likely due to the large number of files that BL has to choose from (roughly 10 times more than the other repositories), which makes predicting the correct files much more difficult. On the other hand, given this reasoning, the poor accuracy on repository

6 is somewhat surprising, since it is on the smaller end in terms of number of files. Perhaps the relatively smaller number of issues evaluated ($N = 39$) may have led to the poorer evaluation metrics. These results convinced us that we needed to find ways to improve BL in order to meet our applicability requirements.

4 BugLocator Extensions

The results of RQ1 show that, for Adobe Analytics, BL is insufficient to be used by developers as a reliable debugging tool. Here, we propose two extensions to BL for improving accuracy to a chosen target applicability level, which is a Top 10 of 70% for developers new to a repository.

4.1 BL+: Extending BL with Additional Corpora

During our implementation of BL, we found that there were additional data sources in the bug reports and source management system that BL was not making use of. In our experiments with BL, we noticed that optimal parameter settings differed for each repository (see Figure 7). Therefore our first attempt to improve the accuracy of BL was to add additional configurable settings and data corpora to the set of information that BL uses. These new data corpora were easily obtainable, since we were already collecting source code and bug reports for BL. More importantly, they contain natural language descriptions associated with code, which improve BL recommendations. We incorporate these changes to BL to create BL+. It has the following 4 parameters with $2 \times 63 \times 2 \times 11 = 2{,}772$ possible configurations:

i. Pre-processing (2 options):

BL+ has a parameter that allows one to choose whether stop word removal and stemming should be performed in the pre-processing step of BL or not.

ii. rVSMScore computation (63 options):

For computing $rVSMScore(b,f)$ (see Equation 1), BL compares the summary and description of a target bug report b against content of source code file f. BL+ has a parameter that allows one to choose to compare the summary and description of the target bug report against one out of 63 possible k-combinations of the 6 following corpora ($k \in \{1, 2, 3, 4, 5, 6\}$):

1) *Source code file content.* The content of the file itself.
2) *Source code differences.* This is a collection of all changes performed to the source code file, up to the time when the target bug report is reported.
3) *Commit messages.* This is a collection of messages from commits that modified the source code file, up to the time when the target bug report is reported.
4) *Bug report summary.* Commits that modified the source code file might be linked to past bug reports that describe bugs that were fixed by modifying the file. For such bug reports, we collect their summaries.
5) *Bug report description.* Similar to *bug report summary*, we collect the descriptions of past bug reports that match the same criteria.
6) *Bug report comments.* Similar to *bug report summary*, we collect the comments of past bug reports that match the same criteria.

iii. SimiScore computation (2 options):

For computing $SimiScore(b,f)$ (see Equation 2), BL compares the summary and description of a target bug report b against the summary and description of past bug reports. BL+ has a parameter that allows one to choose whether comments of past bug reports should be considered or not. The comments for the target bug report are not considered as they are unavailable for a newly reported bug. We refer to the SimiScore that considers bug report comments as *Extended SimiScore*.

iv. FinalScore computation (11 options):

For computing $FinalScore(b,f)$ (see Equation 3), BL+ has a parameter that allows one to choose an α value among 11 possible options in the range of 0.0 to 0.3 (in steps of approximately 0.033). The step size was chosen to roughly divide the range evenly. Specifically, the values are 0.0, 0.010, 0.033, 0.066, 0.100, 0.133, 0.166, 0.200, 0.233, 0.266, 0.300.

Note that, Zhou et al. [3] found the best value of α to be between 0.2 and 0.3. We found that, when adding corpora, values less than 0.2 can be better at locating buggy files. However, values more than 0.3 were not found to be useful.

4.2 Legion: Composing BL+ Configurations

The addition of 2,772 parameter configurations for BL+ naturally led to the problem of hyper parameter calibration. For any given repository, we have no way of knowing a priori what the optimal configuration should be. Indeed, we observed that for each bug report we evaluated, the best performing configuration was different. This dilemma triggered the idea to use all 2,772 BL+ parameter configurations as an ensemble of feature extractors, and train a supervised model to predict a score given the BL+ scores as inputs. Legion composes the results produced by the various configurations of BL+ by following these steps:

i. Run all BL+ configurations:

Legion first runs all possible configurations of BL+. Note that for each parameter configuration, BL+ produces a $FinalScore$ for each source code file. Thus, overall, there are 2,772 $FinalScore$s produced.

ii. Generate stacked scores:

For each file, Legion then sums the generated $FinalScore$s for the file; we refer to this sum as the *stacked score*. One may imagine that stacked scores of files can possibly be used to re-rank the source code files. However, our initial experiments show that while this may result in some gains for files ranked poorly, it may have an undesirable effect of lowering correct highly ranked files.

iii. Learn a supervised model to rank files:

To learn how to improve rank for files ranked poorly while retaining the correct highly ranked files, Legion considers the stacked score and the 2,772 scores from BL+ as features. We refer to these as *BL+ features*. These features, along with a label indicating whether the source code file is buggy or not is considered as a classification instance. Given a set of such classification instances, Random Forest [18] is then used to learn a model that scores the features so that many buggy source code files can be ranked higher than non-buggy source code files. This model can then be used to

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TSE.2021.3075215, IEEE Transactions on Software Engineering

rank source code files for a new bug report. Given a new bug report, for each file, it outputs a probability score indicating how likely it is for the file to be buggy. The files are then ranked based on these probabilities.

5 Applying BL Extensions to Adobe Analytics Repositories

5.1 Datasets

For the application of BL+ and Legion at Adobe, we used the repositories discussed in Section 3.1. We collected the additional corpora discussed in Section 4.1 following these steps:

1) Using the bug report creation date, find the closest production release for the repository that occurred *before* the creation date. This gives us the best chance of looking at the same code that the bug was reported against.
2) Clone the repository and loop through all source code files collecting the following information for each file:
   a) The file itself.
   b) The file size.
   c) All source code differences over the entire history of the file.
   d) All commit messages in which this file was included.
   e) If a commit is linked to other bug reports in the dataset, we collect the bug reports' summaries, descriptions, and comments. The comments are combined with a timestamp indicating when the comment was added.

5.2 Research Questions and Experiment Settings

RQ2: *How effective is the inclusion of additional corpora in boosting the accuracy of BL?*

In order to determine if adding additional corpora to BL was useful, we ran BL+ with all 2,772 possible parameter configurations and measured the accuracy of each configuration. Each parameter configuration produces a different ranking for the source code files. We investigated which parameter configuration produces the best accuracy for each evaluation metric defined in Section 3.2. We report the best accuracy for BL+. This is the upper bound on the accuracy of BL+ assuming we know in advance which configuration setting performs the best. In practice however, we have no way to know which configuration will be the best performer. This dilemma triggered us to turn to the Random Forest model in Legion to overcome this limitation.

RQ3: *How effective is Random Forest when applied to compose BL+ features?*

For Legion, we obtained scores for each file in each experiment using the following cross-validation method: (1) The experiments were divided into 10 groups. (2) We held out the data from one group and trained a Random Forest classifier on the data from the other 9 groups. (3) We scored the data from the hold-out group using the trained Random Forest model. We repeated steps (2) and (3) 9 more times using a different group as the hold-out set, training a new Random Forest classifier each time, until we had scores for every file in every experiment. This cross-validation approach was necessary for evaluation of our historical data set. In practice, we train the Random Forest only once on the entire data set, and use the trained model for new bug reports as they arrive.

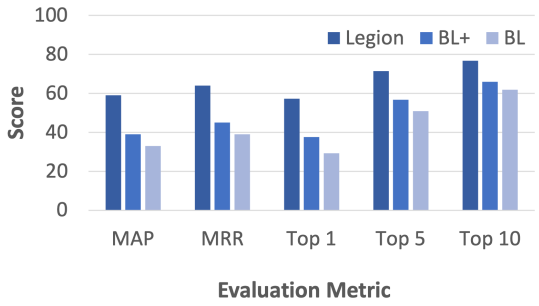

Fig. 4. Average results for all repositories (in percentage). See Table 3 in the Appendix for more details

RQ4: *How efficient are BL+ and Legion?*

In some situations, the efficiency of a bug localization tool might be of high importance. In order to address the issue of how long it takes to get results from each of the approaches, timing information was tracked across all repositories and experiments.

5.3 Results

Figures 4, 5 and 6 summarize the results. Figure 6 shows Top 1, 5 and 10 results along with MAP and MRR for each repository. In all cases, BL+ and/or Legion showed substantial improvements over BL.

RQ2: As shown in Figure 4, BL+ improved over BL across all evaluation metrics. For Top 1, 5, and 10, BL+ achieved an average score of 37.5%, 56.7% and 66.0%, respectively. This is an improvement over BL by 29.0%, 14.1% and 8.7% in terms of Top 1, 5, and 10, respectively (see Figure 5 for details).

RQ3: As shown in Figure 4, Legion achieved an average Top 1, 5, and 10 scores of 57.2%, 71.5% and 76.8% across all repositories, respectively. This is an improvement over BL by 143.4%, 58.1% and 36.4% in terms of Top 1, 5 and 10, respectively (see Figure 5 for details).

RQ4: The time it takes to complete an experiment (i.e., an application of a bug localization approach to localize a bug report) depends on the number of files in a repository and the size of the corpora. On average, 95% of the experiment time is spent collecting and pre-processing the corpora into vector form in VSM. Excluding repository 7, which is very large in terms of corpora size, the average time to run BL+ over all 2,772 parameter configurations is 34.8 minutes. Legion runtime for these 6 repositories took an average of 15.12 minutes. This includes both training and testing of the Random Forests. Experiments were carried out on machines having 4-12 cores and 8-32GB memory. Timing information per machine was not collected.

Concerns about efficiency of Legion can be divided into two kinds: (1) efficiency in training the model, and (2) efficiency in running the trained model. Efficiency of type (1) is not considered highly important for the application

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TSE.2021.3075215, IEEE Transactions on Software Engineering

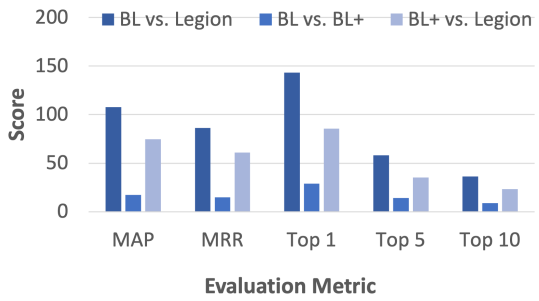

Fig. 5. Average accuracy improvement (in percentage). See Table 4 in the Appendix for more details.

of Legion at Adobe Analytics. It is possible to run automated processes to update Random Forest models periodically and have these models available for new customer-reported bugs when needed. Although this training may take hours, or even days for large repositories, this can be run in the background periodically while using the previously trained model for new incoming issues. Additionally, applying these models to incoming bug reports can also be automated. It is also possible to provide a developer with the recommended files directly within the bug report management system so that a developer does not have to run the tool themselves. When running the pre-trained model on a new issue (type 2 efficiency), performance has to be better than the current time required to get a customer-reported bug into the hands of the correct developer. This is typically on the order of days. This provides ample time to create a Legion recommended files list even for the largest dataset. Therefore, although the runtime for a new issue may take up to an hour, this level of efficiency is suitable for our requirements.

6 Additional Findings

BL is a strong research tool for recommending files to be fixed given a bug report. When applying BL at Adobe Analytics, results across 5 of 7 repositories found good results similar to those reported in [3]. However, for adoption within Adobe Analytics, these results are not sufficient to pass BL as a reliable debugging tool. BL failed to meet our minimum applicability requirements for both developers new to a code base and developers having extensive experience with the code base. We have shown that, by adding the following steps, BL can be improved to the required level of accuracy and dependability:

1) Adding additional corpora (BL+).
2) Composing BL+ features using a Random Forest (Legion).

To further analyze the improvement of Legion over BL and evaluate its ability in real world usage, we asked the following research questions:

RQ5: *Are all BL+ features needed?*

In order to achieve the minimum applicability requirement, we use many BL+ features to improve the accuracy of BL. We want to dig deeper to understand if all these features are needed. We answer this RQ in Section 6.1.

RQ6: *How do Adobe developers perceive Legion's recommendations?*

We have shown that Legion achieves the minimum applicability requirement that Adobe developers expected to have. Here, we would like to get inputs from Adobe developers on Legion's outputs for recent customer-reported issues. We answer this RQ in Section 6.2.

6.1 Are all BL+ features needed?

It is not possible to know ahead of time which BL+ features will do well in recommending buggy files for a given bug report. There might be large variations in both quality and quantity of the corpora. BL+ and Legion attempt to solve this problem by exploring the ranking of source code files in all parameter configurations. We hypothesize that some configurations can recommend files that others might miss. In order to maximize good recommendations, we use Random Forest to weight features in a way that retains well ranked files from all configurations. Yet, is this what actually occurs in practice? To answer this question, we investigate two dimensions: observation and correlation.

Observation. After running BL+ with all possible parameter configurations and computing the accuracy of each configuration, we take a look at the best performing parameter configuration that results in the highest score for each pair of repository and evaluation metric. Figure 7 shows the parameter that contributes to the best performance for each pair of repository and evaluation metric. Most parameter values, except α, appear in at least one of the best performing parameter configurations. For α, only 8 of the 11 possible values are in such configurations. Interestingly, α values in the best performing configurations are ≤ 0.2. This is slightly different from what Zhou et al. [3] found. They found that α values between 0.2 and 0.3 worked best. These values were occasionally in the top 10 best performing parameter configurations (8 times), but never in the best performing one. If a given parameter was of little value, we presume that it would not be present in any best performing parameter configuration. Since every parameter does occur, we consider that all parameters play a part and are, therefore, necessary.

Correlation. Although all parameters contribute unique information to the problem, we expected that scores resulting from similar parameter configurations would be highly correlated. We calculated the pairwise correlation between the source code file scores from the 2,772 parameter configurations using Pearson and Spearman correlation, and Principal Component Analysis (PCA). As expected, most parameter configurations resulted in highly correlated scores. For example, Figure 6 shows the percentage of Pearson correlation coefficients for each pairwise comparison of scores in repository 6. Over 61% of Pearson's r coefficients were greater than 0.8, and more than 26% greater than 0.9. For PCA, the first component explained 81.9% of variance in the scores, followed by 7.8% and 2.6% of variance for the second and third components.

This high multicollinearity could present problems for combining the scores using logistic regression. For this reason, we chose to use a Random Forest, which is more robust to collinear data [19]. Other algorithms could be

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TSE.2021.3075215, IEEE Transactions on Software Engineering

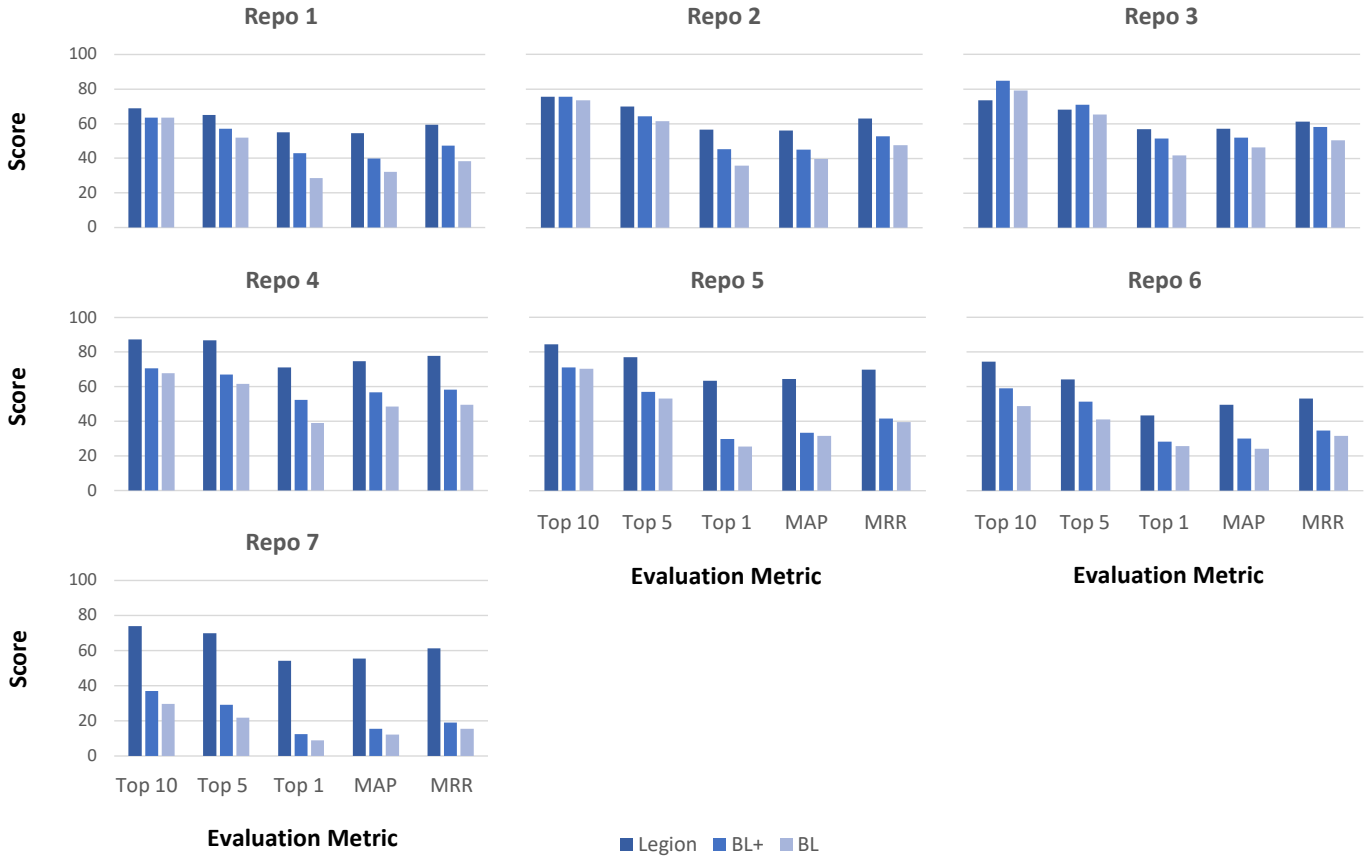

Fig. 6. Breakdown of results by repository (in percentage). See Table 6 in the Appendix for more details.

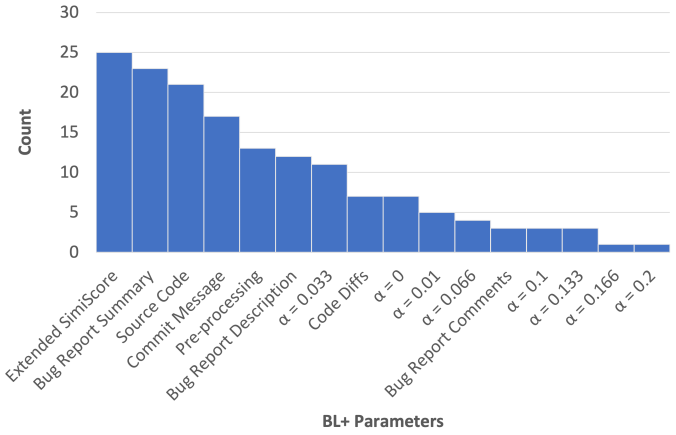

Fig. 7. Distribution of BL+ parameters that contributes to the best performance (out of 35 possible pairs of evaluation metric and repository). See Table 7 in the Appendix for more details.

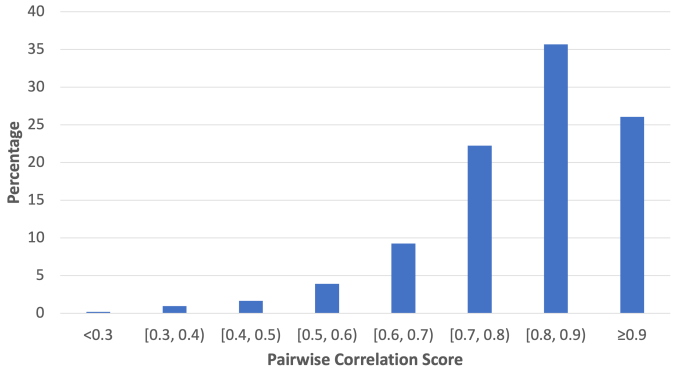

Fig. 8. Percentage of pairwise correlation for 2,772 BL+ parameter configurations in repository 6. See Table 6 in the Appendix for more details.

used as well. The high multicollinearity is also reflected in feature importance scores from the Random Forest model. We calculated these scores using the Mean Decrease in Gini Impurity, as part of the Scikit-learn Python library [20]. The scores decrease slowly, starting from 0.0073 for the top ranked feature to 0.0022 for rank 50. This shows that there is no single BL+ parameter configuration that dominates the Random Forest scores. 2 configurations in the top 5 remove stop words and use stemming. Only 2 use the source code. All 5 use commit messages and bug report descriptions. 4 use the bug report summary. Only the top ranked configuration uses the extended *SimiScore*. Values for α range from 0.033 to 0.2. Source code differences and bug report comments do not appear in the top 5 configurations.

6.2 How Do Developers Perceive Legion's Outputs?

We conducted a pilot study with a team at Adobe Analytics. Over a 9 month period, this team received 11 customer-reported issues that they subsequently fixed. We evaluated these issues using Legion, which produced a Top-10 list of files likely to be changed for each issue. Of the 11 issues, we correctly predicted a changed file in the Top-10 in 7 of the

issues. Then, we conducted a pilot study where we asked the developer who fixed the issue (each issue was fixed by one developer) to evaluate effectiveness of the predictions. Consequently, study participants are limited to 3 senior developers, each of whom fixed some of the 11 issues.

Note that our pilot study is not meant to produce generalizable knowledge as the study is only with 3 individuals, each providing inputs on recommendations that Legion provides for just several issue reports. To get generalizable and conclusive results, a full-fledged user study over more developers and issues is needed, which we reserve for future work. Our goal here is to get initial responses and insights on what can be done and what factors to consider in such future study.

In our pilot study, we sent the 3 developers an email with the following text:

"We are developing a tool that attempts to locate files containing defects given the JIRA description in the Customer Reported Issue. We would like to get your feedback on the potential usefulness of this tool for locating defects more quickly. To this end, we have identified several issues that occurred over the last 9 months that you fixed. We have included the list of files that our tool identified as potentially containing defects at the time of the reported issue, along with the files that were changed in commits tagged with the JIRA issue. Given these results, please answer the following questions concerning the tool."

We asked them to provide the following inputs:

- I1 *On a scale of 1 - 5 where 1 is extremely unhelpful and 5 is extremely helpful, how useful could this be in debugging?*
- I2 *On a scale of 1 - 5 where 1 is extremely ridiculous and 5 is extremely useful, rate the recommendations.*
- I3 *On a scale of 1 - 5 where 1 is extremely obvious and 5 is extremely surprising, rate the correct recommendations.*
- I4 *One a scale of 1 - 5 where 1 is extremely junior and 5 is extremely experienced, what experience level would find these recommendations helpful?*
- I5 *Please tell us what could make this tool more useful, or what other kinds of things you would like to see.*

TABLE 1
Developer responses to questions I1-I4

	Dev 1	Dev 2	Dev 3	Avg
I1 Unhelpful vs. Helpful	2	5	3	3.33
I2 Ridiculous vs. Useful	2	4	2.5	2.83
I3 Obvious vs. Surprising	3	2	2	2.33
I4 Junior vs. Senior	2	3	3	2.67

Note that we collect their responses via emails, and thus for I1-I4 developers may also provide some short texts that explain their rationales.

The survey results are shown in Table 1. For I1 and I2, considering an average rating higher than 2.5 as positive, and below it as negative, the senior developers responses were positive. Developer 1 was negative, Developer 2 was very positive, and Developer 3 was marginally positive. Developer 1 was only given Legion's recommendations for two issues that he fixed and Legion was unable to include the buggy file in its top-10 recommendation for one of them; the developer provides the following reason for his low rating for I1 and I2: "it didn't have correct file in 1 instance". Developers 2 and 3 were happy about the helpfulness of the tool (I1), with Developer 2 stating, "Our codebase is very large and complex, and despite 4.5 years of experience on it I frequently struggle to find the appropriate location for a change." However, they were affected by incorrect recommendations that Legion provided (I2), as Legion was able to uncover correct files in the top-10 for only 7 out of the 11 issues. All of our participants are senior developers, and our earlier informal discussion with developers (see Section 1) uncovers that for developers familiar with a repository, a bug localization tool should identify a buggy file in the top-5 recommendations at least 80% of the time.

For I3, the senior developers perceived the recommendations to be more obvious to them than surprising – the ratings received are 3, 2, and 2, with an average of 2.33. Our study is a retrospective one though – it was performed after the issues were fixed – and the perception of developers, especially on the surprisingness of the recommendations, may differ if the study was performed prior to the issues being fixed (see Section 7.3). For I4, the senior developers perceived that the tool will benefit people who are junior or mid-level developers, rather than those who are extremely experienced – the ratings received are 2, 3, and 3, with an average of 2.67.

Feedback from I5 provides additional inputs to improve Legion further. Developer 1 mentioned: "Out of 2 bugs this tool didn't have a correct file in 1 instance. Does this tool use only description or also xml? (I think it has better chance finding correct source using xml)". This highlights the potential of further expanding the input to Legion to also consider contents of related XML and configuration files. Developer 2 mentioned: "The results are ranked, but I'm assuming they are sorted according to some kind of weighting. I think it could be useful to see that weighting if 1-3 are all functionally the same, I would react to that differently than if #1 had 10x the weight of #2." This highlights the value of including the raw scores from the Legion tool to the list. Interestingly, a similar heuristic was employed by Le et al. [21] to create a trust score that can be used by developers to assess recommendations given by a bug localization tool. Developer 3 mentioned: "Pi in the sky: Automatic reducing of test cases to minimal ones". This highlights the value of combining bug localization with testing. A recent study has demonstrated the value of combining information contained in bug reports and failed test cases for more effective localization of bugs [22], [23].

Overall, these results show that senior developers find the recommendations provided by the tool to be moderately helpful. It stands to reason that developers who are less experienced with the codebase or people new to the team stand to benefit more. The team that we invited for the pilot study only received a low number of issues (11 issues) in a relatively long period of time (9 months); the benefit of a bug localization tool is likely to be more for teams who received a high number of issues to resolve daily. Still, from the survey, it is clear that there is room to improve upon these results, possibly by incorporating other advances that have been proposed in other bug localization studies that have not been included in Legion – for example, the incorporation of trust score [21], [24], the integration of additional sources of information [25], [26], [27], [22], [23], usage of more

advanced machine learning (deep learning) approaches [6], [28], [29], design of intuitive and interactive interfaces, etc. (c.f., Section 8.1).

7 Discussion

7.1 Reflection

In this study, our main goal is to experiment with a bug localization approach, applying it to industrial data, and improving its accuracy to an acceptable level, based on requirements identified by developers. This goal is both important and novel; all previous studies on bug localization only make use of open-source data. Can a bug localization tool perform well on industrial data? Our study has succeeded in answering this question; we found that relatively high accuracy scores reported in the literature can also be achieved for industrial data, and developer requirements can be partially met. Still, we are unable to claim generalizability of the findings as all our data comes from only one company. Thus, we encourage further studies to perform a replication of our work on data from other companies. Although such studies are not easy to perform due to the closed nature of industrial data, different interests between researchers and developers, and the changing priorities of a company (indeed, a reorganization within Adobe prevents us from doing more at this point in time), we believe they are needed to push bug localization tools to go from research prototypes to widely-adopted industrial tools.

Also, our study finds that the requirement for junior developers ("the tool should identify a buggy file in the top 10 recommendations at least 70% of the time) cannot be met by BugLocator, but it can be met by Legion that extends it. However, Legion is still unable to fully meet the requirement for senior developers ("the tool should identify a buggy file in the top 5 recommendations at least 80% of the time). This is partly corroborated by findings from our pilot study that peeks into how several senior developers respond to Legion's recommendations. We find that the developer ratings are very much related to the accuracy of Legion. For Developer 1, 2, and 3, Legion identified a buggy file in the top 10 recommendations 50%, 100% and 62.5% of the time respectively. We find that their overall ratings of Legion (for I1) are correlated to these accuracy scores; the ratings were 2, 5, and 3 respectively. Developer 1 was not satisfied, Developer 2 was very satisfied, while Developer 3 was marginally satisfied. This suggests the value of research effort on boosting the accuracy of bug localization tools; while a small boost in accuracy may not matter much, a substantial (e.g., $\geq$ 10%) boost in accuracy can result in a noticeable increase in developer satisfaction (in our pilot study, 12.5% and 37.5% boost in accuracy increased overall rating by 1 and 2 points, respectively).

Legion incorporates new features that were not investigated in prior studies (see Section 8.1). Still, many features that were proposed in the literature have not been incorporated in Legion. Indeed, Legion is based on a classic bug localization approach, named BugLocator [3], which was published close to a decade ago. It is possible that the incorporation of other features can boost Legion's accuracy further, up to a level that is deemed acceptable by senior developers. We leave the exploration of this possibility to future work.

Findings from our pilot study pose a question as to who should be the target users of a bug localization tool. Should they be junior developers? Can senior developers benefit too? The participants seem to suggest that Legion is more beneficial to junior and mid-level developers (their ratings were either 2 or 3 out of 5, with 5 corresponding to "extremely experienced). Still, we do not have a conclusive answer. Our findings are based on a pilot study involving a team that received few issue reports. The findings may differ if we consider teams that are inundated with issue reports. Future industrial studies on a larger pool of developers, including both junior and senior developers, working in teams that receive varying levels of bug reports are needed to fully answer this question.

Lastly, we would like to comment that a company-wide deployment requires more than part-time effort from a few developers keen on adopting state-of-the-art research tools, but rather long-term and continuous support from a dedicated team within a company. The team needs to develop a user-friendly interface that is integrated into developer workflow, actively promote the tool among developers, manage developer expectations [30], and provide continuous maintenance support [31]. Although we are unable to do this at this point in time, we hope that this can be realized in the future, either at Adobe or beyond.

7.2 Lessons Learned

Relatively high accuracy scores can be realized for industrial datasets and customer-reported issues: Prior works, e.g., [32], [33], [25], [3], [4], [5], use open source datasets consisting of mainly developer-reported issues due to their availability. This is the first work that makes use of industrial datasets consisting of customer-reported issues. Prior to this work, the accuracy of bug localization on industrial datasets and customer-reported issues were unknown. It is possible that the accuracy of bug localization on industrial datasets and customer-reported issues are very different than the ones reported in existing works. Through this study, we demonstrate that it is not the case; Legion can achieve an average Top-10 score of 76.8% across all repositories, and an average Top-5 score of 71.5%.

Developers, especially highly experienced ones, have high expectations: Our study, through informal water cooler discussions and a pilot study, highlights that developers have high expectations for adoption of bug localization tools. The thresholds for adoption differ depending on the seniority and experience level of developers. Senior developers have higher expectations as they are very familiar with the codebase that they have worked on for years.

One interesting direction for future work is to design a solution that can elicit domain knowledge from senior developers that can be embedded into a bug localization tool that can in turn be used to aid less experienced developers or newcomers to a development team. Another possible direction is to build a personalized recommendation system that considers a developer's prior experience (e.g., only output recommendations when a potentially buggy file falls

outside the files that developers normally maintain). Yet, another possibility is to incorporate solution designed by Le et al. that can output "trust" scores to bug localization outputs [21]; "unsure" bug localization recommendations can be marked as such and developers can focus on high-confidence recommendations. Bug localization tools can then focus on satisfying developers' high expectations for those high-confidence cases.

Additional data sources should be used to boost bug localization performance: Our BL+ considers various additional corpora that were not considered in prior work. The corpora were easy to collect, as they consist of artifacts inherently linked to the bug reports and source commits (e.g. commit messages and bug report comments). These corpora strengthen the associations between source code files and natural language bug reports which allow the BL algorithm to find better matches. Our findings augment the existing body of work that has used *other kinds* of additional pieces of information to boost bug localization performance—see Section 8.1. Some of these additional pieces of information may be harder to obtain or less available than the others (e.g., execution traces, etc.).

An interesting direction of future work is to combine all the additional data sources considered in the literature to boost performance of Legion further. Another interesting direction is to compare the efficacy and availability of these data sources for bug localization purposes considering various contexts (e.g., open source vs. industrial, etc.).

Tuning matters for bug localization: Each bug report is different from many other bug reports. Each repository is different from many other repositories. Thus, an adaptive approach that can be tuned for various repositories and bug reports can perform better than a one-size-fits-all approach (in our case: one configuration of BL+). This is supported by the superior results that Legion achieves over the baselines. The benefit of tuning and an adaptive approach has also been reported in many prior works, e.g., [34].

In this work, we only investigate the use of Random Forest to tune a composition of rankers (in our case: different configurations of BL+) for improved bug localization. Many other approaches can potentially be used to ensemble these rankers. We encourage further research to be done in this direction.

7.3 Threats to Validity

Legion worked well within the minimum applicability requirement specified by Adobe Analytics developers. We also received positive experiences from our small pilot study involving a team of Adobe Analytics developers. However, it is unclear whether the same level of satisfaction would be observed by other teams in Adobe Analytics. That said, we believe the results should extend well to them due to the shared culture. On the other hand, it remains to be seen whether Legion would achieve the same level of applicability on other teams outside Adobe Analytics. Nevertheless, we believe our work provides valuable insights on the applicability of bug localization tools in Adobe and possibly on the industrial setting in general.

In the pilot study, there is a threat that the developers may not have answered the questions truthfully. However, we believe this threat is minimal as the developers were aware that we are developing Legion for possible adoption in Adobe. Therefore, we believe it was in their best interest to provide their honest opinions about Legion as they may possibly use it in the future. Indeed, we find that our participants were honest in providing their ratings; for example, Developer 1 found that Legion was successful in only one of the two issue reports that he had fixed, and the developer gave a rating of 2 (out of 5) for both I1 and I2. Also, our pilot study is a retrospective study; we collected developers' perception of Legion's outputs for bugs that they had fixed before. Developers' perception of Legion's outputs may have differed if we had asked the developers to provide inputs prior to them fixing the bugs. For search tasks with specific goals, prior studies, e.g., [35], reported that people's perceptions of search task difficulty *decreases* after the task is completed. Thus, the ratings that we obtained, e.g., for helpfulness of Legion's output (I1), how obvious Legion's recommendations were (I3), can be a lower bound on the ratings that developers would provide if the study were replicated prior to them fixing the bugs. Also, we did not limit or recommend what the developers should do when answering the questions. Some of them may have looked back to the task or just recalled the answers from memory. This could be a threat if the developers wrongly recalled from memory.

Another possible threat is related to the correctness of our implementation and experiments. We have checked our implementation and experiments multiple times. Thus, we believe the threat is minimal.

8 Related Work

8.1 Bug Localization Tools

Bug localization approaches can be put in two main families: IR based bug localization and spectrum based bug localization. IR based bug localization treats bug localization as an information retrieval problem, e.g., [32], [33], [25], [3], [4], [5], [36], [6], [7], [28], [37], [38]. These approaches treat a bug report as a query that is used to search program elements (e.g., files) that should be modified to fix the bug described in the bug report. Spectrum based bug localization analyzes program spectra (i.e., statistics related to program execution traces) to find program elements (e.g., files) that are likely to contain bug(s), e.g., [39], [40], [41], [42], [43], [44], [45], [46], [47], [48], [49], [50], [51], [52], [22], [53], [54], [55]. This work focuses on the first. Since bugs in Adobe Analytics repositories are usually reported by customers, IR based bug localization is a natural choice. Such bugs usually only have descriptions from customers and lack other information such as program spectra.

We describe some related prior IR based bug localization work categorized into three families based on their major contributions (e.g., use of new data analytics algorithms, consideration of additional data sources, and incorporation of additional processing steps) below. Our survey is by no means complete; for a comprehensive survey of the topic, please kindly refer to [56].

Different data analytics algorithms: Different data analytics algorithms have been employed for bug localization. For

example, Rao and Kak [33] conducted a comparative study of generic and text based models for information retrieval approaches at the time. They found that Unified Model and Vector Space Model (VSM) are more effective for bug localization than Latent Dirichlet Allocation. Thomas et al. [57] cast IR based bug localization as a classification problem, and showed that the bug localization performance is sensitive to the classifier parameters, and combining different classifiers usually helps to improve performance. Ali et al. [58] proposed LIBCROSS, an approach that utilizes Binary Class Relationship (BCR) to improve standard IR based bug localization approach such as VSM. BCR includes class relationships such as inheritance, association, and aggregation and each of them has a "vote" for the relevancy between a bug report and a source code file. Zhou et al. [3] proposed BugLocator, an approach that uses a revised Vector Space Model (rVSM) and bug report similarity to localize bugs. Saha et al. [4] proposed a structured retrieval for bug localization. This retrieval considers the structure of bug reports and source code files. Ye et al. [6] proposed a bug localization approach that uses surface lexical similarity, API-enriched lexical similarity, collaborative filtering, class name similarity, bug fix recency, and bug fix frequency as features that are fed to a learning-to-rank algorithm. Lam et al. [7], [28] proposed an approach that improves bug localization performance using a combination of Deep Neural Network (DNN) and rVSM. DNN bridges lexical gaps between bug reports and source code files by learning the relation between terms in bug reports with terms in source code files. Xiao et al. [29] used a Convolutional Neural Network for bug localization, together with bug fix recency and frequency, bug report embedding, and source code file embedding as features.

Different sources of information: Different sources of information have been leveraged to boost the performance of bug localization. For example, Sisman and Kak [25] estimated the bug proneness of a source code file, along with time decay, to improve bug localization. Wang et al. [26] used stack trace information included in a bug report to improve the performance of IR based bug localization. They defined crash correlation groups and used them as features for a Bayesian Belief Network that learns to rank source code files in order of their likelihood of being buggy. Similarly, Moreno et al. [59] calculated similarities between program elements in the stack trace with program elements in the source code. Rath et al. [27] utilized the project's requirements and trace links for bug localization. Le et al. [22] proposed an Adaptive Multi-modal bug Localization (AML) approach that combines IR based bug localization and spectrum based bug localization. They also introduced a concept of suspicious words, which are words that are associated with a bug.

Additional processing steps: Different processing steps have been added to the IR-based bug localization pipeline to improve performance. For example, Wong et al. [60] boosted existing bug localization performance by segmenting source code files and analyzing file position in the stack trace. Sisman and Kak [61] reformulated bug report queries using terms that consistently appear in the response of the original queries. Chapparo et al. [38] improved bug localization performance by reformulating low-quality bug report queries to only the terms that described the observed behaviour. Rahman and Roy [62] reformulated bug report queries by first categorizing the bug report and then performing graph based term weighting according to its category. Terms with the highest weights are added to the bug report query. Chapparo et al. [63] evaluated bug report query reformulation strategies that selectively pick certain parts of a bug report. They found that using the bug report's title along with the observed behaviour entry gives the best performance among different bug localization techniques and granularities (e.g., files, methods).

Comparison with this study: In this work, BugLocator is used as the bug localization approach of choice since it is a classic IR-based approach that is easy to re-implement and extend. Comparison of bug localization approaches is beyond the scope of this work. Such an effort is costly and requires tremendous care to ensure all the reimplementations are correct. Additionally, the extensions that we propose here on top of BL can possibly be used for other bug localization tools. Moreover, to the best of our knowledge, as also evident from the descriptions of the long list of related work that have we reviewed above, the extensions that we propose here—BL+ adds additional corpora (source code diffs, commit messages, bug report comments, etc.) while Legion composes different configurations of BL+ using Random Forest—have not been considered by any prior work.

8.2 Usability and Bias Studies

There have also been studies that investigate the usefulness of bug localization tools, e.g., [64], [65] as well as bias in the evaluation of such tools [17]. Both Wang et al. [64] and Xia et al. [65] investigate the usability of bug localization tools via a user study; they highlighted both the benefits and limitations of the tools. The user studies however were not performed on bug reports from industrial projects, and are not necessarily customer-reported bugs. Kochhar et al. highlighted that consideration of developer-reported bugs may cause a significant bias in the evaluation of the tools [17]; those bug reports might already explicitly specify the buggy program files and for these reports bug localization tools are not needed. Our study is inspired by the aforementioned studies and extends them by the consideration of industrial data, and customer-reported bug reports. Also, while previous studies consider participants who are not developers of the respective buggy projects (they are mostly graduate students), our study investigates the perspectives of industrial developers who fixed the buggy code.

9 Conclusion and Future Work

BugLocator (BL) proposed by Zhou et al. [3] is a well-known and effective tool and has a good track record in the research community for localizing buggy files that need to be fixed given a bug report. This study is an attempt to show that BL can be useful in an industrial setting. Replication of Zhou et al.'s work showed that, across 5 of 7 highly important repositories in Adobe Analytics, BL performed as well as the results reported for open source repositories. However, for a bug localization tool to gain adoption, higher accuracy

seems necessary. Based on inputs from Adobe developers, two different target accuracy scores were selected as the most likely needed for developer adoption within Adobe Analytics. These minimums are: (1) for a developer new to a repository, a Top 10 score of 70% must be reached for the tool to be considered useful and; (2) for a developer experienced with the repository, a minimum Top 5 score of 80% is required. BL failed to meet these requirements.

However, this study went beyond replication and found that the minimum Adobe Analytics requirements could be met by extending BL. By adding additional corpora to BL, buggy file localization accuracy is improved. By further applying a re-scoring approach using Random Forest, an average Top 10 score of 76.8% across all repositories is achieved. Therefore, the minimum requirement for a developer new to a repository has been met. Legion obtained an average Top 5 score of 71.5%. While this result is below the requirement for experienced developers, one repository did reach this standard with Top 5 score of 86.7%, and another came very close with Top 5 score of 77.1%. This suggests that further improvements could potentially achieve this goal.

In the future, we plan to evaluate BL, BL+, and Legion on more repositories and bug reports from Adobe Analytics and beyond (e.g., open-source projects). It will also be interesting to also investigate internal reports more carefully (in terms of what support developers need, how to clean the dataset from biases, and what additional sources of information, beyond what we have considered in Legion, could potentially be useful). We also plan to evaluate the effectiveness of other bug localization approaches on Adobe Analytics repositories. Also, we plan to improve Legion further, e.g., by finding correlating BL+ configurations (not just single parameters) and trying to improve results by removing them, by considering inputs given by our pilot study participants, by incorporating other advances proposed in prior bug localization studies, etc. Moreover, we encourage future studies to extend our limited pilot study with developers; the extended study can include more bug reports, more developers, and inclusion of junior developers who are likely to benefit more from automated bug localization tools.

REFERENCES

[1] B. Beizer, *Software testing techniques*. Dreamtech Press, 2003.

[2] M. Newman, "Software errors cost us economy $59.5 billion annually," *NIST Assesses Technical Needs of Industry to Improve Software-Testing*, 2002.

[3] 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 *2012 34th International Conference on Software Engineering (ICSE)*. IEEE, 2012, pp. 14–24.

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

[5] S. Wang and D. Lo, "Version history, similar report, and structure: Putting them together for improved bug localization," in *Proceedings of the 22nd International Conference on Program Comprehension*. ACM, 2014, pp. 53–63.

[6] X. Ye, R. Bunescu, and C. Liu, "Learning to rank relevant files for bug reports using domain knowledge," in *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*. ACM, 2014, pp. 689–699.

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

[8] E. Garcia, "Description, advantages and limitations of the classic vector space model," http://www.miislita.com/term-vector/term-vector-3.html, 2006.

[9] N. E. Fenton and N. Ohlsson, "Quantitative analysis of faults and failures in a complex software system," *IEEE Transactions on Software engineering*, vol. 26, no. 8, pp. 797–814, 2000.

[10] T. J. Ostrand, E. J. Weyuker, and R. M. Bell, "Predicting the location and number of faults in large software systems," *IEEE Transactions on Software Engineering*, vol. 31, no. 4, pp. 340–355, 2005.

[11] H. Zhang, "An investigation of the relationships between lines of code and defects," in *IEEE International Conference on Software Maintenance*. IEEE, 2009, pp. 274–283.

[12] T. F. Bissyandé, F. Thung, S. Wang, D. Lo, L. Jiang, and L. Réveillère, "Empirical evaluation of bug linking," in *17th European Conference on Software Maintenance and Reengineering, CSMR 2013, Genova, Italy, March 5-8, 2013*, 2013, pp. 89–98.

[13] T. Dao, L. Zhang, and N. Meng, "How does execution information help with information-retrieval based bug localization?" in *Proceedings of the 25th International Conference on Program Comprehension*. IEEE Press, 2017, pp. 241–250.

[14] K. Herzig, S. Just, and A. Zeller, "It's not a bug, it's a feature: how misclassification impacts bug prediction," in *35th International Conference on Software Engineering, ICSE '13, San Francisco, CA, USA, May 18-26, 2013*, 2013, pp. 392–401.

[15] C. Bird, A. Bachmann, E. Aune, J. Duffy, A. Bernstein, V. Filkov, and P. T. Devanbu, "Fair and balanced?: bias in bug-fix datasets," in *Proceedings of the 7th joint meeting of the European Software Engineering Conference and the ACM SIGSOFT International Symposium on Foundations of Software Engineering, 2009, Amsterdam, The Netherlands, August 24-28, 2009*, 2009, pp. 121–130.

[16] G. Antoniol, K. Ayari, M. D. Penta, F. Khomh, and Y. Guéhéneuc, "Is it a bug or an enhancement?: a text-based approach to classify change requests," in *Proceedings of the 2008 conference of the Centre for Advanced Studies on Collaborative Research, October 27-30, 2008, Richmond Hill, Ontario, Canada*, 2008, p. 23.

[17] P. S. Kochhar, Y. Tian, and D. Lo, "Potential biases in bug localization: Do they matter?" in *Proceedings of the 29th ACM/IEEE international conference on Automated software engineering*. ACM, 2014, pp. 803–814.

[18] L. Breiman, "Random forests," *Machine Learning*, vol. 45, no. 1, pp. 5–32, Oct 2001. [Online]. Available: https://doi.org/10.1023/A:1010933404324

[19] C. F. Dormann, J. Elith, S. Bacher, C. Buchmann, G. Carl, G. Carré, J. R. G. Marquéz, B. Gruber, B. Lafourcade, P. J. Leitão *et al.*, "Collinearity: a review of methods to deal with it and a simulation study evaluating their performance," *Ecography*, vol. 36, no. 1, pp. 27–46, 2013.

[20] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, J. Vanderplas, A. Passos, D. Cournapeau, M. Brucher, M. Perrot, and E. Duchesnay, "Scikit-learn: Machine learning in Python," *Journal of Machine Learning Research*, vol. 12, pp. 2825–2830, 2011.

[21] T. B. Le, F. Thung, and D. Lo, "Will this localization tool be effective for this bug? mitigating the impact of unreliability of information retrieval based bug localization tools," *Empir. Softw. Eng.*, vol. 22, no. 4, pp. 2237–2279, 2017.

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

[23] T. Hoang, R. J. Oentaryo, T. B. Le, and D. Lo, "Network-clustered multi-modal bug localization," *IEEE Trans. Software Eng.*, vol. 45, no. 10, pp. 1002–1023, 2019.

[24] M. Golagha, A. Pretschner, and L. C. Briand, "Can we predict the quality of spectrum-based fault localization?" in *13th IEEE International Conference on Software Testing, Validation and Verification, ICST 2020, Porto, Portugal, October 24-28, 2020*, 2020, pp. 4–15.

[25] B. Sisman and A. C. Kak, "Incorporating version histories in information retrieval based bug localization," in *9th IEEE Working Conference on Mining Software Repositories (MSR)*. IEEE, 2012, pp. 50–59.

[26] S. Wang, F. Khomh, and Y. Zou, "Improving bug localization using correlations in crash reports," in *Proceedings of the 10th Working Conference on Mining Software Repositories*. IEEE Press, 2013, pp. 247–256.

[27] M. Rath, D. Lo, and P. Mäder, "Analyzing requirements and traceability information to improve bug localization," in *IEEE/ACM 15th International Conference on Mining Software Repositories (MSR)*. IEEE, 2018, pp. 442–453.

[28] A. N. Lam, A. T. Nguyen, H. A. Nguyen, and T. N. Nguyen, "Bug localization with combination of deep learning and information retrieval," in *IEEE/ACM 25th International Conference on Program Comprehension (ICPC)*. IEEE, 2017, pp. 218–229.

[29] Y. Xiao, J. Keung, K. E. Bennin, and Q. Mi, "Improving bug localization with word embedding and enhanced convolutional neural networks," *Information and Software Technology*, vol. 105, pp. 17–29, 2019.

[30] C. Tantithamthavorn and A. E. Hassan, "An experience report on defect modelling in practice: pitfalls and challenges," in *Proceedings of the 40th International Conference on Software Engineering: Software Engineering in Practice, ICSE (SEIP) 2018, Gothenburg, Sweden, May 27 - June 03, 2018*, 2018, pp. 286–295.

[31] C. Bird, T. Carnahan, and M. Greiler, "Lessons learned from building and deploying a code review analytics platform," in *12th IEEE/ACM Working Conference on Mining Software Repositories, MSR 2015, Florence, Italy, May 16-17, 2015*, 2015, pp. 191–201.

[32] S. K. Lukins, N. A. Kraft, and L. H. Etzkorn, "Bug localization using latent dirichlet allocation," *Information and Software Technology*, vol. 52, no. 9, pp. 972–990, 2010.

[33] 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*. ACM, 2011, pp. 43–52.

[34] A. Agrawal and T. Menzies, "Is "better data" better than "better data miners"?: on the benefits of tuning SMOTE for defect prediction," in *Proceedings of the 40th International Conference on Software Engineering, ICSE 2018, Gothenburg, Sweden, May 27 - June 03, 2018*. ACM, 2018, pp. 1050–1061.

[35] J. Liu, C. Liu, X. Yuan, and N. J. Belkin, "Understanding searchers' perception of task difficulty: Relationships with task type," in *Bridging the Gulf: Communication and Information in Society, Technology, and Work - Proceedings of the 74th ASIS&T Annual Meeting, ASIST 2011, New Orleans, LA, USA, October 9-12, 2011*, ser. Proceedings of the Association for Information Science and Technology, vol. 48, no. 1, pp. 1–10.

[36] D. Kim, Y. Tao, S. Kim, and A. Zeller, "Where should we fix this bug? a two-phase recommendation model," *IEEE transactions on software Engineering*, vol. 39, no. 11, pp. 1597–1610, 2013.

[37] M. Wen, R. Wu, and S.-C. Cheung, "Locus: Locating bugs from software changes," in *31st IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 2016, pp. 262–273.

[38] O. Chaparro, J. M. Florez, and A. Marcus, "Using observed behavior to reformulate queries during text retrieval-based bug localization," in *2017 IEEE International Conference on Software Maintenance and Evolution (ICSME)*. IEEE, 2017, pp. 376–387.

[39] 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*. ACM, 2005, pp. 273–282.

[40] R. Abreu, P. Zoeteweij, R. Golsteijn, and A. J. Van Gemund, "A practical evaluation of spectrum-based fault localization," *Journal of Systems and Software*, vol. 82, no. 11, pp. 1780–1792, 2009.

[41] L. Lucia, D. Lo, L. Jiang, and A. Budi, "Comprehensive evaluation of association measures for fault localization," in *2010 IEEE International Conference on Software Maintenance*. IEEE, 2010, pp. 1–10.

[42] L. Lucia, D. Lo, L. Jiang, F. Thung, and A. Budi, "Extended comprehensive study of association measures for fault localization," *Journal of software: Evolution and Process*, vol. 26, no. 2, pp. 172–219, 2014.

[43] X. Xie, T. Y. Chen, F.-C. Kuo, and B. Xu, "A theoretical analysis of the risk evaluation formulas for spectrum-based fault localization," *ACM Transactions on Software Engineering and Methodology (TOSEM)*, vol. 22, no. 4, p. 31, 2013.

[44] S. Yoo, "Evolving human competitive spectra-based fault localisation techniques," in *International Symposium on Search Based Software Engineering*. Springer, 2012, pp. 244–258.

[45] X. Xie, F.-C. Kuo, T. Y. Chen, S. Yoo, and M. Harman, "Provably optimal and human-competitive results in sbse for spectrum based fault localisation," in *International Symposium on Search Based Software Engineering*. Springer, 2013, pp. 224–238.

[46] J. Xuan and M. Monperrus, "Learning to combine multiple ranking metrics for fault localization," in *IEEE International Conference on Software Maintenance and Evolution*. IEEE, 2014, pp. 191–200.

[47] B. Pytlik, M. Renieris, S. Krishnamurthi, and S. P. Reiss, "Automated fault localization using potential invariants," *arXiv preprint cs/0310040*, 2003.

[48] R. Abreu, A. González, P. Zoeteweij, and A. J. van Gemund, "Automatic software fault localization using generic program invariants," in *Proceedings of the 2008 ACM symposium on Applied computing*. ACM, 2008, pp. 712–717.

[49] S. K. Sahoo, J. Criswell, C. Geigle, and V. Adve, "Using likely invariants for automated software fault localization," in *ACM SIGARCH Computer Architecture News*, vol. 41, no. 1. ACM, 2013, pp. 139–152.

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

[51] V. Dallmeier, C. Lindig, and A. Zeller, "Lightweight defect localization for java," in *European conference on object-oriented programming*. Springer, 2005, pp. 528–550.

[52] C. Liu, X. Yan, L. Fei, J. Han, and S. P. Midkiff, "Sober: Statistical model-based bug localization," in *Proceedings of the 10th European Software Engineering Conference Held Jointly with 13th ACM SIGSOFT International Symposium on Foundations of Software Engineering*. New York, NY, USA: ACM, 2005, pp. 286–295.

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

[54] J. Sohn and S. Yoo, "Fluccs: Using code and change metrics to improve fault localization," in *Proceedings of the 26th ACM SIGSOFT International Symposium on Software Testing and Analysis*. ACM, 2017, pp. 273–283.

[55] M. Zhang, X. Li, L. Zhang, and S. Khurshid, "Boosting spectrum-based fault localization using pagerank," in *Proceedings of the 26th ACM SIGSOFT International Symposium on Software Testing and Analysis*. ACM, 2017, pp. 261–272.

[56] W. E. Wong, R. Gao, Y. Li, R. Abreu, and F. Wotawa, "A survey on software fault localization," *IEEE Trans. Software Eng.*, vol. 42, no. 8, pp. 707–740, 2016.

[57] S. W. Thomas, M. Nagappan, D. Blostein, and A. E. Hassan, "The impact of classifier configuration and classifier combination on bug localization," *IEEE Transactions on Software Engineering*, vol. 39, no. 10, pp. 1427–1443, 2013.

[58] N. Ali, A. Sabane, Y.-G. Gueheneuc, and G. Antoniol, "Improving bug location using binary class relationships," in *2012 IEEE 12th International Working Conference on Source Code Analysis and Manipulation*. IEEE, 2012, pp. 174–183.

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

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

[61] B. Sisman and A. C. Kak, "Assisting code search with automatic query reformulation for bug localization," in *Proceedings of the 10th Working Conference on Mining Software Repositories*. IEEE Press, 2013, pp. 309–318.

[62] M. M. Rahman and C. K. Roy, "Improving ir-based bug localization with context-aware query reformulation," in *Proceedings of the 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. ACM, 2018, pp. 621–632.

[63] O. Chaparro, J. M. Florez, and A. Marcus, "Using bug descriptions to reformulate queries during text-retrieval-based bug localization," *Empirical Software Engineering*, pp. 1–61, 2019.

[64] Q. Wang, C. Parnin, and A. Orso, "Evaluating the usefulness of ir-based fault localization techniques," in *Proceedings of the 2015 International Symposium on Software Testing and Analysis, ISSTA 2015, Baltimore, MD, USA, July 12-17, 2015*, pp. 1–11.

[65] X. Xia, L. Bao, D. Lo, and S. Li, ""automated debugging considered harmful" considered harmful: A user study revisiting the usefulness of spectra-based fault localization techniques with professionals using real bugs from large systems," in *2016 IEEE International Conference on Software Maintenance and Evolution, ICSME 2016, Raleigh, NC, USA, October 2-7, 2016*.