Potential Biases in Bug Localization: Do They Matter?

Pavneet Singh Kochhar, Yuan Tian and David Lo
School of Information Systems
Singapore Management University
{kochharps.2012, yuan.tian.2012, davidlo}@smu.edu.sg

ABSTRACT

Issue tracking systems are valuable resources during software maintenance activities and contain information about the issues faced during the development of a project as well as after its release. Many projects receive many reports of bugs and it is challenging for developers to manually debug and fix them. To mitigate this problem, past studies have proposed information retrieval (IR)-based bug localization techniques, which takes as input a textual description of a bug stored in an issue tracking system, and returns a list of potentially buggy source code files.

These studies often evaluate their effectiveness on issue reports marked as bugs in issue tracking systems, using as ground truth the set of files that are modified in commits that fix each bug. However, there are a number of potential biases that can impact the validity of the results reported in these studies. First, issue reports marked as bugs might not be reports of bugs due to error in the reporting and classification process. Many issue reports are about documentation update, request for improvement, refactoring, code cleanups, etc. Second, bug reports might already explicitly specify the buggy program files and for these reports bug localization techniques are not needed. Third, files that get modified in commits that fix the bugs might not contain the bug.

This study investigates the extent these potential biases affect the results of a bug localization technique and whether bug localization researchers need to consider these potential biases when evaluating their solutions. In this paper, we analyse issue reports from three different projects: HTTP-Client, Jackrabbit, and Lucene-Java to examine the impact of above three biases on bug localization. Our results show that one of these biases significantly and substantially impacts bug localization results, while the other two biases have negligible or minor impact.

Categories and Subject Descriptors

D.2.7 [**Software**]: Software Engineering—*Distribution, Maintenance, and Enhancement*

General Terms

Experimentation

Keywords

Issue Reports, Bug Localization, Bias, Empirical Study

1. INTRODUCTION

Issue tracking systems, which contains information related to issues faced during the development as well as after the release of a software project, is an integral part of software development activity. Issue tracking systems such as JIRA or Bugzilla can help reporters report various kinds of issues such as bug reports, documentation update, refactoring request, addition of new feature and so on. Well-known projects often receive large number of issue reports which might be difficult for developers to handle. A mozilla developer accepted that the project receives over 300 bugs per day which needs triaging [5]. Therefore, it is important to have techniques which can help developers find buggy files quickly, which can help them resolve the bug faster.

To overcome the above issue, researchers have proposed techniques which use information given in the bug report to identify source code files that contain the bug [20, 21, 26]. These techniques often use standard information retrieval (IR) techniques to compute the similarity between the textual description of bug report and textual description of source code. Based on the similarity scores, these IR-based bug localization techniques return a ranked list of source code files which are likely to be buggy for that bug report. These techniques are evaluated using closed and fixed issue reports marked as bugs collected from issue tracking systems. The evaluation involves comparison of files returned by bug localization techniques with the actual files changed to fix the bug. Past studies indicate that the performance of these techniques are promising – up to 80% of bug reports can be localized by just inspecting 5 source code files [21].

Despite the promising results of IR-based bug localization approaches, a number of potential biases can affect the validity of results reported in prior studies. If these biases significantly affect the results of bug localization studies, future researchers need to put more care in *cleaning* their evaluation datasets when evaluating the performances of their techniques. In this work, we focus on investigating three potential biases:

1. **Wrongly Classified Reports.** Herzig et al. reported that many issue reports in issue tracking systems are

wrongly classified [12]. About one third of all issue reports marked as bugs are not really bugs. Herzig et al. have shown that this potential bias significantly affects bug prediction studies that predict whether a file is potentially buggy or not based on the history of prior bugs. This potential bias might affect bug localization studies too as the characteristics of bug reports and other issues, e.g., refactoring requests, can be very different. Refactoring can touch a large number of files, while bug fixes are often more localized [16, 24]. Thus, there is a need to investigate whether wrongly classified reports significantly skew effectiveness results of bug localization approaches.

2. **Already Localized Reports.** Our manual investigation of a number of bug reports find that the textual descriptions of many reports have already specified the files that contain the bug. These *localized reports* do not require bug localization approaches. The buggy files are already localized and only need to be fixed. Evaluating bug localization approaches with these localized reports will unfairly inflate the effectiveness results. Thus, there is a need to investigate how common are localized reports and whether their presence in the evaluation data set can significantly skew the effectiveness results of bug localization approaches.

3. **Incorrect Ground Truth Files.** Kawrykow and Robillard reported that many changes made to source code files are non-essential changes [14]. These non-essential changes include cosmetic changes made to source code which do not affect the behavior of systems. Past fault localization studies often use as ground truth source code files that are touched by commits that fix the bugs [21, 26]. However, no manual investigation was done to check if these files are affected by essential or non-essential changes. Files that are affected by non-essential changes should be excluded from the ground truth files as they do not contain the bug. Including these non-essential changes as ground truth files can unfairly inflate the effectiveness results – with more ground truth files, there is a higher chance that one of them will be identified by a bug localization tool. Thus, there is a need to investigate how common are the incorrect ground truth files and whether their presence in the evaluation data set can significantly skew the effectiveness results of bug localization approaches.

To investigate the impact of the above mentioned possible biases, in this study, we analyse the following research questions:

RQ1: *What are the effects of wrongly classified issue reports on bug localization?*
RQ2: *What are the effects of localized bug reports on bug localization?*
RQ3: *What are the effects of wrongly identified ground truth files on bug localization?*

The first research question has been partly answered in our short paper in MSR 2014 [15]. We extend our short paper by performing a per-project analysis instead of an all-project analysis; we also employ an additional test to shed more light on the research question. The other two research questions have not been investigated in prior work.

To answer the above research questions, we reuse the manually categorized issue reports dataset of Herzig et al. [12]. Herzig et al.'s dataset consists of issue reports from Bugzilla and JIRA issue tracking systems. Several studies in the past have shown that the bug reports in Bugzilla are poorly linked [6, 8, 25], whereas bug reports in JIRA are well linked as JIRA provides add-ons to help connect issues to commits in the version control systems [9]. Therefore, we only use projects from Herzig et al.'s dataset that use JIRA. We investigate 5,591 issue reports stored in issue tracking systems of three projects: HTTPClient, Jackrabbit and Lucene-Java. HTTPClient is a Java library for implementing the client side of the most recent HTTP standards and recommendations [1]. Jackrabbit is a content repository written in Java [2]. Lucene-Java is a high-performance search engine library written in Java [3]. Table 1 shows the dataset we use for this study.

Table 1: Project Details

Project	Issue Tracker	#Issue Reports
HTTPClient	JIRA	746
Jackrabbit	JIRA	2,402
Lucene-Java	JIRA	2,443

These reports have been manually classified by Herzig et al. into bug reports and other kinds of issues [12]. We perform suitable statistical tests (Mann-Whitney-Wilcoxon test [17] and Fisher exact test [11]) and compute effect size measure (Cohen's d [10]) to investigate whether the skew introduced by the above potential biases matters. We find that some biases do not significantly or substantially affect bug localization techniques performance. However, one of them significantly and substantially affect bug localization techniques performance and researchers need to consider this bias by cleaning their evaluation dataset to correctly measure the effectiveness of bug localization solutions.

The contributions of this paper are as follows:

1. We extend our preliminary study [15], to analyze the effect of wrongly classified issue reports on the effectiveness of bug localization tools.

2. We analyze the effect of localized bug reports on the effectiveness of bug localization tools. To do this, we manually investigate bug reports and categorize them as localized, partially localized, or non-localized. We also build an automated technique that can categorize bug reports into these three categories with high accuracy.

3. We analyze the effect of incorrect ground truth files on the effectiveness of bug localization tools. To do this, we manually investigate source code files that are touched by commits to fix various bugs and detect irrelevant files that do not contain bugs.

4. We release a clean dataset that researchers can use to evaluate future bug localization techniques.

The structure of this paper is as follows. In Section 2, we give a brief summary of IR-based bug localization techniques. In Sections 3, 4 and 5, we investigate RQ1, RQ2, and RQ3 respectively. We discuss other interesting findings and threats to validity in Section 6. Related work is discussed in Section 7. We finally conclude and mention future work in Section 8.

2. BUG LOCALIZATION: A PRIMER

In Recent years, a number of studies have proposed information retrieval based approaches to automatically locate bugs in source code files. They take bug reports as input queries and source code files as input documents and output a list of relevant source code files to a particular bug report. The major goal is to find a well-designed metrics to capture the similarity between bug reports and source code files. Many techniques have been proposed to compute such similarity based on textual information stored in bug reports and source code, e.g., words appear in bug reports and source code files.

In Section 2.1 we describe a general bug localization framework and highlight one popular technique based on Vector Space Model (VSM). In Section 2.2, we describe how bug localization studies often obtain ground truths to evaluate the effectiveness of their techniques when applied on bug reports. In Section 2.3, we describe a popular metric to measure the effectiveness of bug localization techniques namely Mean Average Precision (MAP).

2.1 General Framework

The general framework contains three major steps: code corpus pre-processing and indexing, bug report pre-processing, and retrieval and ranking. We describe the details of each step as follows.

Step 1: Code Corpus Pre-processing and Indexing. We apply three preprocessing steps to process code corpus (i.e., source code files): normalization, stop word removal, and stemming. In the normalization step, we first extract comments, identifiers, and string literals from source code files. Tools like Eclipse JDT[1] can be used to extract Abstract Syntax Trees (ASTs) that can then be traversed to extract these textual contents from source code files. Next, we remove punctuation marks, special symbols, and number literals from the extracted text. We also convert all words into lower case. In the stop word removal step, we first remove commonly occurring English words (e.g., "I", "you", "we", "are", etc.). We use the stop word list from: `http://dev.mysql.com/doc/refman/5.6/en/fulltext-stopwords.html`. We also remove programming language keywords, e.g., *public*, *class*, *if*, *for*, etc. In the stemming step, we apply the famous Porter Stemming Algorithm[2] to reduce a word to its root form. For example, we reduce "mapping", "mapped", and "maps" to "map".

At the end of the above steps, each of the files in the corpus are represented by a bag of words. We then index these bags of words so that one can locate files containing a particular word efficiently.

Step 2: Bug Report Pre-Processing. IR-based bug localization approaches regard one bug report as a query. Textual information inside a bug report such as the content of the summary and description fields are extracted. Table 2 shows an example bug report with ID JCR-2718 from Jackrabbit Project along with the contents of its summary and description fields. After the text of the bug reports are extracted, we perform several pre-processing steps: tokenization, stop-word removal, and stemming. In the tokenization step, we convert the textual content of a bug report into a multi-set (bag) of words that appear in it. This bag of words are then input to the stop-word removal and stemming steps which are are the same ones that are applied to pre-process the code corpus.

Table 2: Issue Report JCR-2718 from Jackrabbit

Summary:	Incorrect results from joins on multi-valued properties
Description:	It looks like join conditions on multi-valued properties only use one of the multiple values for the comparison.

Step 3: Retrieval and Ranking. In this step, given a bug report, we want to retrieve and rank relevant source code files. Retrieval and ranking of relevant source code files is based on the similarity between the bug report and each of the files in the code corpus. In the IR domain, various models have been proposed to measure the similarities of a query (in our case: bug report) with a document (in our case: a source code file). These include Vector Space Model (VSM), Smoothed Unigram Model (SUM), Latent Dirichlet Allocation (LDA), etc.

In this work, we focus on VSM, which is the foundation of many state-of-the-art IR-based bug localization approaches, e.g., [21, 26], and has been shown to outperform many other models [20]. VSM takes the pre-processed bug report as a query and pre-processed source code files as documents. In VSM, each query or document is expressed as a vector of weights, where each weight corresponds to a word that appears in the query or document. The weight is usually computed using the tf-idf weighting scheme. The tf-idf weight of word w in document d given a set of documents D, denoted as $tf - idf(w, d, D)$, is computed as:

$$tf - idf(w, d, D) = log(f(w, d) + 1) \times log \frac{|D|}{|d_i \in D : w \in d_i|} \quad (1)$$

In the above equation, $f(w, d)$ is the number of times word w occurs in document d, and $d_i \in D : w \in d_i$ represents documents that contain word w. After converting queries and documents into vectors of weights, VSM computes the similarity between a query and a document as the cosine similarity between the two corresponding vectors. Equation 2 shows the function to calculate the cosine similarity between a query q and a document d.

$$Similarity(q, d) = cos(q, d) = \frac{\overline{V_q} \bullet \overline{V_d}}{|\overline{V_q}||\overline{V_d}|} \quad (2)$$

where $\overline{V_q}$ and $\overline{V_d}$ are vector of term weights for query q and document d, respectively. $\overline{V_q} \bullet \overline{V_d}$ represents the inner product of the two vectors.

2.2 Ground Truth Identification

IR-based bug localization approaches are often evaluated on bug reports submitted to various issue tracking systems.

[1]`http://www.eclipse.org/jdt/`
[2]`http://tartarus.org/martin/PorterStemmer/`

For each bug report, there is a need to get the *ground truth files* which are the source code files that contain the bug.

Manual identification of ground truth files would takes a lot of effort, therefore, researchers have come up with an automated yet imperfect approach to identify these files. The underlying idea is to map a bug report to commits in version control systems that fix the bug. Often the identifier of the bug report appears in the logs of the corresponding commits. After these commits are identified, files that are touched by one or more of these commits are considered as ground truth files. Bug localization techniques are then evaluated based on their ability to recover these ground truth files from the bug report.

2.3 Evaluation Metric

A bug localization technique outputs a ranked list of files for every bug report. Given a set of bug reports, the technique will output a set of ranked lists. A number of metrics can be used to evaluate the effectiveness of the technique based on the position of the buggy files in the ranked lists. One of the most popular metrics is Mean Average Precision (MAP) which has been used to evaluate many recent studies [20, 21, 26].

To compute MAP, for each ranked list, we need to compute Average Precision (AP), which is defined as follows:

$$AP = \sum_{i=1}^{M} \frac{P(i) \times rel(i)}{\#\text{All buggy files}} \tag{3}$$

where M is the number of retrieved source code files, $rel(i)$ is a binary value that represents whether the ith retrieved source file is buggy or not. $P(i)$ is the precision at position i of the ranked list, which is defined as:

$$P(i) = \frac{\#\text{Buggy files retrieved in top i positions}}{\text{i}} \tag{4}$$

MAP is then the mean of the average precisions over all ranked lists produced for the bug reports. The higher the MAP of a bug localization technique, the more effective is the technique.

3. BIAS 1: REPORT MISCLASSIFICATION

In this section, we investigate the first research question: *What are the effects of wrongly classified issue reports on bug localization?* We describe the motivation of answering this question in Section 3.1, the methodology of our experiments in Section 3.2, and the results of our experiments which answer the question in Section 3.3.

3.1 Motivation

Issue tracking systems contain reports of several types of issues such as bugs, requests for improvement, documentation, refactoring, etc. Herzig et al. report that a substantial number of issue reports marked as bugs, are not bugs but other kinds of issues. Their results show that these misclassifications have a significant impact on bug prediction. In this question, we want to analyse the consequences of misclassification on bug localization.

3.2 Methodology

Step 1: Data Acquisition. We use Herzig et al.'s dataset of manually analyzed issue reports (www.st.cs.unisaarland.de/-softevo/bugclassify). We download the issue reports from the associated JIRA repositories and extract the textual contents of the summary and description of the reports. After downloading, we perform the preprocessing steps described previously. In JIRA, each issue report has a unique identifier represented by the project name and a unique number. For example, *HTTPCLIENT-974* represents issue number 974 of project *HTTPClient*. We use the *git* version control system of the projects to get the commit log files, which are used to map issue reports to their corresponding commits. Commit logs contain unique identifier of the issue report as part of the commit message. We use these mapped commits to check out the source code files prior to the commits that address the issue and the source code files when the issue is resolved. For each source code file, we perform a similar preprocessing step to represent a file as a bag-of-words.

Step 2: Bug Localization. After the data acquisition, we have the textual content of the issue reports, the textual content of each source code file in the revision prior to the fix, and a set of ground truth files that are changed to fix the issue report. We give the textual content of the issue reports and the revision's source code files as input to the bug localization technique, which outputs a ranked list of files sorted based on the similarity to the bug report.

Step 3: Effectiveness Measurement & Statistical Analysis. After Step 2, for each issue report, we have a ranked list of source code files and a list of supposed ground truth files. We compare these two lists to compute the average precision score.

We divide the issue reports into two categories: issue reports marked as bugs in the tracking system (Reported) and issue reports that are actual bugs i.e., manually labeled by Herzig et al. (Actual). In Herzig et al.'s dataset, the set Actual is a subset of Reported. We compute the MAP scores and use Mann-Whitney U test to examine the difference between these two categories at 0.05 significance level. We use Cohen's d to measure the effect size, which is the standardised difference between two means. To interpret the effect size, we use the interpretation given by Cohen [10], i.e., d < 0.2 means trivial, $0.20 \leq$ d < 0.5 means small, $0.5 \leq$ d < 0.8 means medium, $0.80 \leq$ d < 1.3 means large, and d $\geq$ 1.3 means very large.

Table 3: Mean Average Precision (MAP) Scores for Reported and Actual

Project	Reported	Actual	Difference	d
HTTPClient	0.429	0.419	-2.33%	0.13
Jackrabbit	0.302	0.339	12.25%	0.06
Lucene-Java	0.301	0.322	6.98%	0.04

3.3 Results

Effect of Misclassification on Bug Localization. Table 3 shows the MAP scores for the two categories: reports marked as bugs (Reported) and manually classified bug reports (Actual). We observe that there are differences of -2.33%, 12.25% and 6.98% in the MAP scores for HTTPClient, Jackrabbit and Lucene-Java, respectively. We perform the Mann-Whitney Wilcoxon test and compute Cohen's d to examine the differences between the two categories. The results are also presented in Table 3. From the results, we observe that, for HTTPClient and Lucene-Java, the differences are statistically insignificant and the effect sizes are trivial (i.e., less than 0.2). For Jackrabbit, the effect size is trivial, however, the difference is statistically significant.

Effect of Different Misclassification Types. We now analyse the misclassification type that has the most impact on the difference of MAP scores between Reported and Actual. Herzig et al. classify issue reports into 13 categories: `BUG`, `RFE`, `IMPROVEMENT`, `DOCUMENTATION`, `REFACTORING`, `BACKPORT`, `CLEANUP`, `SPEC`, `TASK`, `TEST`, `BUILD_SYSTEM`, `DESIGN_DEFECT`, and `OTHERS`. We omit issue reports that are misclassified one category at a time and recalculate the MAP score. For example, `RFE` to `BUG` represents issue reports which are RFE (Actual) but are misclassified as BUG (Reported). Table 4 shows the MAP scores when we remove issue reports of particular misclassification types one at a time. Each row corresponds to a subset of reports where reports of a misclassification type is removed. We observe that `TEST` to `BUG` has the largest difference in the MAP score followed by misclassification from `IMPROVEMENT` to `BUG`.

Table 4: Mean Average Precision (MAP) Scores when Issue Reports of a Particular Misclassification Type are Omitted. Omit. = Omitted, Misclass. = Misclassification, HC = HTTPClient, JB = Jackrabbit, LJ = Lucene-Java. The last column is the MAP of all three projects.

Omit. Misclass. Type (Actual to Reported)	HC	JB	LJ	Overall
None	0.429	0.302	0.301	0.312
`RFE` to `BUG`	0.427	0.303	0.304	0.313
`DOCUMENTATION` to `BUG`	0.43	0.304	0.305	0.315
`IMPROVEMENT` to `BUG`	0.416	0.299	0.295	0.307
`REFACTORING` to `BUG`	0.428	0.301	0.301	0.311
`BACKPORT` to `BUG`	0.43	0.303	0.300	0.313
`CLEANUP` to `BUG`	0.429	0.303	0.303	0.314
`SPEC` to `BUG`	0.435	0.302	0.303	0.312
`TASK` to `BUG`	0.432	0.302	0.301	0.312
`TEST` to `BUG`	0.429	0.328	0.313	0.334
`BUILD_SYSTEM` to `BUG`	0.429	0.306	0.303	0.315
`DESIGN_DEFECT` to `BUG`	0.424	0.301	0.301	0.311
`OTHERS` to `BUG`	0.439	0.303	0.301	0.313

Bias 1, which is wrongly classified issue reports, significantly impacts bug localization result for one out of the three projects. However, the effect of this bias is ***negligible*** *– the effect sizes are less than 0.2.*

4. BIAS 2: LOCALIZED BUG REPORTS

In this section, we investigate the second research question: *What are the effects of localized bug reports on bug localization?* We describe the motivation of answering this question in Section 4.1, the methodology of our experiments in Section 4.2, and the results of our experiments which answer the question in Section 4.3.

4.1 Motivation

Localized bug reports are those whose buggy files have been identified in the report itself. For these reports, the remaining task to resolve the bug is simply to fix the buggy files. These bug reports do not benefit or require bug localization solutions. Past studies on bug localization do not separate localized from non-localized bug reports. In this research question, we want to investigate the number of localized bug reports and the impact of including localized bug reports in the evaluation of bug localization tools. If bias exists, then future bug localization solutions need to be careful to perform a data cleaning step to remove these localized bug reports from their evaluation dataset.

4.2 Methodology

To investigate this research question, we first need to identify localized bug reports. We start by manual investigating of a smaller subset of bug reports and identify localized ones. We then developed an automated means to find localized bug reports so that our analysis can scale to a larger number of bug reports. Finally, we input these reports to a number of IR-based bug localization tools to investigate whether localized reports skew the results of bug localization tools.

Table 5: Fully Localized, Partially Localized, and Not Localized Reports

Category	Description
Fully	Bug reports where all the files containing the bugs are explicitly specified in the report.
Partially	Bug reports where some of the files containing the bugs are explicitly mentioned in the report.
Not	Bug reports which do not explicitly specify any of the buggy files.

Step 1: Manually Identifying Localized Bug Reports. We manually analysed 350 issue reports that Herzig et al. labeled as bug reports. Out of the 5,591 issue reports from the three projects, Herzig et al. labeled 1,191 of them as bug reports. We randomly selected these 350 from the pool of bug reports from the three software projects. For our manual analysis, we read the summary and description fields of each bug report. We also collected the corresponding files changed to fix each bug. We classified each bug report into one the three categories shown in Table 5. Table 6, 7, and 8 show example bug reports that are fully localized, partially localized, and not localized.

Table 6: Fully Localized Report: HTTPCLIENT-1078

Summary:	**DecompressingEntity** not calling close on InputStream retrieved by getContent
Description:	The method **DecompressingEntity**.writeTo(OutputStream outstream) does not close the InputStream retrieved by getContent(). According to the documentation of HttpEntity.writeTo: IMPORTANT: Please note all entity implementations must ensure that all allocated resources are properly deallocated when this method returns. -> imho this is not satisfied in **DecompressingEntity**.writeTo
Buggy Files:	**DecompressingEntity.java**

Table 7: Partially Localized Report: JCR-814

Summary:	Oracle bundle PM fails checking schema if 2 users use the same database
Description:	When using the OracleBundlePersistenceManager there is an issue when two users use the same database for persistence. In that case, the checkSchema() method of the **BundleDbPersistenceManager** does not work like it should. More precisely, the call "metaData.getTables(null, null, tableName, null);" will also includes table names of other schemas/users. Effectively, only the first user of a database is able to create the schema. probably same issue as here: JCR-582
Buggy Files:	**BundleDbPersistenceManager.java**, **OraclePersistenceManager.java**

Table 8: Not Localized Report: LUCENE-3721

Summary:	CharFilters not being invoked in Solr
Description:	On Solr trunk, all CharFilters have been non-functional since LUCENE-3396 was committed in r1175297 on 25 Sept 2011, until Yonik's fix today in r1235810; Solr 3.x was not affected - CharFilters have been working there all along.
Buggy Files:	**TokenizerChain.java**

Step 2: Automatic Identification of Localized Reports. In this step, we build an algorithm that takes in a set of files that are changed in bug fixing commits and a bug report, and outputs one of the three categories described in Table 5. Our algorithm first extracts the text that appear in the summary and description fields of bug reports. Next, it tokenizes this text into a set of word tokens. Finally, it checks whether the name of each buggy file (ignoring its filename extension) appears as a word token in the set. If all names appear in the set, our algorithm categorizes the report as *fully localized.* If only some of the names appears in the set, it categorizes the bug report as *partially localized.* Otherwise, it categorizes the bug report as *not localized.* We have evaluated our algorithm on the 350 manually labeled bug reports and find that its accuracy is close to 100%.

Step 3: Application of IR-Based Bug Localization Techniques. After localized, partially localized, and not localized reports are identified, we create three groups of bug reports. We feed each of them into the VSM-based bug localization tool described in Section 2. We then evaluate the effectiveness of these tools for each of the three groups of reports.

Step 4: Statistical Analysis. We perform two statistical analyses. First, we compare the average precision scores achieved by VSM-based bug localization tool for the set of fully localized, partially localized, and not localized reports using Mann-Whitney-Wilcoxon test at 5% significance level. We also compute Cohen's d on the average precision scores to see if the effect size is small, medium or large.

Second, we compare a subset of bug reports where the VSM-based bug localization technique performs *the best* and another subset where the VSM-based bug localization techniques performs *the worst.* We then compare the distribution of fully, partially, and not localized bugs in these two subsets. We employ Fisher exact test [11] to see if the distribution for the first subset significantly differs with the distribution for the second subset.

4.3 Results

Number of Fully Localized, Partially Localized, and Not Localized Reports. The numbers of bug reports that are identified as fully, partially, and not localized are shown in Table 9. We can observe that out of 1,191 bug reports, 398 (33.41%) bug reports are fully localized i.e., the bug reports contains the name of all the class files changed to fix the bug. Over 50% of the bug reports are either fully or partially localized. This shows that a significant number of bug reports are already localized, and do not benefit from a bug localization algorithm. On the other hand, 546 bug reports (45.84%) are not localized at all.

Table 9: Fully, Partially, and Not Localized Reports

Project	Category	Number	Proportion
HTTPClient	Fully	36	3.02%
	Partially	28	2.35%
	Not	35	2.93%
Jackrabbit	Fully	299	25.10%
	Partially	132	11.08%
	Not	402	33.75%
Lucene-Java	Fully	63	5.28%
	Partially	87	7.30%
	Not	109	9.15%

Average Precision Scores of Fully vs. Partially vs. Not Localized Reports. Table 10 shows the Mean Average Precision (MAP) of the VSM-based bug localization technique when applied to the set of fully, partially, and not-localized reports. We can note that the MAP score differences between fully localized and not localized bug reports for HTTPClient, Jackrabbit, and Lucene-Java are 84.39%, 99.86% and 91.16% respectively. Also, the MAP score differences between partially localized and not localized bug reports for HTTPClient, Jackrabbit, and Lucene-Java are 33.05%, 66.42% and 52.71% respectively.

Table 10: MAP Scores: Fully vs. Partially vs. Not

Project	Fully	Partially	Not
HTTPClient	0.615	0.349	0.250
Jackrabbit	0.560	0.373	0.187
Lucene-Java	0.527	0.338	0.197

We also perform Mann-Whitney Wilcoxon test to examine the difference between the following categories: fully & partially, partially & not and fully & not. Table 11 shows the p-values between different categories. The results show that there are significant differences between average precision scores of fully localized and partially localized bug reports, fully localized and partially localized bug reports,

Table 11: Comparison: Fully vs. Partially vs. Not

Project	Fully-Partially			Partially-Not			Fully-Not		
	p-value	d	Effect Size	p-value	d	Effect Size	p-value	d	Effect Size
HTTPClient	0.007	**0.94**	Large	0.007	0.53	Medium	$3.094e^{-05}$	**1.27**	Large
Jackrabbit	$4.544e^{-05}$	0.56	Medium	$< 2.2e^{-16}$	0.55	Medium	$< 2.2e^{-16}$	**1.14**	Large
Lucene-Java	0.010	0.53	Medium	$1.851e^{-05}$	0.41	Small	$3.183e^{-09}$	**1.04**	Large

and partially localized and not localized bug reports, i.e., all the p-values are less than 0.05. We also compute Cohen's d to measure an effect size and find that the effect sizes are small to large. The effect sizes between average precision scores of fully localized and not localized bug reports are large for all three projects. This shows that there is a large substantial difference in the effectiveness of a bug localization tool when applied to bug reports that are fully localized and those that are not localized.

Best vs. Worst Bug Reports. We want to examine the difference between the proportion of bug reports that are fully, partially, and not localized in the upper and lower quartile of the bug reports based on the ability of the VSM-based bug localization tool to localize them. We simply sort the bug reports based on their average precision scores and identify the subset that appear in the top 25% of the list (upper quartile) and another subset that appear in the bottom 25% of the list (lower quartile). For Jackrabbit and Lucene-Java, we randomly select 50 bug reports from the upper quartile and another 50 from the lower quartile. For HTTPClient, we randomly select 25 bug reports from the upper quartile and another 25 from the lower quartile – since in our dataset, HTTPClient has less than 100 bug reports.

Table 12 shows the number of fully, partially and not localized bugs for each of the projects. We use Fisher exact test to examine the difference between the distribution of fully localized, partially localized, and not localized bug reports in the upper and lower quartiles. The null hypothesis is that there is no difference between the distribution of fully, partially, and not localized bug reports in the upper and lower quartiles. The alternate hypothesis is that there is a significant difference between the distribution of bug reports in the upper and lower quartiles. We find that the p-values for all the projects are very small, which shows that there is a significance difference in the distribution of fully localized, partially localized, and not localized bug reports between the best and worst bug reports.

Bias 2, which is localized bug reports, ***significantly and substantially impacts*** *bug localization results.* ***More than 50%*** *of the bugs are (already) localized either fully or partially; these reports explicitly mention some or all of the files that are buggy and thus do not require a bug localization algorithm. The mean average precision scores for fully and partially localized bug reports are* ***much higher*** *(i.e., significantly and substantially higher) than those for not localized bug reports. The effect sizes of average precision scores between fully and not localized bug reports are large for all three projects.*

Table 12: Fisher Exact Test: Best vs. Worst Reports

Project		Fully	Partially	Not	p-value
HTTPClient	Upper	16	5	4	0.0041
	Lower	6	4	15	
Jackrabbit	Upper	35	9	6	$2.807e^{-13}$
	Lower	7	1	42	
Lucene-Java	Upper	22	18	10	$8.724e^{-05}$
	Lower	5	18	27	

5. BIAS 3: NON-BUGGY FILES

In this section, we investigate the third research question: *What are the effects of wrongly identified ground truth files on bug localization?* We describe the motivation of answering this question in Section 5.1, the methodology of our experiments in Section 5.2, and the results of our experiments which answer the question in Section 5.3.

5.1 Motivation

Another issue which can bias the result is wrongly identified ground truth files. In past studies, wrongly identified ground truth files have not been removed since they require additional analysis. These wrongly identified ground truth files can potentially skew the result of existing bug localization solutions. In this research question, we want to investigate to what extent do wrongly identified ground truth files affect bug localization.

5.2 Methodology

Step 1: Manually Identifying Wrong Ground Truth Files. We randomly select 100 bug reports that are not (already) localized (i.e., these reports do not explicitly mention any of the buggy files) and investigate the files that are modified in the bug fixing commits. We manually perform a *diff* that gives us the differences between the modified file and the original file. Based on these differences we manually decide if a file contains a bug or not. Files that are only affected by cosmetic changes, refactorings, etc. are considered as non-buggy files. Based on this manual analysis, for each bug report we have the set of *clean* ground truth files and another set of *dirty* ground truth files.

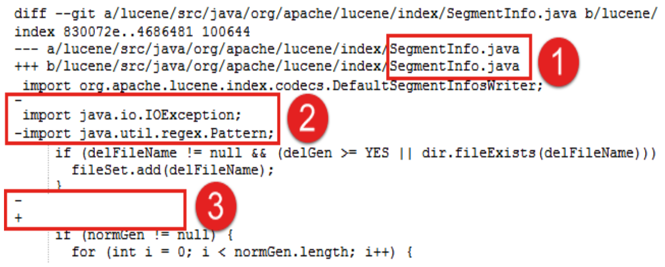
Thung et al. have extended Kawrykow and Robillard work [14] to automatically identify real ground truth files [22]. However the accuracy of their proposed technique is still relatively low (i.e., precision and recall scores of 76.42% and 71.88%). Hence, we do not employ any automated tool to identify wrong ground truth files. We also cannot extend the study to investigate a large number of bug reports since the identification of wrong ground truth files is time consuming.

Step 2: Application of IR-Based Bug Localization Techniques. After the set of clean and dirty ground truth files are identified for each of the 100 bug reports, we input the 100 bug reports to a VSM-based bug localization tool described in Section 2. We evaluate the results of the tool on dirty and clean ground truth files.

Step 3: Statistical Analysis. We compare the average precision scores achieved by the VSM-based bug localization tool for the 100 bug reports with clean and dirty ground truth files using Mann-Whitney-Wilcoxon test at 5% significance level. We also compute Cohen's d on the average precision scores to see if the effect size is small, medium or large.

5.3 Results

Number of Wrong Ground Truth Files. We found that out of 498 files changed to fix the 100 bugs, only 358 files are really buggy. The other 140 files (28.11 %) do not contain any of the bugs but are changed because of refactorings, modifications to program comments, due to changes made to the buggy files, etc. Figure 1 shows the diff of a file that is changed in a commit that fix bug report LUCENE-2616. The content of the bug report with ID LUCENE-2616 is shown in Table 13.



```
diff --git a/lucene/src/java/org/apache/lucene/index/SegmentInfo.java b/lucene/
index 830072e..4686481 100644
--- a/lucene/src/java/org/apache/lucene/index/SegmentInfo.java
+++ b/lucene/src/java/org/apache/lucene/index/SegmentInfo.java
 import org.apache.lucene.index.codecs.DefaultSegmentInfosWriter;
-
 import java.io.IOException;
-import java.util.regex.Pattern;
     if (delFileName != null && (delGen >= YES || dir.fileExists(delFileName)))
       fileSet.add(delFileName);
     }
-
+
     if (normGen != null) {
       for (int i = 0; i < normGen.length; i++) {
```

Figure 1: Example Diff of a File that is Changed to Fix a Bug in Lucene-Java Project with ID LUCENE-2616. Note: (1) The name of the file: SegmentInfo.java; (2) An empty line and an import statement are deleted; (3) An empty line is deleted and another one is added.

Table 13: Bug Report: LUCENE-2616

Summary:	FastVectorHighlighter: out of alignment when the first value is empty in multiValued field
Description:	-
Non-Buggy File:	SegmentInfo.java

Table 14: MAP Scores: Dirty vs. Clean Ground Truths

Project	Dirty	Clean	Difference	d
HTTPClient	0.207	0.171	0.036	0.08
Jackrabbit	0.115	0.115	0.000	0.08
Lucene-Java	0.271	0.239	0.032	0.17

MAP Scores: Dirty vs. Clean. We compare the Mean Average Precision (MAP) scores of these 100 bug reports when evaluated on dirty and clean ground truths. Table 14 shows that the differences in the MAP scores are between 0 to 0.036. We also ran Mann-Whitney Wilcoxon test and compute Cohen's d to check if each difference is significant or substantial. We find that the difference is not statistically significant and the effect size is trivial (< 0.2).

*Bias 3, which is incorrect ground truth files, **neither significantly nor substantially affects** bug localization results. We notice that **28.11%** of the files present in the ground truth (i.e., they are changed in a commit that fix a bug) are non-buggy. Also, there is a difference of 0-0.036 between the MAP scores when a bug localization tool is evaluated on dirty and clean ground truth. However, this difference is neither statistically significant nor substantial.*

6. OTHER FINDINGS AND THREATS

In this section, we first describe the effects of the biases measured by several other popular evaluation metrics. Next, we describe some threats to validity.

6.1 Other Evaluation Metrics

Beside Mean Average Precision (MAP) which we used in the previous sections, HIT@N and MRR have also been used to evaluate bug localization studies [20, 21, 26]. HIT@N and MRR are presented below:

- **HIT@N:** This metric counts the percentage of bug reports with at least one buggy file found in the top N (e.g., 1) ranked results.
- **MRR (Mean Reciprocal Rank):** The reciprocal rank of a bug report is the inverse of the rank of the first buggy file in the ranked results. The mean reciprocal rank takes the average of the reciprocal ranks of all bug reports. For a set of bug reports Q, MRR is defined as:

$$MRR = \frac{1}{|Q|}\sum_{i=1}^{Q}\frac{1}{rank_i} \tag{5}$$

where $rank_i$ is the rank of the first buggy file in the output ranked list.

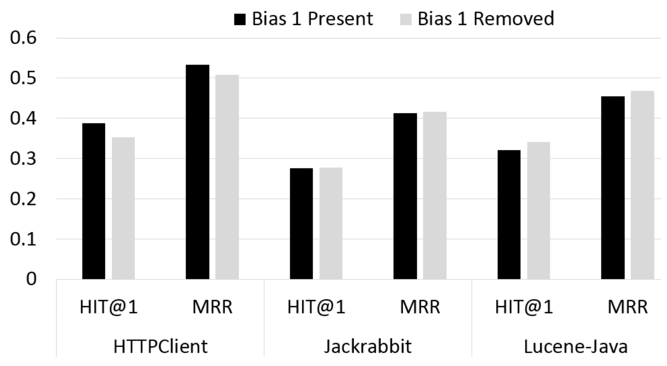


Figure 2: Before and After Removing Bias 1

The effect of bias 1, bias 2, and bias 3 measured by HIT@1 and MRR are shown in Figures 2 to 4. Figure 2 shows that for bias 1, its effect in terms of HIT@1 and MRR scores is

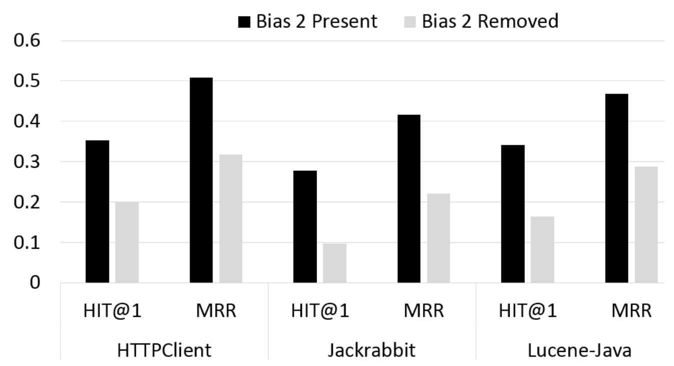


Figure 3: Before and After Removing Bias 2

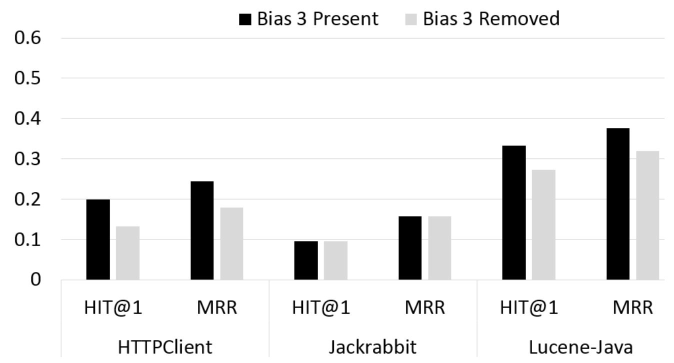


Figure 4: Before and After Removing Bias 3

minimal. Figure 3 shows that for bias 2, its effect in terms of HIT@1 and MRR score is substantial. Figure 4 shows that for bias 3, for Jackrabbit, its effect is minimal. For HTTPClient and Lucene-Java, its effect is more apparent albeit not as substantial as the effect of bias 2.

For MRR since it is a mean of a distribution, we also run Mann-Whitney-Wilcoxon test and compute Cohen's d values. The results are shown in Table 15. We find that for bias 1, its effect is not statistically significant for all projects. For bias 2, its effect is both statistically significant and substantial when comparing the results of (fully or partially) localized bug reports with results of not localized bug reports. For bias 3, its effect is not statistically significant for all projects.

To conclude, the above results show that bias 2 has substantial effect on the performance of bug localization techniques. The effects of bias 1 and 3 are more minor or even negligible. These results are in line with the findings of Sections 3, 4, and 5.

6.2 Threats to Validity

Threats to internal validity corresponds to errors in our experiments and our labeling. In this work, these threats are coming from human classification of bug reports. For bias 1, we consider the same issue reports that were manually categorized by Herzig et al. [12]. For bias 2, we manually categorize 350 bug reports as fully localized, partially localized, or not localized. Also, we design an automated algorithm to identify localized reports and find that it performs very well on the manually labeled bug reports. However, it is not clear if performs as well on other bug reports in our dataset. For bias 3, we manually categorize files that are changed in commits that fix 100 bug reports. To reduce bias, two PhD students majoring in software engineering analyze the bug reports and agree on the labels. Threats to external validity relates to the generalizability of our findings. In this work, we consider three open source projects: HTTPClient, Jackrabbit, and Lucene-Java. We analyze 5,591 bug reports for RQ1 (bias 1), 1,191 bug reports for RQ2 (bias 2), and 100 bug reports for RQ3 (bias 3). We plan to include more projects and analyze more bug reports in a future work. We have also only analyzed VSM-based bug localization approach. There are many other techniques proposed in the literature and we plan to analyze them in a future work. Many of these techniques are based on VSM, e.g., [21, 23, 26], and they are likely to be affected by the biases in a similar way as plain VSM. Threats to construct validity relates to the suitability of our evaluation metrics. We make use of MAP, HIT@N, and MRR which are popular metrics that have been used in many past bug localization studies [20, 21, 26]. We also perform two widely used statistical tests (i.e., Mann-Whitney-Wilcoxon test [17] and Fisher exact test [11]), and compute one widely used effect size measure (i.e., Cohen's d [10]).

Table 15: Results of Mann-Whitney-Wilcoxon Test and Cohen'd Computation for MRR. (F-P) = Fully Localized vs. Partially Localized. (P-N) = Partially Localized vs. Not Localized. (F-N) = Fully Localized vs. Not Localized.

Bias Type	Project		p-value	d
Bias 1	HTTPClient		0.6667	0.241
	Jackrabbit		0.7855	0.050
	Lucene-Java		0.7336	0.043
Bias 2	HTTPClient	(F-P)	0.5465	0.142
		(P-N)	**0.0008925**	0.364
		(F-N)	**0.0003381**	0.634
	Jackrabbit	(F-P)	0.075	0.128
		(P-N)	**$<$ 2.2e-16**	**1.421**
		(F-N)	**$<$ 2.2e-16**	**0.962**
	Lucene-Java	(F-P)	0.2024	0.097
		(P-N)	**8.201e-08**	**0.944**
		(F-N)	**3.805e-06**	0.775
Bias 3	HTTPClient		0.6464	0.163
	Jackrabbit		0.9404	0.088
	Lucene-Java		0.7449	0.137

7. RELATED WORK

In this section, we describe studies that analyze bias in software engineering and IR-based bug localization studies. Our survey here is by no means complete.

7.1 Bias in Software Engineering

Many software engineering studies are highly dependant on data stored in software repositories. However, the dataset is not always clean, which means it might contain bias. A set of research work have shown that such bias in a dataset might impact software engineering studies, e.g., [7, 12, 13, 14, 15, 19]. We highlight some of them especially closely related ones below.

Antoniol et al. noted that bug tracking system not only maintains reports of bugs, but also other issue reports, such as reports that contain feature enhancement requests [4]. Later, Herzig et al. studied misclassified bug reports of multiple projects by manually checking around 7,000 issue reports [12]. They found that around 33.8% of their sampled bug reports are misclassified and such bias affects bug prediction techniques. Bird et al. investigated whether the bugs sampled based on commit logs are fair representation of the full set of fixed bugs or not [7].

Kawrykow and Robillard observed that many changes to source code corpus contain non-essential modifications, such as renaming variable and adding comments, and thus might cause bias to techniques that analyze version control repositories [14]. Thung et al. extended Kawrykow and Robillard's approach to detect root causes of bugs from commits [22]. Recently, Herzig and Zeller investigated the impact of tangled code changes, which are multiple changes for different tasks inside a single commit [13]. Nguyen et al. analyzed 1,296 bug fixing commits and found that more than 10% of the changed files are non-buggy files [19].

This work, especially the first research question, is an extension of our previous short paper that analyzes how issue report misclassification affects bug localization [15]. For our first research question, we extend the previous work by performing a per-project analysis instead of all-project analysis, and by computing Cohen's d to check if the impact of misclassification is substantial or not. In the previous work, we find that bias 1 significantly impacts bug localization results. In this study, we find that bias 1 only significantly impacts bug localization results for one out of the three projects. The other two research questions are newly proposed in this work.

7.2 IR-based Bug Localization Approaches

There are many IR-based bug localization approaches that retrieve source code files that are relevant to an input bug report [18, 20, 21, 23, 26]. Rao and Kak conducted a comparative study on the performance of a number of general IR models on bug localization task [20]. They found that simple text models such as Vector Space Model (VSM) and Smoothed Unigram Model (SUM) perform better than more sophisticated models such as Latent Dirichlet Allocation (LDA). Since then, a number of works have been done to improve the effectiveness of standard IR models by considering more information, applying advanced techniques, and refining queried bug reports.

Zhou et al. proposed an extended vector space model named rSVM to localize bug reports by leveraging information from similar bug reports [26]. Saha et al. made use of code structure information retrieved from source code (e.g, whether a word is used as a class name or a variable name), and bug report structure (e.g., whether a word is appeared in the title or description filed of a bug report) to improve the effectiveness of IR-based bug localization [21]. Based on the assumption that a bug report and its relevant code files share several latent technical aspects, Nguyen et al. developed a customized topic model approach named BugScout to localize bug reports [18]. Recently, Wang et al. proposed an integrated bug localization approach by considering multiple resources (i.e, version history, similar bug reports, and structure information) [23].

In this work, we focus on potential biases that might impact bug localization techniques. Our study highlights several steps that researchers need to take to clean up datasets used to evaluate the performance of bug localization techniques.

8. CONCLUSION AND FUTURE WORK

Many studies have proposed IR-based bug localization techniques to aid developers in finding buggy files given a bug report. These studies often evaluate their effectiveness on issue reports marked as bugs in issue tracking systems, using as ground truth the set of files that are modified in commits that fix each bug. However, there are several potential biases that can impact the results of these bug localization studies. Firstly, issue reports marked as bugs in issue tracking systems might not be bugs due to errors in the reporting and classification process. Secondly, bug reports might already be localized, i.e., they might explicitly mention the buggy files, which obviates the need to run localization on these bug reports. Thirdly, files modified as part of a bug fix commit might not be buggy, i.e., their modifications only involve cosmetic changes such as declaring empty variable, changing comments and so on. Our study analyzes the impact of these potential biases on bug localization results. Our empirical study highlights the following results:

1. Wrongly classified issue reports do not statistically significantly impact bug localization results on two out of the three projects. They also do not substantially impact bug localization results on all three projects (effect size < 0.2).

2. (Already) localized bug reports statistically significantly and substantially impact bug localization results (p-value < 0.05 and effect size > 0.8).

3. Existence of non-buggy files in the ground truth does not statistically significantly or substantially impact bug localization results (effect size < 0.2).

Our findings suggest that future bug localization researchers need to at least remove (already) localized bug reports from their evaluation dataset since such reports have significant and substantial impact on the performance of bug localization techniques.

As a future work, we plan to investigate more bug reports from additional systems to reduce the threats to external validity. We also plan to investigate additional biases that might affect bug localization studies.

Acknowledgement

We would like to thank Kim Herzig, Sascha Just, and Andreas Zeller for making the dataset used in their work [12] publicly available.

Dataset

Our dataset is made publicly available at `https://github.com/smusis/buglocalizationbiases`.

9. REFERENCES

[1] HTTPClient. `http://hc.apache.org/httpcomponents-client-ga/index.html`.

[2] Jackrabbit. `https://jackrabbit.apache.org/`.

[3] Lucene-java. `http://lucene.apache.org/`.

[4] G. Antoniol, K. Ayari, M. Di Penta, F. Khomh, and Y.-G. Guéhéneuc. Is it a bug or an enhancement?: a text-based approach to classify change requests. In *Proceedings of the 2008 conference of the center for advanced studies on collaborative research: meeting of minds*, page 23. ACM, 2008.

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

[6] A. Bachmann, C. Bird, F. Rahman, P. T. Devanbu, and A. Bernstein. The missing links: bugs and bug-fix commits. In *FSE*, 2010.

[7] C. Bird, A. Bachmann, E. Aune, J. Duffy, A. Bernstein, V. Filkov, and P. Devanbu. Fair and balanced?: bias in bug-fix datasets. In *Proceedings of the the 7th joint meeting of the European software engineering conference and the ACM SIGSOFT symposium on The foundations of software engineering*, pages 121–130. ACM, 2009.

[8] 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 *ESEC/FSE*, 2009.

[9] T. F. Bissyandé, F. Thung, S. Wang, D. Lo, L. Jiang, and L. Réveillère. Empirical evaluation of bug linking. In *CSMR*, 2013.

[10] J. Cohen. *Statistical Power Analysis for the Behavioral Sciences.* Hillsdale: Lawrence Erlbaum, 1988.

[11] R. A. Fisher. On the Interpretation of ÏĞ2 from Contingency Tables, and the Calculation of P. *Journal of the Royal Statistical Society*, 85:87–94, 1922.

[12] K. Herzig, S. Just, and A. Zeller. It's not a bug, it's a feature: How misclassification impacts bug prediction. In *Proceedings of the 2013 International Conference on Software Engineering*, pages 392–401. IEEE Press, 2013.

[13] K. Herzig and A. Zeller. The impact of tangled code changes. In *Mining Software Repositories (MSR), 2013 10th IEEE Working Conference on*, pages 121–130. IEEE, 2013.

[14] D. Kawrykow and M. P. Robillard. Non-essential changes in version histories. In *Proceedings of the 33rd International Conference on Software Engineering*, pages 351–360. ACM, 2011.

[15] P. S. Kochhar, T.-D. Le, and D. Lo. It's not a bug, it's a feature: Does misclassification affect bug localization? In *MSR*, 2014.

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

[17] H. B. Mann, D. R. Whitney, et al. On a test of whether one of two random variables is stochastically larger than the other. *The annals of mathematical statistics*, 18(1):50–60, 1947.

[18] A. T. Nguyen, T. T. Nguyen, J. Al-Kofahi, H. V. Nguyen, and T. N. Nguyen. A topic-based approach for narrowing the search space of buggy files from a bug report. In *Automated Software Engineering (ASE), 2011 26th IEEE/ACM International Conference on*, pages 263–272. IEEE, 2011.

[19] H. A. Nguyen, A. T. Nguyen, and T. N. Nguyen. Filtering noise in mixed-purpose fixing commits to improve defect prediction and localization. In *Software Reliability Engineering (ISSRE), 2013 IEEE 24th International Symposium on*, pages 138–147. IEEE, 2013.

[20] 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*, pages 43–52. ACM, 2011.

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

[22] F. Thung, D. Lo, and L. Jiang. Automatic recovery of root causes from bug-fixing changes. In *Reverse Engineering (WCRE), 2013 20th Working Conference on*, pages 92–101. IEEE, 2013.

[23] S. Wang and D. Lo. Version history, similar report, and structure: Putting them together for improved bug localization. In *ICPC*, 2014.

[24] S. Wang, D. Lo, and L. Jiang. Understanding widespread changes: A taxonomic study. In *CSMR*, pages 5–14, 2013.

[25] R. Wu, H. Zhang, S. Kim, and S.-C. Cheung. Relink: recovering links between bugs and changes. In *FSE*, 2011.

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