On the Relationship between Bug Reports and Queries for Text Retrieval-based Bug Localization

Chris Mills · Esteban Parra · Jevgenija Pantiuchina · Gabriele Bavota · Sonia Haiduc

Abstract As societal dependence on software continues to grow, bugs are becoming increasingly costly in terms of financial resources as well as human safety. *Bug localization* is the process by which a developer identifies buggy code that needs to be fixed to make a system safer and more reliable. Unfortunately, manually attempting to locate bugs solely from the information in a bug report requires advanced knowledge of how a system is constructed and the way its constituent pieces interact. Therefore, previous work has investigated numerous techniques for reducing the human effort spent in bug localization. One of the most common approaches is Text Retrieval (TR) in which a system's source code is indexed into a search space that is then queried for code relevant to a given bug report. In the last decade, dozens of papers have proposed improvements to bug localization using TR with largely positive results. However, several other studies have called the technique into question. According to these studies, evaluations of TR-based approaches often lack sufficient controls on biases that artificially inflate the results, namely: misclassified bugs, tangled commits, and localization hints. Here we argue that contemporary evaluations of TR approaches also include a negative bias that outweighs the previously identified positive biases: while TR approaches expect a natural language query, most evaluations simply formulate this query as the full text of a bug report. In this study we show that highly performing queries can be extracted from the bug report text, in order to make TR effective even without the aforementioned positive biases. Further, we analyze the provenance of terms in these highly

C. Mills · E. Parra · S. Haiduc
Florida State University, 600 W College Ave, Tallahassee, Florida, 32306
Tel.: +1-850-300-0283
E-mail: cmills@cs.fsu.edu, parrarod@cs.fsu.edu, shaiduc@cs.fsu.edu

J. Pantiuchina · G. Bavota
Università della Svizzera italiana, Via Giuseppe Buffi 13, 6900 Lugano, Switzerland E-mail: jevgenija.pantiuchina@usi.ch, gabriele.bavota@usi.ch

performing queries to drive future work in automatic query extraction from bug reports.

Keywords Bug Localization · Query Formulation · Text Retrieval

1 Introduction

Bug localization techniques are used to identify source code components that are likely to be responsible for a given bug. These techniques represent an important aid to reduce the time and effort spent on bug fixing activities. For this reason, many researchers defined various bug localization approaches, with Text Retrieval (TR) techniques playing a major role in this context (Dit et al., 2013). The basic idea in TR is that the user (usually, a software developer trying to localize the bug) formulates a natural language query describing the observed bug. The query is then run through a TR engine, which returns a ranked list of code components (*e.g.,* classes or methods, depending on the desired granularity), containing the most relevant results (*i.e.,* the components likely related to the bug) in the top-most positions.

Despite the many successful applications of TR-based approaches to bug localization, recent research questioned their actual usefulness by pointing to major weaknesses in the way these techniques are evaluated (Kochhar et al., 2014). Before describing these weaknesses, let us briefly summarize the four main steps behind the most used design adopted for the empirical evaluation of TR-based bug localization techniques. First, a set of *fixed* bug reports (*i.e.,* bug reports related to bugs that have already been fixed by developers) is collected by mining the issue tracking system of the subject software. Second, for each collected bug report, the code components impacted by the bug fixing activity (*i.e.,* modified/deleted methods or classes, depending on the granularity of the approach) are identified by analyzing the versioning system and/or by downloading the patch attached to the bug report. This allows the creation of a ground truth reporting what the relevant code components are for a given bug report. Third, the text of the bug report is used to generate a query, usually composed by the report's title and description. This query is used to simulate a user-formulated query that is then provided as input to the TR-based technique and run against the corpus of documents represented by the code components of the system. Finally, the ranked list of code components returned by the TR approach and the previously defined ground truth are used to compute performance metrics assessing the quality of the approach.

Three main biases have been identified in the assessment of TR-based techniques using the described design (Kochhar et al., 2014). First, using misclassified bugs, *e.g.,* issues that have been classified as bugs, when in fact they represent new features. Second, using bloated ground truths that include code changes irrelevant to a bug fix. Third, using bug reports that explicitly point to a code location such as a code snippet, file or class name, etc., which we refer to as *localization hints*. Previous work (Kochhar et al., 2014) showed that while the first two do not significantly impact the evaluation, bug reports

containing localization hints have a major impact on TR results. Other work (Wang et al., 2015) further analyzed the bias introduced by three types of localization hints: program entity names, test cases, and stack traces. Of the three, they found that program entity names significantly boost retrieval results.

Summarizing, these studies highlight the fact that *TR-based approaches perform very well when the bug report description already includes the bug location*, by listing all (or some) of the buggy code components. *However, in such scenarios TR-based bug localization techniques are not needed in the first place, as the bug has already been localized* (Kochhar et al., 2014). On the other hand, *when hints to localize the bug are not found in the bug report description* (*e.g.,* when the bug report is written by a user of the system that lacks knowledge of the system's technical implementation), the studies showed that *the performance of TR-based bug localization techniques significantly drops, thus questioning their usefulness in this scenario as well.*

While we agree on the performance inflation provided by positive biases identified in previous work, most of the studies focusing on TR-based bug localization, including the ones questioning their validity, also include an additional bias that negatively affects the performance of the TR technique: they use the text of the bug report (*i.e.,* its title, description, or a combination of both) as the TR query, without considering alternatives. Since *TR effectiveness strongly depends on the quality of the formulated query* (Mills et al., 2017), the choice of query is crucial in ensuring the success of TR. Bug reports, however, can often be lengthy and contain "noise" (*i.e.,* words that do not efficiently describe the bug) (Chaparro and Marcus, 2016). We therefore conjecture that TR approaches have a lot more *potential* for bug localization than they have been given credit for, so long as they can be provided with an optimized query.

In this paper we present an empirical study providing new evidence on the true potential of TR bug localization approaches and the significant impact that optimizing queries can have on their effectiveness. Further, we show that given a bug report, we can often obtain an optimal query using only words selected from its vocabulary, even when localization hints are not present.

To perform this study, we first collected 1,037 bug reports from 15 open source systems used in previous bug localization experiments (Wang et al., 2015; Ye et al., 2015; Chaparro and Marcus, 2016). To account for the potential biases named above, we manually analyzed each bug report (BR) and its fixing commits. From the BR, we extracted two queries: one containing all terms in BR's title and description (Q_a), and one (Q_{nH}) obtained by removing all code-related terms possibly representing localization hints (*e.g.,* class/method/file names, stack traces, test cases, *etc.*) from Q_a. This allowed us to investigate the bias that the presence of localization information can have on TR results, as discussed in prior work (Kochhar et al., 2014; Wang et al., 2015). Then, in the fixing commit of BR, we also manually analyzed each modified file to verify that two conditions were met. First, we verified that the change was actually made to fix the bug rather than being the result of a tangled commit (Herzig and Zeller, 2013). Second, we checked that the change actually modified the functionality of the system and was not purely aesthetic (*e.g.,* changes to

comments or code formatting). This manual verification of the changes ensured a reliable ground truth for our experiments, free of the misclassified bugs and bloated ground truth biases previously identified (Kochhar et al., 2014). With this extracted data, we show that it is almost always possible to formulate an effective query returning relevant results in the top positions of the ranked result list by only using the terms present in either Q_a or in Q_{nH}. Therefore, we are able to derive an effective query in either case: with or without localization hints in the BR.

Our study differs from previous work in that we are interested in establishing the potential of TR for bug localization by finding the most effective query that can be extracted from the BR and evaluating the performance of that query, rather than considering the default query composed of the entire bug title and/or description. Finding this query by brute force, however, would require testing all possible queries of any length that can be obtained from the BR's vocabulary. For example, assuming a BR composed of n distinct terms, the number of possible queries to test is 2^n, representing all queries of any length that can be generated using the n terms in the bug report, without considering word order. Given that our Q_{nH} and Q_a queries have on average 24 and 40 distinct terms, respectively, this would mean running between 1.7E+7 and 1.1E+12 queries for each bug report, which is computationally infeasible.

For this reason, we devised a Genetic Algorithm (GA) able to converge towards an optimal query obtained from a bug report vocabulary, knowing *a priori* the ground truth for the query. While our GA *is not meant to be used in a real bug localization scenario, where the ground truth is unknown*, it represents a needed tool to run our large-scale study and provide evidence on the potential of TR bug localization with optimal queries. We also analyze the provenance of terms in the obtained GA queries to drive future work in automatic query extraction from bug reports.

This work builds upon our previously published paper (Mills et al., 2018) establishing the applicability of TR-based approaches to bug localization, and extends it in several ways:

- In our previous work, we investigated a single fitness function to drive our GA through the formulation of near-optimal queries. In contrast, here we experiment with two additional fitness functions. While the original work focused only on effectiveness, which gauges a query's ability to find *a single* relevant document, this paper also considers average precision and recall, which represent a query's ability to find *all* of the relevant documents.
- In our previous work, we investigated a single search engine to drive our GA through the formulation of near-optimal queries. In contrast, in this paper, we investigate how much the particular search engine impacts the results of the experiments by using two different versions of the Lucene search engine. The selected versions employ different ranking functions, thus allowing us to measure the impact of the search engine on the possibility to formulate high-quality queries.
- Our previous work focused solely on determining if TR approaches are still viable solutions to bug localization or if they are suffering from crippling

biases that make them no longer worthy of study. More than showing that TR-based approaches are relevant, this paper also seeks to understand how near-optimal queries are derived from a bug report, to provide lessons learned that we believe will be useful to devise the next generation of TR query formulation techniques.
– This paper aims to increase the generalizability of the results by expanding the set of queries in our dataset through the addition of 200 new bug reports (+24% as compared to our previous work) from two additional open source systems, namely, Birt and Tomcat.

2 Background and Related Work

2.1 TR Approaches to Bug Localization

TR allows users to search in a corpus (*i.e.,* a set of text documents) for documents relevant to a given text-based query. In bug localization, developers must find buggy code given a bug report. Both the system's code and the bug report can be stored as textual documents. Therefore, TR perfectly fits in the bug localization process: the source code can be seen as the document corpus (*e.g.,* by considering each class/method/file as a different textual document), while the text from a bug report can be used to formulate a query aimed at retrieving code documents that are semantically similar to that text. The underlying assumption of TR-based bug localization is that code components that contain terms similar to the formulated query are likely related to the observed bug, and thus are recommended for inspection.

Many studies have focused on the application of TR to the bug localization domain (Marcus et al., 2004; Lukins et al., 2008; Rao and Kak, 2011), on providing tool support (Poshyvanyk et al., 2005; Zhao et al., 2006; Linstead et al., 2008; Savage et al., 2010; McMillan et al., 2011; Zhou et al., 2012; Shepherd et al., 2012) and improvements over standard TR techniques (Lukins et al., 2010; Saha et al., 2013; Wang et al., 2014; Wang and Lo, 2014). Due to the breadth of existing research and space constraints, we direct the interested reader to a survey (Dit et al., 2013), which contains a detailed account of many approaches to bug localization, including TR-based techniques. Alternatively, a more recent study on feature localization (Razzaq et al., 2018), a tangential software task focused on locating features rather than bugs, provides additional context for these approaches. Despite this immense body of work, there are still doubts within the software engineering community regarding the applicability of TR-based approaches to bug localization. Therefore, rather than introducing a new approach or application of TR to bug localization, our study focuses on the general potential of these techniques and how they are evaluated based on information extracted from bug repositories.

2.2 Potential Biases in Empirical Evaluations of TR-based Bug Localization

A previous study (Kochhar et al., 2014) empirically investigated three potential biases: bug misclassification, bloated golden sets, and localized bugs reports. The researchers found that localized bug reports (*i.e.,* bug reports which already list in their text the code location where the bug is present) lead to significantly better results for TR-based bug localization, while no significant impact was observed for the other two potential biases. This finding questions the usefulness of TR-based bug localization, as this process is actually needed only when localization hints are **not** present in the bug report, *i.e.,* exactly when their performance significantly drops (Kochhar et al., 2014). We partially replicate this study by investigating TR performance on localized *vs* unlocalized bug reports in our dataset. Our study is conducted on a larger set of software projects and above all it shows that the chosen query matters more than the presence of localization information.

Further, another study (Kawrykow and Robillard, 2011) reported the presence of changes unneeded to fix a bug in bug-fixing commits (*e.g.,* renaming a variable) as a possible source of bias in the evaluation of bug localization techniques. Indeed, these changes artificially increase the number of code components that appear relevant to a bug fix but are actually unrelated. This is the same form of bias defined by (Herzig and Zeller, 2013) as “tangled commits", meaning commits containing changes addressing a specific bug mixed with unrelated changes (*e.g.,* refactoring). If TR approaches are evaluated on these commits, their performance might be artificially boosted by the fact that there are many more “relevant" files to be found in the search space, even though finding some of these files would not practically assist with localizing the bug. In order to remove such a bias from our study, we manually analyzed all the changed files in each bug fix commit and removed any file whose changes were not directly related to the bug being fixed.

Another study (Wang et al., 2015) provides a detailed analysis of the impact that localized bugs have on evaluating TR-based bug localization. The authors consider different types of *identifiable information* that localize a bug explicitly: program entity names, stack traces, and test cases. They found that the presence of program entity names significantly improves the performance of TR-based bug localization. Our study also addresses the impact of localization hints on TR performance, but does so in a different way. First, Wang *et al.* classify bug reports as containing/not containing localization hints on the basis of the information that they report, and then compare the performance of TR-based bug localization on these disjoint sets of bugs. Instead, we manually derive from each bug two versions: one containing the complete title and description, and one from which we manually remove any reference to code components (*i.e.,* any possible localization hint). This allows us to compare the performance of TR-based bug localization with and without localization hints between the same set of bugs, thus removing conflating variables that are introduced by comparing disjoint bug sets. Through this analysis, we show

that the formulated query is more important than the presence or absence of localization hints in ensuring the success of TR in bug localization.

Finally, according to (Bettenburg et al., 2008), the inclusion of identifiable information is considered to be important when writing a good bug report, yet developers answering a survey indicated that few bugs contain error messages (53%), code examples (36%), or test cases (56%). Moreover, developers who took the survey indicated that the biggest detracting factors in bug reports were unstructured, lengthy text and non-technical language. This further suggests that localization techniques should focus on optimizing performance in these difficult situations. Through investigating queries in the Q_{nH} set, this study also investigates whether sufficient information to localize a bug exists even for bugs composed of strictly unstructured, natural language. This concept is related to other work which has sought to quantify the quality of a query used for a software engineering task (Haiduc et al., 2012; Mills et al., 2017) and derive a more effective query when required.

2.3 Query Reformulation for TR-based Bug Localization Approaches

In the event that a query leads to poor results, there have been numerous techniques devised to reformulate the query (Shepherd et al., 2007; Haiduc et al., 2013; Roldan-vega et al., 2013; Rahman and Roy, 2017; Rahman and Roy, 2017; Lawrie and Binkley, 2018) using either expansion (Carpineto and Romano, 2012) (*i.e.,* adding additional terms to broaden the query) or reduction (Haiduc et al., 2013; Chaparro and Marcus, 2016) (*i.e.,* removing words unlikely to contribute to the inherent meaning of the query, in order to reduce noise).

A recent study by (Chaparro and Marcus, 2016) empirically quantified the improvement in verbose queries achieved by eliminating words that negatively impact effectiveness by removing up to six “noisy" terms from the query. These terms are identified through a brute-force approach. In a subsequent study (Chaparro et al., 2017a), Chaparro *et al.* manually reduced noisy, ineffective queries to reformulated queries that contain only terms that describe *observed behaviors* and find that the reformulated queries have much-improved performance. In an extension, (Chaparro et al., 2019) showed that selecting the steps to reproduce or the expected behavior information (when available) along with the bug title and the observed behavior leads to higher performance. (Chaparro et al., 2017b) identified the presence of 154 discourse patterns that indicate the presence of information describing expected behavior and steps-to-reproduce in bug report descriptions and introduced a set of strategies to identify them.

Chaparro *et al.*'s studies are the closest related to the work presented in this paper. Therefore, we include a *post-hoc* analysis of the effectiveness of queries common to both studies in Section 4. Further, we also incorporate these concepts in our provenance analysis, specifically identifying which type of terms remain in near-optimal queries.

Some previous work has also focused on reformulating queries based on the quality of a bug report. (Rahman and Roy, 2018) introduce BLIZZARD,

a query reformulation technique for bug location which evaluates the quality of a bug report, and applies different reformulation strategies on it based on its quality. BLIZZARD uses pseudo-relevance feedback to improve queries by using the terms in the top-N results retrieved after applying TR approaches.

(Kim and Lee, 2019) present a query reformulation technique that expands upon BLIZZARD and reformulates poor quality queries by expanding the query using the content of textual attachments to the bug report (e.g., patches, stack traces), removing terms that express emotion, executing the resulting query using TR, and applying relevance feedback to reformulate a new query. By using information from patch attachments, Kim *et al.*'s approach turns unlocalized bug reports into localized bug reports, consequently improving the results of TR-based bug localization.

Work by (Lawrie and Binkley, 2018) applied a sequence-to-sequence neural network model to generate summaries of the bug reports used in this study to be used as queries for TR-based bug localization. However, those summaries were unable to achieve performance close to the near-optimal queries presented here. This provides further evidence that a larger-scale data collection effort and advanced techniques to mitigate the need for immense amounts of training data should be the focus of future work.

Different from previous studies, we leverage a genetic algorithm which allows us to perform any reformulation (using the bug report) that converges to an improved, near-optimal query using a performance metric as a cost function. From this perspective, a major contribution of our study is empirical evidence that more complex techniques leveraging external sources of context (Youm et al., 2017) are not needed to optimize queries in most cases. Furthermore, our results show that while observable behaviors help with refining a query based on bug report text, there exist even more effective queries that can be derived from the bug report. That is, there exists some combination of terms from the bug report text that serve as an effective query for localization that may or may not have any relationship to the observable behavior of the bug.

3 Empirical Study Design

In this paper we investigate TR-based approaches for bug localization from two perspectives. First, we show that TR is able to effectively support bug localization despite recently raised *positive* biases that suggest previously reported performance is artificially inflated. We do this by exposing a larger *negative* bias in previous work that makes TR performance appear much worse than it is in reality: most evaluations use the full bug report as a query. To show the effects of this bias we use a genetic algorithm (GA) to formulate near-optimal queries. Using these queries rather than the full bug report, we show that TR achieves high performance even when controlling for the aforementioned positive biases. Second, we analyze near-optimal queries to determine the provenance of their terms in the components of a bug report (*i.e.,* title, description, or a combination of the two), files from the golden set for the

bug, and bug report text related to specific types of technical information such as observed/expected behavior (Chaparro et al., 2017a). The latter analysis serves as an important step toward understanding how near-optimal queries are derived from bug report text by a GA in an effort to translate that process into one understandable and implementable by humans.

3.1 Research Questions

We start by investigating the potential effectiveness of TR-based bug localization assuming the ability to formulate an optimal query from the bug report vocabulary. In particular, in the first part of our study, we address the following research questions:

RQ_1: What is the effectiveness of TR-based bug localization techniques when using the whole bug report text as a query? This is a preliminary research question in which we establish a baseline by studying the performance of out-of-the-box TR-based bug localization techniques when using the whole text contained in the bug report (*i.e.,* a concatenation of its title and description) as a query, as usually done in empirical evaluations of these techniques (Kochhar et al., 2014). RQ_1 serves as a term of comparison for our second research question (RQ_2), in which we study what the *potential* effectiveness of these techniques could be, given the ability to formulate an "*optimal query*" starting from the bug report vocabulary. As part of this research question we also present a differentiated, partial replication of the work by (Kochhar et al., 2014), in which we analyze the impact that localization hints in the bug report text have on the performance of TR-based bug localization.

RQ_2: What is the effectiveness of TR-based bug localization techniques when using an optimal query selected from the vocabulary of the bug report? In this research question, we study what the potential effectiveness of out-of-the-box TR-based bug localization techniques truly are. Specifically, we devised an experimental design allowing us to formulate a "*near-optimal query*" from a bug report (*i.e.,* the most effective query that can be derived by a GA from solely the vocabulary of the bug report). By using near-optimal queries, the *negative* bias of poor quality query selection is removed and allows us to determine the potential of TR approaches controlling for both types of common biases. This research question can be further divided into sub-research questions based on the overall goal of the optimization: finding a single or multiple relevant documents in the top part of the ranked results.

$RQ_{2.1}$: Can queries based on bug report text be optimized to find a single relevant class?

$RQ_{2.2}$: Can queries based on bug report text be optimized to find multiple relevant classes?

RQ_3: Does changing TR engine implementation impact query optimization?

For RQ_2, queries were optimized using Lucene 2.9.4. In this research question, we perform the same experiments used to answer $RQ_{2.1}$, but with Lucene

8.2.0, which uses a different ranking function. In both cases, we use the default similarity metric class for computing document ranks in the result set. For 2.9.4, the default similarity is based on tf-idf[1]. Lucene 8.2.0, on the other hand, uses a similarity based on BM25[2]. This research question measures how dependent optimization is on the specific TR engine implementation. We use two out-of-the-box approaches with minimal customization over more complex, state-of-the-art approaches to show that in light of optimization, more complex TR engines are not strictly required to obtain good results. To that end, we also replicate the experiments used to answer RQ_1 using Lucene 8.2.0.

As will be shown later, the results of the first three research questions show that near-optimal queries are on average much shorter than their unoptimized counterparts. However, it is difficult to gain insights on the origin of the terms in near-optimal queries from such a finding. Therefore, in the second part of our study we aim at understanding the provenance of query terms that appear in near-optimal queries. That is, we analyze where the most effective query terms in a bug report come from, and whether there are patterns that can be observed to assist with formulating near-optimal queries without *a priori* knowledge of the ground truth. We consider provenance from three different perspectives. First, we analyze where in a bug report the most useful query terms come from: its title, description, or both. Second, we investigate the proportion of query terms that overlap between the bug report text and the vocabulary extracted from the files that were changed to fix the bug. Third, we compute the proportion of near-optimal queries containing terms coming from sentences in the bug report describing the expected and observed behaviors (EB and OB, respectively) and the steps to reproduce the bug (S2R). More specifically, we seek to answer the following research questions:

RQ_4: Where in a bug report do effective near-optimal query terms appear?

By answering this research question, we can begin to observe patterns that could eventually lead to automatically formulating better queries through boosting terms that are most likely to appear in a near-optimal query and/or by penalizing those least likely to contribute to a near-optimal query.

RQ_5: What is the term overlap between near-optimal queries and changed files in the associated bug report's golden set? The intent of this investigation is to determine the overlap existing between terms in the bug reports and those in the related code components *after localization hints are removed from the bug report* (*i.e.,* excluding the names of classes, exceptions, *etc.*). Effectively, this analysis illustrates the effects of proper alignment between identifiers within the code and domain concepts expressed in bug reports on near-optimal query formulation.

RQ_6: What is the overlap between near-optimal query terms and the EB, OB, and S2R as described in the bug report? Previous research

[1] `https://lucene.apache.org/core/2_9_4/api/core/org/apache/lucene/search/Similarity.html`

[2] `https://lucene.apache.org/core/8_2_0/core/org/apache/lucene/search/similarities/BM25Similarity.html`

Table 1 The fifteen systems in our dataset, with their corresponding number of bug reports, average number of files changed per bug report, and average query lengths

Project	**# Original Bugs**	**# Cleaned Bugs**	**Avg. Changed Files**	**Avg. Q_a Size**	**Avg. Q_{nH} Size**
AspectJ	286	188	2.50	224.35	49.10
Birt	100	97	2.94	119.16	53.20
BookKeeper	40	24	3.96	83.63	24.83
Derby	96	49	2.64	190.41	44.14
JodaTime	9	7	1.00	126.57	46.14
Lucene	34	32	2.09	232.90	27.43
Mahout	30	25	3.12	97.72	34.24
OpenJpa	18	16	2.06	166.06	41.06
Pig	48	36	1.53	117.44	25.00
Solr	55	45	2.22	87.80	42.93
SWT	98	85	1.68	100.56	42.80
Tika	23	21	2.38	74.33	26.00
Tomcat	100	86	1.43	120.71	41.13
ZooKeeper	80	78	1.91	106.09	38.14
ZXing	20	14	1.07	123.31	73.23
Total	1037	803	2.17	131.40	40.62

(Bettenburg et al., 2008; Chaparro et al., 2017a) has shown that specific technical language like EB, OB, and S2R is instrumental in assisting developers to take a bug report and find relevant buggy code within the system. However, no previous work has looked at how terms used to describe these bug features overlap with terms in near-optimal queries derived from the bug report text. Understanding which of these bug features contribute the most terms to a near-optimal query provides insight into which type of feature is most helpful for linking bug reports and buggy code via a TR-based approach.

For research questions RQ_4 to RQ_6 we used the near-optimal queries generated using Lucene 2.9.4 as the search engine.

3.2 Data Collection

We used a set of 803 bug reports manually extracted and verified from 15 software systems. To obtain these bug reports, we initially started from datasets that were used in three previous studies to analyze the effectiveness of TR-based bug/feature localization techniques (Wang et al., 2015; Ye et al., 2015; Chaparro and Marcus, 2016). We then manually inspected each bug report in these datasets and their corresponding commit and further cleaned the data, as described below.

First, each commit associated with a bug report BR in the datasets was manually analyzed to verify the validity of the ground truth (*i.e.,* the set of files actually modified to fix the bug described in BR). This step was necessary to ensure that only valid files and bugs are being used in our study. In particular, two of the authors manually verified the data, each of them focusing on around half of the bug reports in the dataset and their fixes. Moreover, a third author double-checked the manual labeling done by the other two authors, in

Table 2 Changes unrelated to bug-fixing identified by the two evaluators

Change	Description	#Files	%Files
Added code only	A file was changed by only adding new code (*i.e.,* existing code was not modified/deleted)	395	46.58
Test code	Changes to the test code, not impacting the system's behavior	262	30.90
Refactoring	Changes do not affect the system's behavior (*e.g.,* renaming a variable)	74	8.73
Comments	Adding/removing/modifying code comments	72	8.49

order to identify any involuntary errors or misunderstandings, as well as to determine any cases in which additional screening or discussion was required. The authors first identified each BR's bug-fixing commit fix_{BR}, by looking in the project's versioning system for commit notes explicitly reporting the BR id (*e.g.,* `DERBY − 6150`). Note that `git` is used by all subject systems except JodaTime, which uses `SVN`; however, the process for JodaTime remains the same using analogous `SVN` features. The mapping between bugs and their respective fixing commits was possible for all the bugs included in our study. Since the presence of a bug id does not always guarantee that the commit contains only the needed bug fix (Kawrykow and Robillard, 2011), a manual inspection of the actual changes was needed to ensure that only files relevant to the bug are included in the ground truth. Therefore, the two authors manually inspected the changes performed in fix_{BR} by exploiting the `git show` command (or the equivalent `svn diff −c` command), and tagged each modified file as *true positive* (*i.e.,* the file has been actually modified to fix the bug reported in BR) or as *false positive* (*i.e.,* the file has been modified as the result of a tangled commit and/or the changes in the file do not indicate the fixing of a bug). For *false positive* changes, the authors also tagged the change with a reason why it was considered not relevant to fixing the bug; these tags are available as part of the replication package for our study (Mills, 2019). In total, 2,311 files have been modified in the bug-fixing commits, but only 1,154 of them were indeed found to be *true positives*. The most common reasons for which the two evaluators excluded modified files from the set of *true positives* are reported in Table 2. While the rationale for excluding changes related to refactoring and code comments is quite obvious, the other two "categories of changes" we excluded may need clarifications. For what concerns the "Test code" category, while the test code is certainly relevant in the context of bug-fixing activities (*i.e.,* to verify that the bug has actually been fixed), it is not the main target of bug localization, which usually focuses on production code. Regarding the exclusion of instances where only new code was added, we only excluded those scenarios in which a new class was established. This is because the granularity of this work is at the class level. Should we include these scenarios, we would be considering instances in which entirely new classes were created to resolve an issue, and therefore no bug localization technique would be able to locate these files, since they do not already exist in the code. Finally, while we do not list "tangled changes" in Table 2 as one of the main reasons for excluding files, note that all reasons listed in Table 2 are potential tangled changes (*e.g.,* a commit fixes a bug *and* refactors the code.

This manual verification process resulted in the exclusion of 234 bug reports from the 1,037 bugs found in the original datasets. Of those, 198 bugs were

excluded because no modified files were left for them in the *true positives* set after verification. An additional 36 bug reports were excluded due to insufficient information in the downloaded source files to reconcile the bug's golden set with the code used to construct the corpus (*e.g.,* package migrations with no documentation linking the old and new location, platform-specific code that was not available in the current source download, *etc.*). These cases arose most frequently for the ZXing project, which has been migrated between several version control systems. The final result was a set of 803 bugs that we used in our study.

Such a cleaning process of our dataset was needed to avoid the use of a *bloated ground truth* or *misclassified bugs* in our study (Kochhar et al., 2014). Table 1 shows the number of bugs before and after the manual verification process in each dataset we used.

Next, for each remaining bug report BR in our data, we extracted a basic query by concatenating BR's title and description. This type of query (from now on called Q_a, as it contains all the terms in the bug report) is the one most often used to automatically assess the effectiveness of TR-based bug localization techniques (Kochhar et al., 2014). In addition, we also manually extracted a second query (from now on, Q_{nH}) obtained from Q_a by removing all code-specific terms that could represent localization hints: package, class, method, and identifier names, stack traces, code snippets, file paths, fully qualified names, and version control URLs pointing to code locations. Note that for words embedded in a localization hint (*e.g.,* the word "pointcut" in the localization hint "IfPointCut"), the word itself is still considered relevant and kept. However, the complete localization hint (*i.e.,* "IfPointCut") refers to a specific class and is removed from both the bug title and description. In total, we extracted 1,606 queries from the 803 bug reports. Table 1 reports the systems we considered, the number of bug reports per system, and the average size of the extracted queries in number of words.

We then downloaded the source code of each system and constructed a corpus by considering each Java file as a document. We applied preprocessing to both the queries we previously extracted and the corpus documents in order to remove English stop words and reserved Java keywords, stem words to their root form, and split identifiers in the source code based on CamelCase and the underscore separator. Each preprocessed query was then run on its corresponding document corpus (*i.e.,* the code files of the related project) by using the `lucene`[3] implementation of the Vector Space Model (VSM).

VSM is an approach that represents each document in a corpus in an n-dimensional space, where each dimension represents a particular word. A document is represented as a vector within this "term space". To obtain the vector corresponding to a document, the document is split up into words. For each word in the document, a real number indicating its weight is placed in the corresponding dimension. A common way to represent the weight of a word for a document is *tf-idf* (Term Frequency-Inverse Document Frequency). Tf-idf is

[3] `https://lucene.apache.org/`

a function of the frequency of the word in the document and the number of documents in the corpus in which the term appears (Salton et al., 1975).

VSM with tf-idf is a popular TR approach in software engineering applications. A common method to evaluate how similar a document is to another document or to a query is to calculate the cosine of the angle between their vectors (Wong et al., 1985). In this work, we primarily use VSM with tf-idf because it is a simple but very commonly used TR approach in bug localization and by showing that queries based on the text of a bug report can be optimized for this approach implicitly shows that more advanced TR techniques could potentially do even better when given an optimized query. However, we also provide an analysis of the Lucene 8.2.0 implementation that uses a probabilistic document similarity based on BM25 in order to verify that our near optimal queries are not unique to a single TR-engine implementation.

The data collected until now are enough to answer RQ_1, RQ_2, and RQ_3. However, to analyze the provenance of the terms used in the near-optimal queries (*i.e.,* RQ_4, RQ_5, and RQ_5) we also had to extract some additional data. First, we tagged all terms in each bug report query based on their provenance, meaning whether the term appears in the bug report title, description, or in both (RQ_4). Second, we extracted a vocabulary consisting of every unique term in the preprocessed source code in the golden set of each bug (i.e., the classes that were fixed) to determine the overlap of terms in that vocabulary and each corresponding near-optimal query (*i.e.,* how many of the terms used in the near-optimal queries also appear in the code components relevant for the bug report – RQ_5). Third, we used a series of regular expressions provided in a previous study (Chaparro et al., 2017a) to identify the sentences in the bug report describing the expected and observed behaviors and the steps to reproduce the bug. This allows to identify terms used in the description of these information items (*i.e.,* EB, OB, and S2R) and therefore answer RQ_6.

3.3 Data Analysis

Next, we describe the data analysis performed to answer our research questions.

3.3.1 Effectiveness of TR-based Bug Localization (RQ_1, RQ_2, and RQ_3)

To answer RQ_1 we compare the performance of queries derived from the complete text of each bug report from each system before and after the manual removal of all localization hints. This allows us to directly measure the impact of localization hints on query performance. While similar studies have been conducted in the past (Kochhar et al., 2014; Ye et al., 2015; Wang et al., 2015), our study provides a one-to-one comparison between two versions of the same bug: one with and one without hints. Different from previous work that compared disjoint sets of bugs with and without localization hints, our methodology allows us to remove conflating variables such as bug type or severity that are not controlled for in these other studies.

We use a targeted set of metrics to evaluate queries that consider those biased in "favor" and "against" bugs requiring a large number of files to be modified. The *Effectiveness Score* and the *HITS@K* metrics are biased in favor of bug reports having a large number of relevant documents in the golden set. Further, statistically speaking, it is "easier" to find at least one file relevant to a bug report as the number of its relevant files increases. This is why (Wang et al., 2015) pointed out that, since it is unlikely that all files modified in the commit fixing a bug are actually relevant for the bug fix, these metrics tend to represent a threat to conclusion validity. For this reason, we also consider *MAP*, which takes all files marked as relevant into account. This means that for a bug report having many files associated with it, the *MAP* will be biased against that bug report in contrast to the other two metrics. In the following we formally define these four metrics:

- *Effectiveness* - The highest rank in the list of results of any relevant document (*i.e.,* Java file) in the golden set. The rationale is that given a single relevant file in the top results, developers can then easily navigate from it to the other relevant files using built-in IDE navigation. Because the effectiveness distributions in this study are highly skewed by outlying queries that perform particularly poorly, we report the median scores to provide a fair depiction of the overall effectiveness of queries in each project.
- *HITS@K* - For a set of queries run on a document corpus, it is the percentage of queries that retrieve a relevant file in the top K positions of the ranked list. For example, HITS@1 provides the percentage of queries that return a relevant file as the topmost result. In this study, similar to (Wang et al., 2015), we use HITS@1, HITS@5, and HITS@10.
- *Average Precision (AP)* - This is calculated as the mean of precision values at each k such that the document returned at position k is relevant.
- *Mean Average Precision (MAP)* - Mean average precision is the mean across all average precision values within a set of queries.
- *Recall* - The proportion of relevant documents retrieved to the total number of relevant documents.
- *Mean Reciprocal Rank (MRR)* - The reciprocal rank is the inverse of the rank of the first relevant document in a result set. MRR is the average of all reciprocal ranks within a set of queries.

It is also important to consider different types of metrics because bug localization is not the only software engineering task that has been partially automated by TR (Marcus and Antoniol, 2012). While a single relevant file is sufficient to perform bug localization in most cases, as a developer can easily move from one relevant file to another through IDE support for call graph navigation, this is not true for other tasks such as program comprehension and traceability link recovery. Effectiveness provides insight into how well a query retrieves a single relevant document, but other metrics such as MAP illustrate how well a query retrieves *multiple* relevant documents.

To further analyze data collected to answer RQ_1, we also statistically compare the TR-based bug localization performance (as assessed by the four metrics) when using Q_a *vs* Q_{nH} (*i.e.,* when localization hints are present/not

present in the bug report). While the Mann-Whitney U or Kolmogorov-Smirnov tests provide a mechanism to compare two potentially non-normal, independent distributions, neither of these tests handle frequent ties in compared distributions well. As such, these tests are not directly suitable for our case where, as will be shown in our results discussion, many queries are able to achieve a perfect effectiveness score of one. Therefore, we applied the Asymptotic General Independence test (Strasser and Weber, 1999) implemented in the `coin` *R* package. This is a generalized permutation test that uses random sampling to determine the independence of two distributions based on mean-differences. It is applicable to non-normal, independent, discrete distributions despite the presence of ties. We also assess the magnitude of the observed difference using Cliff's delta (d) effect size (Grissom and Kim, 2005), suitable for non-parametric data. Cliff's d ranges in the interval $[-1, 1]$ and is negligible for $|d| < 0.148$, small for $0.148 \leq |d| < 0.33$, medium for $0.33 \leq |d| < 0.474$, and large for $|d| \geq 0.474$. This statistical analysis is primarily intended to provide additional insight about the deviation of the distribution means. Significance in this context means that the distributions are independent, which suggests that the presence of localization hints fundamentally alters the distribution of the performance metric in question. Note that a lack of significance does not imply that the distributions are identical or even equivalent, we just lack support that they are independent of one another.

To address RQ_2, we still need to extract an "*optimal query*" from the vocabulary of each bug report without the use of localization hints. Given the vocabulary of BR without hints (*i.e.,* the Q_{nH} query) composed of n terms, it quickly becomes computationally infeasible to try all possible queries of any length that can be extracted from Q_{nH} in order to observe which one leads to the best results (this would result in running 2^n queries through the TR engine). For this reason, we instead opt for an approximation of the "optimal" query obtained using a single-objective Genetic Algorithm (GA) (Mitchell, 1998) that quickly converges towards a "near-optimal" query[4] starting from a given vocabulary (in our case, the terms in Q_{nH}).

The solution representation (chromosome) and the GA operators are defined as follows. Given a vocabulary composed of n terms, the chromosome is represented as an n-sized integer array, where the value of the i^{th} element equals 1 if that term is part of the formulated query, and 0 otherwise. The only constraint we set on the generated solutions is that the chromosome must contain at least one "1" (*i.e.,* the query must contain at least one term). The crossover operator is a one-point crossover, while the mutation operator randomly identifies a gene (*i.e.,* a position in the array), and modifies it by randomly assigning it to 0 or 1. This translates to removing/adding a term to the query. The selection operator is the roulette-wheel.

A major component of a single-objective GA is a fitness function that is minimized as a completion criteria for optimization. That is, optimization

[4] We use the term "near-optimal" since the GA will ultimately converge to a local optimal solution which may or may not be the global optima.

is complete after either the fitness function value for an iteration is below a provided threshold or after a maximum number of iterations are performed. For each RQ_2 sub-research question, we use a different quality metric as a fitness function for the GA to determine if different fitness functions result in different optimizations. For $RQ_{2.1}$ we use effectiveness, and for $RQ_{2.2}$ we use inverse average precision and inverse recall. Note that AP and recall are inverted because the fitness function must decrease as the query becomes increasingly optimized. That is, while effectiveness has an inherently inverse relationship with query performance, AP and recall do not. In all of these cases, the fitness function is possible thanks to the fact that the ground truth is known for each bug report. In essence, we look for the query optimizing retrieval performance as represented by either effectiveness, AP, or recall, which are often used to evaluate TR-based approaches for software engineering tasks (Dit et al., 2013).

Our GA is built on top of the `jmetal` framework[5] and uses the following parameter configuration: *population size*: 500; *maximum number of generations*: 30,000; *crossover probability*: 0.9; *mutation probability*: $1/n$ (where n is the number of terms in the bug report). Also, given that a run of the GA involves some elements of randomness (*e.g.,* in the roulette-wheel selection), we run the algorithm ten times and average the results to mitigate threats to validity introduced by chance.

We acknowledge that in many cases, modern automatic query formulation techniques would not be able to derive such an optimal query without knowing the ground truth *a priori*. However, our goal in this study is not to present the GA as a query formulation technique. Instead, we seek to identify a near-optimal query that can be formulated from a bug report in order to empirically assess the *potential* of TR-based bug localization. The existence of a query with sufficient performance despite the absence of positive bias suggests that those biases do not preclude TR-based approaches from being worthy of continued study. Finally, note that the GA is given the vocabulary from Q_{nH} after preprocessing all queries, therefore ensuring the same treatment is applied to both queries and corpus documents.

The same performance metrics/statistical analyses used in RQ_1 are also computed for RQ_2, where we compare the performance of Q_a and Q_{nH} with the performance of the near-optimal queries QGA_a and QGA_{nH}, obtained by the GA starting from the vocabulary present in Q_a and in Q_{nH}, respectively.

For RQ_3 we analyze changing the Lucene implementation from version 2.9.4, which uses a similarity based on tf-idf, to version 8.2.0, which uses a BM25 similarity for document ranking. To perform this analysis we compute the same baseline given for RQ_1 and compare that with the optimized queries using the same experimental procedures of RQ_2.1. Ultimately, this research question investigates how susceptible to the TR implementation our optimization strategy is. Indeed, if query optimization is possible with version 2.9.4 but not 8.2.0, that indicates some issue of stability, but does not diminish the impact of the results provided by the earlier version. However, if version 8.2.0 provides even

[5] `http://jmetal.sourceforge.net`

greater performance, that suggests that more advanced TR implementations may be able to provide even more optimized queries than more rudimentary approaches. Note that in either case, we use an out of the box implementation with minimal customization rather than a state-of-the-art approach to show that optimization reduces the need for complex TR implementations.

3.3.2 Provenance of Terms in Near-Optimal Queries (RQ_4, RQ_5, and RQ_6)

While RQ_1, RQ_2, and RQ_3 show that TR-approaches to bug localization have merit and can achieve high performance given an appropriate selection of query terms, they do not provide much insight into how the GA is able to optimize a given query based on a bug report. Although some trends may be present in the data generated to answer those questions, in the end we need more than an anecdotal understanding of such trends to comprehend what makes a near-optimal query better than the original query. Therefore, for the remaining three research questions, we perform a statistical analysis of the provenance of unique terms that remain in a query through optimization. To that end, we seek to categorize near-optimal search terms in three ways based on original queries from the Q_{nH} set. First, we categorize terms based on where they appear in the bug report: the title, description, or both. Second, we categorize terms based on if the words appear in the files to be fixed even after localization hints have been removed. Third, we categorize query terms based on if the words appear in components of the bug report that represent specific pieces of technical information that have been found helpful for TR queries in previous research (Chaparro et al., 2019): expected behavior (EB), observed behavior (OB), and steps to reproduce (S2R). Based on previous research that suggests effective queries retrieve relevant results in the top ten positions of the result list (Zhou et al., 2012), we focus the analysis on those queries in Q_{nH} that begin with an effectiveness > 10 and result in a near-optimal query with effectiveness ≤ 10. That is, for this analysis we are interested in the provenance of terms in near-optimal queries that began with poor performance, but have high performance after optimization. Finally, note that we base this analysis on queries optimized using effectiveness score.

Specifically, to answer RQ_4, we take each bug report and build three mutually exclusive term sets: title-only, description-only, and both. That is, a term that is in both the title and description belongs to the "both" set and not the other two, while a term that appears in the "title-only" set, or "description-only" set appears only in the title, or only in the description, respectively. Using these term sets, we then calculate the proportion of terms in the initial query from each set and compare that to the proportion of terms in the optimized query from each set. We then aggregate the results of all trials of all bugs for each dataset to arrive at some intuition about how the proportion of query terms in each term set changes through the process of optimization.

To answer RQ_5, we take the vocabulary of unique terms in the golden set of each bug report and compute the proportion of terms in the initial query that belong to that vocabulary and compare it to the proportion of those terms in

the optimized query. This provides insight into how much the direct overlap of terms with relevant documents impacts query optimization, but also illustrates the amount of noise left in the query even after optimization. Developing heuristics for identifying which words are harmful to query performance could also lead to a better understanding of how to optimize queries without access to the ground truth.

To answer RQ_6, we perform a similar process as for RQ_4. First, we applied DEMIBUD-R (Chaparro et al., 2017b) to automatically identify the bug reports for which EB, OB, or S2R information is present. Afterwards, for all the bugs reported by DEMIBUD-R as containing EB, OB, or S2R information, we manually classified the terms into all combinations of term sets based on EB, OB, and S2R. We then compute the proportion of terms from each set in the initial and optimized queries and finally compare them to see how the process of optimization affects those proportions. More than just a field-level analysis of bug report terms, which has been previously exploited by more advanced TR-approaches such as Okapi BM-25 (Saha et al., 2013), we provide a functional analysis of different categorizations of technical information present in bug reports outside of explicit localization hints. This highlights the categories of information contributing the most to a near-optimal query.

Note that for these final three research questions, each of the sets that terms can be categorized into are non-mutually exclusive. That is, a term can appear in one or all of the categories at once. Further, in the context of these research questions, our statistical analysis includes a series of single-tailed t-tests to show the statistical significance between the proportion of terms in optimal queries from a set of possible locations (*e.g.,* a bug report's title or description). We also pair this hypothesis testing with power analysis and provide effect sizes for each statistically significant result.

4 Results

4.1 Effectiveness of TR-based Bug Localization

4.1.1 RQ_1: Localization Hints and Queries from Bug Reports

Table 3 shows all evaluation metrics for each project in the dataset when using the complete bug report as query, both including (Q_a) and excluding (Q_{nH}) localization hints. For each metric calculated for each project, a pair of bold values indicates statistically significant impact of the localization hints at 95% confidence with at least a small effect size. Asterisks indicate medium effect sizes. We observed that the impact of localization hints is not statistically significant across all data; however, there are instances where localization hints are seen to substantially boost TR performance. Overall, looking at the last row in Table 3, it is clear that while localization hints definitely help IR-based bug localization, the difference in performance between Q_a and Q_{nH} is not as strong as observed in previous studies (Kochhar et al., 2014; Wang et al., 2015). Indeed, (Wang

Table 3 RQ_1: Comparison of queries Q_a and Q_{nH}. A pair of bold values indicates statistical significance between them at 95% confidence with at least a small effect size and * indicates medium effect sizes.

	Median Eff.		HITS@1		HITS@5		HITS@10		MAP		MRR	
Project	Q_a	Q_{nH}	Q_a	Q_{nH}	Q_a	Q_{nH}	Q_a	Q_{nH}	Q_a	Q_{nH}	Q_a	Q_{nH}
AspectJ	33	36	0.04	0.06	0.17	0.18	0.28	0.26	0.09	0.10	0.12	0.13
Birt	58	56	0.02	0.02	0.08	0.05	0.15	0.14	0.07	0.06	0.10	0.09
BookKeeper	2	3	0.29	0.21	0.79	0.67	0.83	0.75	0.38	0.33	0.51	0.41
Derby	**15**	**26**	0.16	0.12	0.39	0.31	0.43	0.39	0.17	0.16	0.25	0.21
JodaTime	**2***	**13***	0.14	0.14	**0.57**	**0.14**	**0.57**	**0.29**	0.38	0.21	0.38	0.21
Lucene	**2.5***	**11***	0.31	0.20	**0.69**	**0.33**	0.78	0.47	0.38	0.25	**0.47**	**0.28**
Mahout	**5***	**25***	**0.36**	**0.12**	**0.64**	**0.32**	**0.80**	**0.40**	**0.46***	**0.18***	**0.46***	**0.19***
OpenJpa	**5.5**	**18.5**	0.19	0.13	**0.50***	**0.25***	0.56	0.31	**0.32**	**0.15**	0.35	0.22
Pig	9	10.5	0.25	0.19	0.44	0.39	0.53	0.50	0.30	0.28	0.35	0.30
Solr	**2**	**7**	**0.33**	**0.18**	0.67	0.42	0.76	0.56	0.42	0.26	0.49	0.30
SWT	**3**	**5**	0.35	0.31	0.61	0.53	0.78	0.69	0.43	0.37	0.48	0.43
Tika	**2***	**16***	**0.43***	**0.10***	0.62	0.29	0.81	0.38	**0.38***	**0.16***	**0.53***	**0.21***
Tomcat	2	12.5	0.40	0.23	0.67	0.37	0.74	0.41	0.55	0.31	0.59	0.34
ZooKeeper	**2**	**5**	**0.41***	**0.17***	0.73	0.52	0.79	0.64	**0.49**	**0.28**	**0.55**	**0.32**
ZXing	2	6	0.36	0.31	0.57	0.46	0.64	0.54	0.45	0.36	0.49	0.40
Total	**9.6**	**16.7**	0.27	0.17	0.54	0.36	0.63	0.45	0.35	0.23	**0.41**	**0.27**

et al., 2015) and (Kochhar et al., 2014) found a statistically significant difference in performance when using queries containing/not containing localization hints, accompanied by a large effect size.

These differences in findings might be due to several reasons. First, it is worth noting that the dataset used in our experiment is different from those used in (Kochhar et al., 2014; Wang et al., 2015). Second, as explained previously, we adopt a different experimental design. More specifically, we use the same set of bug reports to derive Q_a and Q_{nH}: the latter is a "worst case scenario" in which all terms matching code components have been manually redacted from the bug report (*i.e.,* from Q_a). This allows us to review the performance of TR-based localization in the presence of a potentially elevated vocabulary mismatch (since matching terms between the code identifiers and the bug report have been removed from the latter) and directly compare the same bug report with and without localization hints. Alternatively, previous studies classify the set of bug reports on the basis of the degree of bug localization they contain (*e.g.,* bug is fully localized in the report, partially localized, or not localized) and compare the performance of TR-based bug localization on these (*disjoint*) sets of bugs. These methodological differences make a direct comparison of the results achieved in our study versus previous studies difficult. At minimum, these results underscore the fact that the impact of localization hints on performance is inextricably linked to the specific project under study and the set of bug reports considered from that project.

Finally, it is worth commenting on the overall performance achieved by using the whole textual content of the bug report as a query, especially when localization hints are not present (*i.e.,* Q_{nH}, the scenario in which TR-based bug localization is needed the most). The median effectiveness across all projects is 13, meaning that for half of the queries, the first relevant document is retrieved after position 13 in the ranked list. Considering that our study is run at file level (*i.e.,* each document in the ranked list represents a Java file), the effort required to analyze false positives in the ranked list would likely be too high

to be considered acceptable by developers. Also, Q_{nH} was able to retrieve a buggy file in the top five positions (HITS@5) for only 36% of queries and in the top ten positions (HITS@10) for 46% of queries on average. This, of course, translates to poor performance also across the rest of the metrics we consider.

The achieved results support doubts raised in previous work about the actual usefulness of TR-based bug localization when hints are not provided. However, there is an important detail to remember: as done in previous work, we are using the whole textual content of the bug report as a query, which, as previous studies showed (Chaparro and Marcus, 2016), can contain noise that hinders TR performance. In the next research question we investigate what the potential effectiveness of IR-based bug localization is given the ability to formulate a near-optimal query instead of using this default one.

RQ$_1$: The presence of localization hints can significantly boost the results of IR-based bug localization at the project-level, but not necessarily for all bug reports and/or in all projects. We also observed that the performance of IR-based bug localization is actually poor in the scenario it is needed the most, with less than 50% of queries able to retrieve a relevant result in the top 10 positions of the ranked list without the aid of localization hints.

4.1.2 RQ$_{2.1}$: Optimizing Bug Report Queries to Retrieve a Single Relevant Document

Table 4 shows all evaluation metrics for each project in the dataset when using a near-optimal query generated by our GA (using effectiveness as fitness function) based on the bug report vocabulary with and without localization hints. The improvements made by selectively formulating a query rather than using the complete bug report vocabulary are immediately noticeable. Beginning with median effectiveness, we now see that for at least half of the queries a user will arrive at a buggy file after inspecting only the first item in the result set. Note that this holds both when localization hints are present in the bug report (QGA_a) and when they are not (QGA_{nH}). This finding is further supported by the average HITS@1 score over the entire dataset, which shows that overall 73% and 64% of the queries with and without localization hints respectively, are able to return a relevant file in the first position in the search results. This is a more than three-fold improvement over the full text queries, which are generally the ones used in the evaluation of TR techniques. Moreover, the MAP of each dataset also improves dramatically, indicating that not only is one relevant file being pushed to the top of the results, but many of the other relevant files are moving up the list as well.

For this research question we performed the same statistical tests between queries that contain localization hints and those that do not. We found that in the case of queries formulated by the GA, the difference in values other than median effectiveness are statistically significant only for Mahout, but even then only with a negligible effect size. Otherwise, near-optimal queries

manage to mitigate the lack of localization hints every time. In addition, we calculated the same statistical tests to measure differences between the queries composed by using the whole text in the bug report (*i.e.,* those used in RQ_1) and the near-optimal queries formulated by the GA, finding a statistically significant difference with medium or large effect sizes in every case. Summarizing these findings, **the achieved results show that the vocabulary of the bug report is all we need to formulate a good query and make the application of IR-based bug localization successful in most cases**.

It is also worth noting that these results have been achieved by using a simple VSM as the TR engine, configured with default parameters and publicly available in an open source library (*i.e.,* `lucene`). While our goal in this study was to determine the best results we can get by only manipulating the query, given a more robust TR engine capable of better handling issues such as vocabulary mismatch, it is reasonable to hypothesize that even better performance could be obtained. Further, these results show that even though there are numerous studies which have itemized and dissected the limitations of applying TR to source code, specifically for bug localization, the query itself is a tremendously important aspect of optimizing IR, potentially more than internal parameters or even the choice of TR engine.

The results we achieved in RQ_2 are very promising for researchers working on TR-based bug localization and its many derivatives, as well as for practitioners that have implemented or are interested in implementing these systems to improve their development process. However, these findings also come with some practical implications that need to be addressed in the evaluation of future TR approaches. Mainly, evaluating TR techniques using the standard approach of building queries by concatenating the contents of the bug report title and description is an additional factor that could introduce bias in an TR experiment's results. Indeed, our data suggests that the particular query formulated starting from a given bug report is a major factor in the performance of TR approaches. Therefore, great care should be taken when formulating queries from bug reports for evaluation to not only address potential biases such as the presence of localization hints and bloated ground truths, but also biases imposed by the query itself. For example, it has been suggested that queries containing a large volume of text are also likely to contain a lot of noise (Chaparro and Marcus, 2016), which innately lowers performance. This is supported by the results of our study, where the GA queries obtain better results and are also shorter (see Table 5).

Clearly, it is often a non-trivial task to read a bug report and convert its text into a meaningful query that TR approaches can use to locate buggy files. In fact, previous work on iterative query refinement through user feedback (Gay et al., 2009) found that developers were often unable to reach a good query given a sufficiently bad starting point. Because the quality of a query is not always readily apparent, one can imagine situations in which studies have an imbalance of queries with various degrees of quality. It is then entirely possible that this balance, whether in favor of poor or high quality queries, injects bias into evaluation results.

Table 4 $RQ_{2.1}$: Performance of *near-optimal queries* QGA_a and QGA_{nH} (using Lucene 2.9.4)

	Median Eff.		HITS@1		HITS@5		HITS@10		MAP		MRR	
Project	QGA_a	QGA_{nH}	QGA_a	QGA_{nH}	QGA_a	QGA_{nH}	QGA_a	QGA_{nH}	QGA_a	QGA_{nH}	QGA_a	QGA_{nH}
AspectJ	1	1	0.46	0.60	0.72	0.79	0.78	0.86	0.41	0.49	0.58	0.68
BookKeeper	1	1	0.88	0.75	1.00	0.96	1.00	0.96	0.56	0.48	0.92	0.85
Birt	1	1	0.34	0.35	0.50	0.51	0.55	0.57	0.43	0.43	0.61	0.64
Derby	1	1	0.55	0.53	0.67	0.65	0.67	0.73	0.40	0.40	0.61	0.60
JodaTime	1	1	0.57	0.57	1.00	0.86	1.00	0.86	0.79	0.73	0.79	0.73
Lucene	1	1	0.80	0.63	0.93	0.80	0.97	0.80	0.63	0.49	0.85	0.72
Mahout	1	1	0.88	0.56	0.96	0.72	1.00	0.76	0.84	0.52	0.91	0.63
OpenJpa	1	1	0.75	0.75	0.88	0.94	0.94	1.00	0.63	0.61	0.82	0.83
Pig	1	1	0.69	0.64	0.86	0.81	0.89	0.86	0.65	0.61	0.77	0.71
Solr	1	1	0.84	0.71	0.91	0.84	0.93	0.89	0.68	0.57	0.88	0.77
SWT	1	1	0.79	0.76	0.89	0.91	0.94	0.93	0.69	0.67	0.85	0.83
Tika	1	1	0.90	0.71	0.90	0.86	0.90	0.86	0.59	0.42	0.91	0.78
Tomcat	1	1	0.77	0.49	0.89	0.68	0.91	0.75	0.85	0.59	0.95	0.70
ZooKeeper	1	1	0.81	0.74	0.95	0.88	0.96	0.94	0.74	0.70	0.86	0.80
ZXing	1	1	0.93	0.85	1.00	0.92	1.00	1.00	0.91	0.86	0.95	0.90
Total	1	1	0.73	0.64	0.81	0.81	0.9	0.85	0.57	0.57	0.82	0.74

Given that initial query formulation is so important to the outcome of an TR approach, it is natural to ask questions about how the research community can best support query formulation given a bug report vocabulary without *a priori* knowledge of the ground truth. As mentioned in Section 2.2, previous work has focused on how to reformulate a query from an initial query. We see the set of near-optimal queries extracted to answer RQ_2 as a precious source that can represent a starting point for further research on this topic, such as studying what the characteristics of high-performing queries are. As a hint on how much different the near-optimal queries formulated by our GA are with respect to the queries including the whole textual content of the bug report, Table 5 shows the average size of Q_a and Q_{nH} before and after the application of the GA. We report the size in terms of unique and non-unique terms. In a vast majority of cases, the GA reduces the terms used in the original query by more than 50%, and in each case it results in a higher ratio of unique to non-unique terms. This further supports the idea that there are words in the original queries in both Q_a and Q_{nH} whose relevance is diminished by other noisy terms in these queries, such that the TR approach is not able to appropriately leverage the information represented by meaningful terms.

Table 6 shows some examples from our dataset of queries before and after the GA application. In the following paragraphs, we discuss these queries to exemplify some interesting properties. From this data, we can see extreme cases in which the GA significantly reduces the number of terms in the query, by one order of magnitude. Finding the right query in these cases could prove challenging for a human, given so many term combinations, therefore stressing the need for automatic techniques to help with this task.

Specifically, for bug 40257 in AspectJ, the GA is able to derive a two-word query from an original eleven-word query which boosts the effectiveness from 124 to 9. Similarly, the large 44-term query for bug 54178 in Tomcat is reduced to a smaller 18-term query, boosting the effectiveness from 44 to 1.

For the bug Lucene-4469, the algorithm is able to derive a query with maximum effectiveness from the original query with an effectiveness of 249 by reducing the 31-word query to a seven-word one. Each of these cases illustrate the amount of information buried in bug reports that could potentially be

Table 5 Average query sizes in number of words before and after applying the GA

	Q_a				Q_{nH}			
	Before GA		After GA		Before GA		After GA	
project	Non-Unique	Unique	Non-Unique	Unique	Non-Unique	Unique	Non-Unique	Unique
AspectJ	224.35	61.10	104.63	39.34	49.10	29.89	20.83	14.95
BookKeeper	83.63	35.83	40.25	22.75	24.83	17.92	11.13	9.29
Birt	119.16	50.39	53.20	28.33	58.82	35.58	24.16	17.77
Derby	190.41	53.29	89.94	35.33	44.14	26.65	19.43	14.43
JodaTime	126.57	44.00	59.86	27.00	46.14	29.43	19.43	15.43
Lucene	232.90	98.00	114.27	57.53	27.43	21.63	11.53	10.17
Mahout	97.72	45.72	47.28	28.36	34.24	24.96	13.96	11.60
OpenJpa	166.06	61.00	77.69	39.63	41.06	29.94	17.50	15.19
Pig	117.44	46.25	53.28	28.61	25.00	19.11	10.50	8.72
Solr	87.80	41.09	41.51	25.00	42.93	29.13	19.16	15.22
SWT	100.56	41.01	47.80	25.13	42.80	27.84	19.31	14.91
Tika	74.33	34.76	35.05	20.57	26.00	19.48	10.90	9.57
Tomcat	130.87	54.98	63.78	34.54	44.91	28.70	19.64	14.95
ZooKeeper	106.09	43.36	52.18	27.65	38.14	25.58	17.92	14.35
ZXing	123.31	72.92	58.62	40.54	73.23	54.92	33.23	27.92
Total (Avg.)	132.08	52.25	62.62	32.02	41.25	28.05	17.91	14.30

uncovered by automated formulation techniques, thus dramatically increasing the performance of TR-based bug localization techniques.

It is also interesting to note that given a query that already retrieves a relevant document in position one, the GA is still able to reduce the number of words in the query while maintaining the same, perfect effectiveness. For example, bug pig-3327 results in an original query of 21 words having perfect effectiveness. The GA is able to derive from it a query still exhibiting perfect effectiveness but using as few as seven words. This indicates that not all noisy words in a query are equivalent. There are some sources of benign noise that do not affect the overall effectiveness of the query and are just ancillary words not really required to represent the information sought in the document corpus.

Finally, we note that there is often not a single and unique *near-optimal query* derived by the GA. In fact, for bug 37576 in AspectJ (see Table 6), the algorithm is able to derive ten distinct queries (*i.e.,* one for each iteration), with lengths ranging from three to ten words, all having perfect effectiveness. This suggests that automatic formulation techniques should not always look for one single, ideal query. Rather, many alternative queries built from the text in the bug report could all lead to exceptional retrieval results.

RQ$_{2.1}$: We find strong evidence that a near-optimal query extracted from the bug report vocabulary significantly improves the performance of TR-based bug localization compared to the default query. This indicates the potential usefulness of TR bug localization, even in cases where the bug report does not contain any localization hint, which has been shown to be an especially problematic situation for developers (Bettenburg et al., 2008). We strongly believe that the design and construction of techniques supporting the formulation of an *optimal query* should be the main research direction to investigate in IR-based bug localization.

Table 6 Selected examples of queries and their effectiveness before and after GA reformulation

BugId	Initial Eff	Initial Query	GA Eff	GA Query
AspectJ-40257	124	pars path lst file broken rel path longer parser properli ajd	9	lst ajd
Tika-1083	208	add link uti valu tika metadata xml tika 1012 ad tika link tika uti patch fill valu	24	add xml
Lucene-4469	249	test appear useless reason guess debug explicitli disabl mdw call dont check valu test current fail wouldnt rememb right catch throwabl record insid statu test chang let mdw run search test	1	appear disabl check valu test wouldnt rememb
Pig-3327	1	pig hit oom fetch task report gc overhead limit exceed hit 23 script launch job ha 80k map arrai caus oom	1	oom task report map arrai caus oom
AspectJ-37576	4	ant task switch boot classpath entiti rc2 ajc task iajc task take nest bootclasspath entiti classpath vice versa	1	ant rc2 ajc iajc take entiti versa
AspectJ-37576	4	ant task switch boot classpath entiti rc2 ajc task iajc task take nest bootclasspath entiti classpath vice versa	1	ant switch rc2 iajc bootclasspath entiti classpath
AspectJ-37576	4	ant task switch boot classpath entiti rc2 ajc task iajc task take nest bootclasspath entiti classpath vice versa	1	ant switch iajc nest entiti classpath versa
AspectJ-37576	4	ant task switch boot classpath entiti rc2 ajc task iajc task take nest bootclasspath entiti classpath vice versa	1	ant classpath entiti rc2 iajc take bootclasspath entiti vice versa
Tomcat-54178	44	bug 54178 cve 2013 2071 make recycl orgin report messag post tomcat call one http request use post method tomcat call strage test 7 0 23 7 0 32 7 0 32 7 0 32 more reproduc 7 0 23 strang issu tomcat ha releas t didn t eye attach imag	1	cve 2013 recycl orgin call one http request method 7 7 0 reproduc 7 strang ha releas t didn t eye
AspectJ-37576	4	ant task switch boot classpath entiti rc2 ajc task iajc task take nest bootclasspath entiti classpath vice versa	1	ant switch rc2 iajc take nest bootclasspath versa
AspectJ-37576	4	ant task switch boot classpath entiti rc2 ajc task iajc task take nest bootclasspath entiti classpath vice versa	1	ant switch rc2 iajc take nest bootclasspath versa
AspectJ-37576	4	ant task switch boot classpath entiti rc2 ajc task iajc task take nest bootclasspath entiti classpath vice versa	1	classpath iajc take
AspectJ-37576	4	ant task switch boot classpath entiti rc2 ajc task iajc task take nest bootclasspath entiti classpath vice versa	1	ant switch boot classpath ajc iajc nest entiti versa
AspectJ-37576	4	ant task switch boot classpath entiti rc2 ajc task iajc task take nest bootclasspath entiti classpath vice versa	1	ant switch classpath iajc
AspectJ-37576	4	ant task switch boot classpath entiti rc2 ajc task iajc task take nest bootclasspath entiti classpath vice versa	1	rc2 ajc iajc take classpath vice versa
AspectJ-37576	4	ant task switch boot classpath entiti rc2 ajc task iajc task take nest bootclasspath entiti classpath vice versa	1	ant switch boot classpath ajc iajc take nest entiti

4.1.3 $RQ_{2.2}$: Optimizing Bug Report Queries to Retrieve Multiple Relevant Documents

To answer $RQ_{2.1}$ we employed a GA using effectiveness as a fitness function. At each iteration, the GA minimized the rank of the first relevant document in the result list (*i.e.,* pushing it upward toward the first position in the list). However, this is only one query metric that could be used as a fitness function. An alternative goal of optimization could be to retrieve all (or at least the largest possible subset) of relevant classes for each bug. This is particularly useful for more general software engineering tasks. As an example, when establishing links between high- and low-level system requirements, there are no explicit dependencies that can be navigated like those available between relevant pieces

Table 7 Performance of effectiveness-based and AP-based *near-optimal queries* with no localization hints. A pair of bold values indicates statistical significance between them at 95% confidence with at least a small effect size and * indicates medium effect sizes.

	Median Eff.		HITS@1		HITS@5		HITS@10		MAP		MRR	
Project	QGA_{Eff}	QGA_{AP}	QGA_{Eff}	QGA_{AP}	QGA_{Eff}	QGA_{AP}	QGA_{Eff}	QGA_{AP}	QGA_{Eff}	QGA_{AP}	QGA_{Eff}	QGA_{AP}
AspectJ	1	1	0.60	0.57	0.79	0.76	0.86	0.84	**0.49***	**0.57***	0.68	0.66
BookKeeper	1	1	0.75	0.78	0.96	0.96	0.96	0.96	**0.48***	**0.64***	0.85	0.85
Birt	1	1	0.35	0.38	0.51	0.52	0.57	0.57	0.43	0.49	0.64	0.61
Derby	1	1	0.53	0.56	0.65	0.66	0.73	0.75	**0.40**	**0.48**	0.60	0.61
JodaTime	1	1	0.57	0.64	0.86	0.84	0.86	0.86	**0.73**	**0.74**	0.73	0.74
Lucene	1	1	0.63	0.58	0.80	0.80	0.80	0.80	**0.49***	**0.55***	0.72	0.68
Mahout	1	1	0.56	0.52	0.72	0.69	0.76	0.73	**0.52**	**0.54**	0.63	0.58
OpenJpa	1	1	**0.75***	**0.69***	**0.94***	**0.84***	**1.00**	**0.92**	0.61	0.59	0.83	0.77
Pig	1	1	0.64	0.65	0.81	0.81	0.86	0.86	**0.61**	**0.65**	0.71	0.72
Solr	1	1	0.71	0.68	0.84	0.83	0.89	0.88	**0.57***	**0.63***	0.77	0.75
SWT	1	1	0.76	0.79	0.91	0.91	0.93	0.93	**0.67***	**0.78***	0.83	0.85
Tika	1	1	0.71	0.71	0.86	0.82	0.86	0.82	**0.42***	**0.65***	0.78	0.77
Tomcat	1	1	0.49	0.40	0.68	0.69	0.75	0.75	0.59	0.61	0.70	0.69
ZooKeeper	1	1	0.74	0.68	0.88	0.84	0.94	0.90	**0.70**	**0.71**	0.80	0.76
ZXing	1	1	**0.85***	**0.91***	0.92	0.94	1.00	1.00	**0.86**	**0.91**	0.90	0.94
Total	1	1	0.62	0.61	0.80	0.79	0.85	0.84	**0.61**	**0.64**	0.74	0.73

Table 8 Performance of effectiveness-based and recall-based *near-optimal queries* with no localization hints. There was no statistical significance between the results for the two sets of queries.

	Median Eff.		HITS@1		HITS@5		HITS@10		MAP		MRR	
Project	QGA_{Eff}	QGA_{Rec}	QGA_{Eff}	QGA_{Rec}	QGA_{Eff}	QGA_{Rec}	QGA_{Eff}	QGA_{Rec}	QGA_{Eff}	QGA_{Rec}	QGA_{Eff}	QGA_{Rec}
AspectJ	1	6	0.60	0.38	0.79	0.49	0.86	0.52	0.49	0.38	0.68	0.44
BookKeeper	1	2	0.75	0.46	0.96	0.83	0.96	0.83	0.48	0.51	0.85	0.60
Birt	1	9	0.35	0.22	0.51	0.31	0.57	0.36	0.43	0.39	0.64	0.52
Derby	1	4	0.53	0.36	0.65	0.55	0.73	0.57	0.40	0.39	0.60	0.43
JodaTime	1	1.4	0.57	0.65	0.86	0.68	0.86	0.73	0.73	0.67	0.73	0.67
Lucene	1	1.67	0.63	0.49	0.80	0.74	0.80	0.76	0.49	0.53	0.72	0.60
Mahout	1	1.33	0.56	0.52	0.72	0.60	0.76	0.61	0.52	0.54	0.63	0.56
OpenJpa	1	2.33	0.75	0.35	0.94	0.63	1.00	0.65	0.61	0.40	0.83	0.47
Pig	1	1	0.64	0.56	0.81	0.66	0.86	0.67	0.61	0.58	0.71	0.61
Solr	1	1.67	0.71	0.49	0.84	0.74	0.89	0.78	0.57	0.54	0.77	0.61
SWT	1	1	0.76	0.65	0.91	0.78	0.93	0.80	0.67	0.67	0.83	0.72
Tika	1	1.33	0.71	0.54	0.86	0.81	0.86	0.81	0.42	0.57	0.78	0.65
Tomcat	1	3	0.49	0.28	0.68	0.34	0.75	0.41	0.59	0.55	0.70	0.66
ZooKeeper	1	1	0.74	0.72	0.88	0.83	0.94	0.86	0.70	0.73	0.80	0.77
ZXing	1	1	0.85	0.75	0.92	0.76	1.00	0.78	0.86	0.73	0.90	0.76
Total	1	2.52	0.64	0.49	0.81	0.65	0.85	0.68	0.57	0.55	0.74	0.60

of source code that are likely to share an edge in a call graph. Therefore, rather than finding a single relevant requirement near the top of the results, a truly *near-optimal* query for this purpose would find *all* of the relevant requirements there. Tables 7 and 8 show the results of generating near-optimal queries using average precision (QGA_{AP}) and recall (QGA_{Rec}) as the fitness function, respectively, compared to the query obtained by the GA when optimizing effectiveness (QGA_{Eff}). Note that average precision and recall were explicitly chosen for evaluation as they both measure a query's ability to cluster all of the relevant documents near the top of the ranked results list. Note also that when implementing the fitness functions, the GA is set to minimize the reciprocal of either average precision or recall to be consistent with the previous experiment which minimized effectiveness. This is because a *near-optimal* query should have a low effectiveness score, but high average precision and recall scores.

When using AP as the fitness function, the performance for finding a single relevant document near the top of the result list is largely unchanged from using the effectiveness. However, the largest difference in average performance across all systems is in MAP, which is the average of all AP scores within a dataset. While the queries are not able to achieve perfect MAP (*i.e.,* 1.00 for grouping all k relevant documents as the top k results), we do see a statistically significant increase in MAP compared to using effectiveness in almost all cases. Therefore, these findings suggest that it is possible to generate queries that are optimized

for retrieving multiple relevant documents. Interestingly, while QGA_{AP} queries outperform those in QGA_{Eff} in terms of MAP substantially, they perform similarly for most of the other metrics, and in the case of some metrics for OpenJpa and ZXing are statistically significantly worse. This is consistent with previous findings that indicate many aspects of software engineering are highly dependent on the project under study. Therefore, while it appears queries can be optimized to increase MAP for some systems, this is not true for all of them.

Also interestingly, using recall as a fitness function produces queries that are statistically significantly worse than either effectiveness or AP. This is most apparent in the QGA_{Rec} results for median effectiveness, which in the case of AspectJ are six times worse than the other two fitness functions. Indeed the QGA_{Rec} queries barely outperform QGA_{Eff} queries even measured by MAP, which is closely conceptually related to recall. However, there is no statistical significance between the MAP results for those sets of queries. Further, in cases where QGA_{Eff} queries outperform QGA_{Rec} queries, they do so substantially and statistically significantly.

Finally, note that for bugs which only have a single class in the golden set, the process of optimizing for AP or recall is irrelevant, as in this case finding the first relevant document is equivalent to finding all relevant documents. Table 9 shows the improvement in MAP achieved by either AP or recall for bugs that required changes in multiple classes to resolve. In these cases, MAP for the original queries constructed from the full bug report minus localization hints are very low with an average of only 0.20. Both the GA with AP and recall as a fitness function more than double the MAP to 0.57 and 0.47, respectively. Therefore, we see that not only is it possible to improve the MAP for a mixed set of queries containing documents with only a single relevant document in the search space, but improvements in MAP for the case that matters most (*i.e.,* where there are multiple relevant documents in the search space) are substantial. These findings paired with the results of the AP fitness function experiment indicate that while it is possible to optimize queries to retrieve multiple relevant files, this is much harder than optimizing a query to find a single relevant document.

RQ$_{2.2}$: While it is possible to optimize a query to find all relevant documents near the top of the result list for some systems, our data suggests that it is much more difficult to do and might not be possible for all systems. However, for bugs with multiple classes in their golden set, MAP improvement by using AP as a fitness function is substantial.

4.1.4 RQ$_3$: Optimizing Bug Report Queries with Lucene 8.2.0

As discussed in Section 3.3.1, we are also interested in the impact that changing the implementation of Lucene has on the GA's ability to optimize queries comprised of a bug report title and description. Table 10 shows the baseline performance of such queries with and without hints prior to applying GA optimization based on effectiveness. This table should be compared with Table 3,

Table 9 MAP comparison of Q_{BR}, Q_{AP}, and Q_{Recall} for bugs that required changes to multiple classes

Project	Q_{BR}	Q_{AP}	Q_{Recall}
AspectJ	0.10	0.46	0.32
BookKeeper	0.19	0.50	0.33
Birt	0.07	0.28	0.24
Derby	0.16	0.45	0.30
Lucene	0.12	0.48	0.34
Mahout	0.29	0.78	0.70
OpenJpa	0.15	0.48	0.33
Pig	0.13	0.40	0.34
Solr	0.14	0.46	0.33
SWT	0.27	0.70	0.53
Tika	0.20	0.65	0.58
Tomcat	0.31	0.54	0.40
ZooKeeper	0.19	0.53	0.52
ZXing	0.51	1.00	1.00
Average	0.20	0.57	0.47

in which the baseline performance using Lucene 2.9.4 is reported. Through that comparison, we see the same average trends across the entire dataset. However, for several specific systems the performance of queries without hints is significantly improved when using the newer version of Lucene. Namely, for both Derby and JodaTime, the HITS@10 ratios are identical with and without hints, whereas there was substantial performance degradation when hints were removed using Lucene 2.9.4. Further, there is a noticeable degradation in effectiveness both with and without hints when using version 8.2.0 compared to 2.9.4, but this is more pronounced for some systems than others. Likewise, there is a marginal increase in the HITS@K ratios with 8.2.0, the magnitude of which varies across systems.

Table 11 and Table 4 show comparative performance data with and without hints for near-optimal queries derived by Lucene 8.2.0 and 2.9.4 respectively. There are two primary takeaways from this data. First, Lucene 8.2.0 has reduced performance as measured by our chosen metrics when compared to Lucene 2.9.4. In many cases, the degradation is ∼10%, indicating that for bug localization, at least for these systems, the tf-idf similarity is superior to the BM25 similarity. Second, we still observe that the delta between optimized queries with and without hints is relatively small. Therefore, even though there appears to be an aggregate reduction in overall performance related to the change in TR implementation, optimized queries are still able to achieve high performance even in the absence of localization hints. Ultimately, this data suggests that optimization is not bound to a single TR-engine implementation or similarity metric. For any two implementations, there will likely be differences in performance, as well as in the amount of optimization that takes place. However, for the two implementations considered here, the implementation with the lower performance still provides drastic improvement when comparing the original and optimized queries.

Table 10 RQ_3: Comparison of queries Q_a and Q_{nH} using Lucene 8.2.0. A pair of bold values indicates statistical significance between them at 95% confidence with at least a small effect size and * indicates medium effect sizes.

	Median Eff.		HITS@1		HITS@5		HITS@10		MAP		MRR	
Project	Q_a	Q_{nH}	Q_a	Q_{nH}	Q_a	Q_{nH}	Q_a	Q_{nH}	Q_a	Q_{nH}	Q_a	Q_{nH}
AspectJ	40	55.5	0.03	0.02	0.22	0.10	0.22	0.17	0.10	0.08	0.12	0.09
Birt	67.5	92	0.02	0.01	0.10	0.05	0.14	0.11	0.07	0.05	0.09	0.08
BookKeeper	3	2.5	0.22	0.21	0.46	0.41	0.59	0.49	0.39	0.35	0.53	0.46
Derby	14	15	0.16	0.08	0.23	0.19	0.29	0.29	0.23	0.16	0.32	0.23
JodaTime	**7**	**10**	0.29	0.29	**0.43**	**0.29**	0.57	0.57	0.40	0.34	0.40	0.34
Lucene	4	17	0.29	0.19	0.54	0.25	0.67	0.31	0.42	0.24	0.46	0.26
Mahout	5	43	0.30	0.02	0.40	0.14	0.65	0.27	0.39	0.09	0.45	0.15
OpenJpa	18	20	**0.18***	**0.00***	**0.36***	**0.11***	0.42	0.23	0.28	0.08	0.33	0.12
Pig	5.5	19.5	0.24	0.16	0.41	0.35	0.53	0.41	0.35	0.26	0.40	0.28
Solr	3	9.5	0.29	0.14	0.53	0.28	0.63	0.44	0.43	0.24	0.48	0.27
SWT	2	5	0.30	0.20	0.59	0.40	0.68	0.57	0.45	0.33	0.50	0.38
Tika	2	13	0.28	0.08	0.59	0.19	**0.64***	**0.28***	**0.49**	**0.18**	0.58	0.27
Tomcat	3	28	0.35	0.19	0.56	0.30	0.65	0.34	0.47	0.26	0.51	0.28
ZooKeeper	2	6.5	0.35	0.16	0.59	0.41	0.65	0.52	0.48	0.29	0.40	0.32
ZXing	1	6	0.50	0.32	0.68	0.44	0.68	0.56	0.59	0.37	0.62	0.40
Total	16	24	0.25	0.14	0.45	0.27	0.53	0.37	0.37	0.22	0.41	0.26

We also analyzed whether our main finding (i.e., the improvement in effectiveness achieved when using the GA queries) was influenced by the choice of the IR engine (Lucene 8.2.0, based on BM25 and Lucene 2.9.4, based on tf-idf). In particular, we applied the same statistical procedure previously explained for RQ_1 (i.e., coin test and Cliff's delta) to study the differences between:

- The effectiveness of the original queries (both when using all terms, as well as when removing localization hints) when run using the two IR engines;
- The effectiveness of the GA queries (with/without localization hints) when run using the two IR engines;
- The gain in effectiveness between the GA queries and the original queries obtained using the two IR engines (with/without localization hints).

We found a statistically significant difference in all comparisons (all p-values < 0.01), indicating that the compared distributions are independent. However, the effect size in all comparisons is negligible (all $d < 0.07$), showing that the magnitude of the differences between the distributions is of no practical relevance. In other words, the IR-engine choice has no noticeable influence on our main finding (e.g., the gain in effectiveness obtained by the GA queries).

RQ_3: While we see some aggregate performance reduction with 8.2.0 in the context of bug localization for the specific systems under study, we also observe that query optimization provides queries that negate the need for localization hints to retrieve relevant documents for either version. For any two implementations we expect there will be differences in both raw performance and optimization potential. However, the data suggests that query optimization has the potential to significantly improve results irrespective of the TR implementation.

Table 11 RQ_3: Performance by system of *near-optimal queries* QGA_a and QGA_{nH} with Lucene 8.2.0

	Median Eff.		HITS@1		HITS@5		HITS@10		MAP		MRR	
Project	Q_a	Q_{nH}	Q_a	Q_{nH}	Q_a	Q_{nH}	Q_a	Q_{nH}	Q_a	Q_{nH}	Q_a	Q_{nH}
AspectJ	1	1	0.37	0.37	0.54	0.53	0.60	0.57	0.47	0.45	0.61	0.58
Birt	1	2	0.35	0.30	0.50	0.45	0.54	0.51	0.43	0.38	0.60	0.56
BookKeeper	1	1	0.47	0.43	0.65	0.58	0.74	0.65	0.63	0.55	0.95	0.88
Derby	1	1	0.37	0.37	0.44	0.53	0.48	0.58	0.44	0.47	0.64	0.66
JodaTime	1	1	0.80	0.71	0.86	0.71	0.87	0.93	0.84	0.75	0.84	0.75
Lucene	1	2	0.77	0.35	0.69	0.55	0.80	0.57	0.76	0.47	0.91	0.61
Mahout	1	1	0.66	0.36	0.79	0.53	0.81	0.60	0.73	0.45	0.93	0.60
OpenJpa	1	1	0.52	0.46	0.68	0.63	0.74	0.65	0.61	0.55	0.78	0.78
Pig	1	1	0.45	0.51	0.69	0.62	0.76	0.64	0.60	0.57	0.71	0.67
Solr	1	1	0.60	0.50	0.65	0.59	0.68	0.64	0.67	0.57	0.84	0.73
SWT	1	1	0.68	0.62	0.76	0.72	0.81	0.77	0.76	0.69	0.88	0.83
Tika	1	1	0.55	0.40	0.68	0.46	0.74	0.59	0.68	0.49	0.91	0.75
Tomcat	1	1	0.69	0.40	0.76	0.55	0.78	0.61	0.74	0.48	0.83	0.57
ZooKeeper	1	1	0.63	0.56	0.78	0.74	0.84	0.78	0.73	0.67	0.85	0.78
ZXing	1	1	0.82	0.62	0.97	0.68	0.97	0.74	0.89	0.66	0.92	0.68
Total	1	1	0.58	0.46	0.70	0.59	0.72	0.66	0.67	0.55	0.81	0.70

4.2 Provenance of Terms in Near-Optimal Queries

4.2.1 RQ_4: Effective Near-Optimal Query Terms from a Bug Report

Table 12 shows the average composition of a query in each system based on term membership in either a bug report's title, description, or both for unoptimized (*i.e.,* Q_{nH}) and near-optimal (*i.e.,* QGA_{Eff}) queries along with the average effectiveness score. Interestingly, overall there are no large shifts between the three term sets, with no statistical significant differences between the composition of queries before and after optimization. While there has been some previous work on TR approaches such as Okapi BM-25F (Saha et al., 2013) that are specifically intended to improve performance by varying term weights based on document fields, this data suggests that the benefit of doing so is extremely project-dependent. Moreover, depending on the project, those techniques may not provide sufficient performance improvement to justify the necessary effort required for tuning those weights compared to VSM. Further, while common intuition is that terms specifically from the title would provide the most important information to locate a bug, near-optimal queries for more than half of the datasets under study have the same or smaller proportion of words from the title compared to the associated Q_{nH} query. However, by the same token, terms that appear in both the title and the description do represent a larger proportion of terms present near-optimal queries in the vast majority of the datasets, but that increase is more than five percentage points in only three of the datasets, with no statistical significance. Moreover, despite the specificity of terms in the bug report title or in both the title and description, terms from only the description make up more than two-thirds of near-optimal queries on average for all of the systems. Therefore, while intuition might suggest terms from the title or title and description are the most optimal query terms, and optimization would remove a larger portion of terms that solely appear in the bug description, our data suggests this is not the case.

Table 12 Provenance of terms and effectiveness of queries before and after GA optimization

	Unoptimized				Near-Optimal			
System	Title	Desc	Both	Eff	Title	Desc	Both	Eff
AspectJ	0.09	0.78	0.13	92	0.09	0.74	0.17	12
Birt	0.10	0.72	0.17	599	0.10	0.66	0.24	93
BookKeeper	0.16	0.67	0.17	13	0.16	0.62	0.22	2
Derby	0.10	0.71	0.19	169	0.10	0.67	0.22	31
JodaTime	0.10	0.86	0.04	30	0.11	0.83	0.06	3
Lucene	0.13	0.76	0.11	73	0.13	0.74	0.12	15
Mahout	0.07	0.71	0.20	234	0.06	0.69	0.24	15
OpenJpa	0.06	0.79	0.14	39	0.06	0.76	0.17	2
Pig	0.15	0.68	0.17	110	0.13	0.68	0.18	49
Solr	0.07	0.76	0.16	173	0.04	0.74	0.21	17
SWT	0.12	0.80	0.08	28	0.12	0.77	0.11	5
Tika	0.11	0.73	0.15	47	0.15	0.66	0.19	7
Tomcat	0.17	0.70	0.13	94	0.14	0.67	0.18	18
ZooKeeper	0.09	0.74	0.17	22	0.07	0.71	0.21	4
ZXing	0.02	0.93	0.05	20	0.02	0.90	0.08	2

Table 13 shows the percent change in each set as a query evolves from the original bug report query to a near-optimal query. As expected based on the previous, anecdotal analysis of the near-optimal queries, the average query size for each system decreases by more than half in all but four of the systems. Interestingly, the percent change in title terms is at least 50% in eight of the systems, and the percent change for terms in both the title and description is at least 50% in four of the systems. On average, about half of the terms from the description are removed for all systems. This indicates that while there is substantial noise in bug descriptions used as TR queries, the same is true for titles which are expected to be far more terse and rich in information. Ultimately, this reinforces the fact that simply prioritizing title terms over those in the description is insufficient to identify near-optimal query terms. We have shown that sufficient information exists for a GA to extract a near-optimal query with high performance; however, the terms in that query are not easily categorized into a single field of the original bug report.

RQ$_4$: Although intuition suggests that terms in the title or title and description of a bug report represent those with the most specific information, less than a third of near-optimal queries consist of those terms for the majority of the systems under study. Moreover, during the evolution of a query to its near-optimal formulation, approximately half of the terms in the original query are removed and those terms come from all three sets of terms. Therefore, there is no single component of a bug report that contains particularly optimal query terms in general.

4.2.2 RQ$_5$: Near-Optimal Query Terms from the Golden Set

Although the queries in this part of the study have had all of the localization hints removed, there are still domain-specific terms that appear in the bug report that can also appear in source code identifiers. These terms represent

Table 13 Percentage of change in query size, term provenance, and effectiveness between original queries and optimal queries

System	Size	Title	Desc	Both	Eff
AspectJ	-0.54	-0.43	-0.56	-0.37	-0.83
Birt	-0.53	-0.57	-0.56	-0.33	-0.88
BookKeeper	-0.47	-0.48	-0.53	-0.25	-0.57
Derby	-0.48	-0.33	-0.48	-0.34	-0.71
JodaTime	-0.42	-0.29	-0.43	-0.14	-0.65
Lucene	-0.50	-0.35	-0.51	-0.28	-0.64
Mahout	-0.53	-0.50	-0.54	-0.34	-0.74
OpenJpa	-0.49	-0.41	-0.51	-0.40	-0.75
Pig	-0.54	-0.54	-0.55	-0.44	-0.64
Solr	-0.50	-0.44	-0.49	-0.31	-0.65
SWT	-0.53	-0.34	-0.56	-0.20	-0.68
Tika	-0.50	-0.36	-0.55	-0.33	-0.74
Tomcat	-0.52	-0.59	-0.53	-0.26	-0.60
ZooKeeper	-0.45	-0.37	-0.47	-0.24	-0.63
ZXing	-0.50	-0.44	-0.51	-0.19	-0.59

specific pieces of information that link the natural language representation of bugs to source code implementations, but do not provide the exact location of buggy code. Therefore, intuition would dictate that near optimal queries would maximize the presence of these terms to boost performance in the absence of localization hints. Table 14 shows the effectiveness and overlap of terms in both unoptimized and near optimal queries and classes that were modified to fix the bug.

As with the other results of this study, we see that near optimal queries have vastly improved performance, with all but one of the datasets seeing an average percent change of more than 70% in terms of effectiveness measure. However, while the overlap of terms between the query and files in the golden set grows in most cases and the change in proportion is statistically significant with at least medium effect sizes in each case, the magnitude of the change is not as large as one might expect. Of particular note are OpenJpa and ZooKeeper, both of which actually have a decrease in the number of shared terms when comparing the near-optimal query to the original. It is important to note that throughout this study we use a näive TR approach, VSM with tf-idf term weighting, that uses a term-by-document matrix as an internal representation. These findings are surprising as they suggest that not only are some near-optimal queries formulated by eliminating some terms from the query that have no overlap with relevant documents, but also some terms that appear in both the query and relevant documents. Further, while there is an increase in overlap with relevant documents for most near-optimal queries, this increase is less than ten percentage points in every case and less than five percent for five of the systems. Additionally, for 12 of the 15 datasets, query terms overlapping with those in files from the golden set account for less than half of the near-optimal query terms. At minimum, this provides further support for the existence of "degrees of noise". That is, while some terms are removed from the query in the 50% term reduction shown in RQ_4, other terms that are not in the relevant

Table 14 Average effectiveness and term overlap between golden set files and queries for the original and near-optimal queries per system. A pair of bold values indicates statistical significance between them at 95% confidence with at least a small effect size and * indicates medium effect sizes.

	Unoptimized		Near-Optimal		
System	Eff	Overlap	Eff	Overlap	Eff %Change
AspectJ	71	**25.56***	10	**32.18***	-0.81
Birt	233	23.88	10	32.58	-0.80
BookKeeper	17	**33.40***	2	**39.75***	-0.62
Derby	170	**38.33***	31	**42.60***	-0.71
JodaTime	31	**30.54***	3	**34.15***	-0.65
Lucene	74	**40.99***	15	**45.59***	-0.64
Mahout	93	**22.21***	5	**29.71***	-0.81
OpenJpa	37	**44.74***	2	**42.15***	-0.76
Pig	111	38.01.	49	38.10	-0.64
Solr	173	**40.37***	17	**41.51***	-0.65
SWT	27	**32.77***	5	**41.59***	-0.73
Tika	34	**35.98***	5	**45.73***	-0.73
Tomcat	68	36.69	15	49.07	-0.59
ZooKeeper	23	38.84	4	37.15	-0.63
ZXing	16	**17.17***	1	**22.24***	-0.44

files are left in the query even after optimization is complete. Further, in most cases the optimized query has an effectiveness ≤ 2. Therefore, any further optimization of query terms would not materially impact the performance of TR. As such, techniques for predicting the material impact of each term on a query's performance or developing specialized stopword lists for a system or even individual components of a system could provide substantial performance improvements for TR-approaches to bug localization.

RQ_5: Despite a substantial increase in performance, near-optimal queries have at maximum a ten percentage point increase in terms that overlap with terms found in files in the golden set. Further, there are some cases in which the percent overlap between near-optimal queries and golden set files decreases compared to the original queries, and in both cases a large percentage of non-overlapping terms remain in the optimized query. This indicates that query reduction performed by the GA is able to identify those noisy query terms that are harmful to performance and remove them rather than solely boosting terms found in the golden set. This finding opens the door for future work to automatically rank query terms by their "noisiness."

4.2.3 RQ_6: Near-Optimal Query Terms from EB, OB, and S2R

Table 15 shows the composition of query terms that belong to each of three previously documented types of technical information: observable behavior (OB *i.e.,* the errant behavior that defines the bug), expected behavior (EB *i.e.,* the behavior that should replace the errant behavior to remove the bug), and steps-to-reproduce (S2R *i.e.,* specific actions that once taken result in the observed, errant behavior). First, note that while EB and OB are found in

Table 15 Composition of original and near-optimal queries with respect to OB, EB, and S2R without localization hints by system

	Unoptimized				Near-Optimal			
system	OB	EB	S2R	Eff	OB	EB	S2R	Eff
AspectJ	52.58	12.82	14.11	92	51.53	12.32	14.49	12
Birt	32.51	13.56	45.36	599	37.13	15.31	47.56	93
Bookkeeper	43.56	30.09	0.00	13	43.64	32.85	0.00	2
Derby	41.47	25.24	23.08	169	42.83	29.32	22.42	31
JodaTime	34.92	12.56	8.53	30	35.81	11.74	9.57	3
Lucene	56.54	30.09	2.85	73	58.81	25.51	2.19	15
Mahout	51.92	25.24	0.00	234	58.85	17.61	0.00	15
OpenJpa	65.59	12.56	7.74	39	73.36	15.41	9.23	2
Pig	68.83	13.58	11.73	110	77.77	16.10	9.76	49
Solr	54.04	14.48	9.38	173	57.39	15.95	9.52	17
SWT	59.06	19.32	28.23	28	58.50	22.50	27.45	5
Tika	26.28	58.68	0.00	47	29.73	53.09	0.00	7
Tomcat	50.19	16.21	13.55	94	55.19	16.82	13.47	18
ZooKeeper	54.90	23.30	10.67	22	58.87	24.51	10.89	4
ZXing	16.43	8.07	24.44	20	18.22	9.26	25.53	2

each of the datasets, S2R is only found in some of the datasets, and entirely missing in three of them: BookKeeper, Tika, and Mahout. Second, note that the proportion of terms in OB is much higher than that in EB in all cases except one. Therefore, while many of the bugs in this study have some discussion of the problem, much less discussion of the expectation is available. This is an important distinction, as without expectations the bug might be difficult to resolve satisfactorily even if it can be located. Finally, note that this second trend is mirrored in the composition of optimized queries and that the percent change between original and optimized queries in terms of these behaviors is even smaller than that seen in RQ_3.

It is also interesting to note that even with limited S2R information available, near optimal queries can still be derived. That is, even if the bug report does not direct a developer to the errant behavior in natural language, there is still sufficient textual data to derive a query that allows TR to return a relevant result near the top of the list. Further, we note that while there are some changes in composition of the query based on these three classifications of terms through optimization, these changes are not statistically significant. That is, all modifications in composition could be due to chance. Therefore, there is no clear conclusion that can be drawn between these types of information and their impact on near-optimal queries based on the results of this study. Paired with the previous two research questions (RQ_4 and RQ_5), this finding highlights the GA's unique ability to exploit highly nuanced statistical relationships to derive a high-performance query that may or may not be grounded in human intuition. This is consistent with anecdotal findings in the data that show cases in which nonsensical, two word queries drastically outperform more verbose formulations. Further, the overall findings represented by the results to RQ_4, RQ_5, and RQ_6 mirror findings in previous work (Lawrie and Binkley, 2018) that attempted to use automatic summarization with less

than ideal results. The results of that study indicate that insufficient training data exists to instruct a model on how to derive near-optimal queries without use of the golden set. Going forward, more granular data on the evolution of near-optimal queries should be gathered. Such data can support a large-scale analysis of the step-by-step optimizations that are performed by the GA that might be sufficient to build an automatic model as well as explain the macro-optimizations captured in this study in human understandable terms.

RQ_5: Although EB, OB, and S2R have been shown to improve bug localization based on bug reports in the past, our results suggest that S2R are not required by the GA to optimize a query and neither EB nor OB have a significant percentage increase between the original and optimized queries. Therefore, while this type of information might help human operators in better understanding the nature of the bugs, they do not directly translate to better queries for use in TR-based bug localization.

5 Threats to Validity

Internal validity: Threats to internal validity refer to the extent to which results can support the conclusions in the context of TR-based bug localization. We reduced these threats by considering bug reports used in previous studies (Wang et al., 2015; Chaparro and Marcus, 2016). Also, we performed a manual verification and cleaning process, which resulted in slightly less data for addressing external validity, but provides more confidence in the correctness of our data.

To lessen the likelihood of errors in the manual cleaning process, the two authors in charge of (i) labeling files changed in bug-fixing commits as true or false positive and (ii) removing localization hints from the text of the bug report, followed an agreed upon definition of localization hints and of the distinction between relevant and irrelevant code changes, as described in the study design. Moreover, a third author verified the correctness of the labeling, identifying cases in which additional screening or discussion was required. Another threat to the study is introduced by the randomness of the genetic algorithm. Due to the nature of the genetic algorithm, two independent executions of the genetic algorithm can lead to different results on the same input. To mitigate this threat, we performed ten trials for each application of the genetic algorithm, averaging across the resulting metrics. For 80% of the Q_a queries and 95% of the Q_{nH} queries there was no change in effectiveness between trials. Also, note that the genetic algorithm, being a metaheuristic search method, could converge towards a local optimum rather than to the desired global optimum. However, as shown in a recent study by Lawrie and Binkley (Lawrie and Binkley, 2018), our GA "*does an excellent job of selecting high-scoring queries assuming words must be drawn from the vocabulary found in the bug reports*" (Lawrie and Binkley, 2018). Indeed, the authors used Information Need Analysis (INA) to generate a distribution of scores over a random-query set. Considering the INA

distributions, they found that "*the GA's queries average the 97th percentile with 422 of the 613 appearing in the 100th percentile*" (Lawrie and Binkley, 2018). This supports the ability of our GA in identifying near-optimal queries.

To reduce the likelihood of errors when identifying expected behavior information, we used a two step approach. First, we applied DEMIBUD-R presented by (Chaparro et al., 2017b) to gauge whether EB and S2R information was present in the bug reports in our dataset. Second, we performed a manual annotation process where one of the authors labeled the sentences and the type of information they convey (*i.e.,* observed behavior, expected behavior, steps-to-reproduce) while another author independently verified the correctness of the annotations.

External validity: Threats to the external validity of our study refer to how our results may not be generalizable to all software systems. We mitigated this threat and increase the extent to which the results may be generalizable by including a varied set of software systems from different domains which have been previously used in bug localization research in our study. Moreover, the collected bug reports cover different types of bugs (*e.g.,* crashes, functional).

We use only Java open source projects in our study. Thus, results may not generalize to commercial systems or those written in other languages. Moreover, our results focus on file-level bug localization and may not hold when working at different granularity levels (*i.e.,* method-level).

Construct validity: Threats to construct validity refer to how we measured the performance of the queries. We mitigated these threats by employing four different metrics widely used to measure the performance of TR-based bug localization techniques (Kochhar et al., 2014). Moreover, we used the Asymptotic General Independence test to determine the statistical significance of our results because, when looking at the effectiveness of individual queries within a system, we are comparing non-normal distributions with a high percentage of ties. For continuity and comparability of results, we use the same generalized test for all data analysis, as the statistical assumptions of the Asymptotic General Independence test hold for data with lower instances of ties (Devore and Farnum, 1999). Moreover, we applied the same statistical procedure when determining if the improvement of the effectiveness achieved by the GA queries was influenced by the IR engine used.

6 Conclusion and Future Work

In this paper, we refute the idea that TR-based approaches to bug localization have come into popularity based on improper evaluations and artificially inflated performance. Further, we show that more important than miss-classified bugs, bloated ground truths, and localization hints, evaluations should use a more careful process for formulating queries than merely using the bug report title, description, or a concatenation of the two. We show that TR-based approaches to bug localization exhibit poor performance when using the full text in a bug report as query, particularly in the case when localization hints are not present.

However, a near-optimal query with high performance can still be extracted from the bug report text even after hints have been removed. Additionally, we provide an analysis on the provenance of the terms that make up a near-optimal query and shows that while there are some intuitive ways humans might look for terms to retain to in a near-optimal query, the optimization performed by our GA makes drastically different choices.

Our results on the effect that the presence of localization hints have on TR results are consistent with previous work (Kochhar et al., 2014; Wang et al., 2015). However, more importantly, we show that given only the vocabulary of a bug report, there exists a near-optimal query capable of drastically improved performance compared to a query containing the entire bug vocabulary in a majority of cases. This holds even in the absence of localization hints and when using a rudimentary TR implementation. As a result, we show the potential of TR-based bug localization in the presence of a near-optimal query and the importance of research seeking to formulate a good initial query given only a bug report vocabulary. Further, we show that queries can also be optimized to retrieve multiple relevant documents near the top of the result set; however, MAP is a much better fitness function than Recall to do so.

For provenance, we see that while intuition might suggest terms taken from the title or title and description of the bug report will have higher information density, the majority of near optimal queries come from bug report descriptions. Further, our results on the overlap of terms in near-optimal queries and terms in golden set files show that there is an innocuous type of noise that the GA allows to remain in the query despite those terms inability to contribute to the similarity score between a relevant document and the query. Finally, our results suggest that steps-to-reproduce information is not required in order to obtain an optimized query and neither EB nor OB have a significant percent increase between the original and optimized queries. Therefore, while this type of information might inform human operators on how to formulate more informative bug reports, they do not directly translate to better queries for use in TR-based bug localization.

There are two primary future directions for this research. The first is to extend the analysis to a larger set of bugs, particularly from more modern systems. Additionally, we should consider method-level golden sets. Indeed, other research (Wang et al., 2015) suggests that a significant amount of effort in bug fixing is expended after the buggy file is located, as the user must still locate the bug in the file that must be changed. By working at method level, the amount of code that a developer must inspect is reduced, which in turn results in diminished effort required to find the precise bug location. The second research direction is to find a way of generating sufficient training data to attempt the construction of automatic models for formulating near-optimal queries. The most intuitive way to do so is to log query metrics for each step in the evolution of a query from an initial vocabulary to a near-optimal query. Doing so allows for the use of machine learning techniques such as anomaly detection to exploit the severe imbalance between poor- and high- quality queries in general. Further, by expanding our data collection to a larger number

of systems and bugs, more pairs of poor- and high- quality queries will be produced, which could be used with techniques such as machine translation to convert a poor-quality query into a high-quality one.

Acknowledgements Sonia Haiduc and Esteban Parra were supported in part by the National Science Foundation grants CCF-1846142 and CCF-1644285. Gabriele Bavota and Jevgenija Pantiuchina acknowledge the support by the Swiss National Science Foundation through the JITRA project, No. 172479.

References

Bettenburg N, Just S, Schröter A, Weiss C, Premraj R, Zimmermann T (2008) What makes a good bug report? In: Proceedings of the 16th ACM SIGSOFT International Symposium on Foundations of Software Engineering (FSE'08), ACM, Atlanta, GA, USA, pp 308–318

Carpineto C, Romano G (2012) A survey of automatic query expansion in information retrieval. ACM Computing Surveys (CSUR) 44(1):1

Chaparro O, Marcus A (2016) On the reduction of verbose queries in text retrieval based software maintenance. In: Proc. of the 38th ACM/IEEE International Conf. on Software Engineering, Austin, TX, pp 716–718

Chaparro O, Florez JM, Marcus A (2017a) Using observed behavior to reformulate queries during text retrieval-based bug localization. In: Proceedings of the 33rd IEEE International Conference on Software Maintenance and Evolution (ICSME'17), IEEE, Shangai, China, pp 376–387

Chaparro O, Lu J, Zampetti F, Moreno L, Di Penta M, Marcus A, Bavota G, Ng V (2017b) Detecting missing information in bug descriptions. In: Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering, ACM, New York, NY, USA, ESEC/FSE 2017, pp 396–407

Chaparro O, Florez JM, Marcus A (2019) Using bug descriptions to reformulate queries during text-retrieval-based bug localization. Empirical Software Engineering pp 1–61

Devore JL, Farnum N (1999) Applied Statistics for Engineers and Scientists. Duxbury

Dit B, Revelle M, Gethers M, Poshyvanyk D (2013) Feature location in source code: a taxonomy and survey. Journal of Software: Evolution and Process 25(1):53–95

Gay G, Haiduc S, Marcus A, Menzies T (2009) On the use of relevance feedback in ir-based concept location. In: Proceedings of the 25th IEEE International Conf. on Software Maintenance, Edmonton, Canada, pp 351–360

Grissom RJ, Kim JJ (2005) Effect Sizes for Research: A Broad Practical Approach, 2nd edn. Lawrence Earlbaum Associates

Haiduc S, Bavota G, Oliveto R, Marcus A, De Lucia A (2012) Evaluating the specificity of text retrieval queries to support software engineering tasks. In: Proceedings of the 34th IEEE/ACM International Conference on Software Engineering (ICSE'12), Zurich, Switzerland, pp 1273–1276

Haiduc S, Bavota G, Marcus A, Oliveto R, De Lucia A, Menzies T (2013) Automatic query reformulations for text retrieval in software engineering. In: Proceedings of the 35th ACM/IEEE International Conference on Software Engineering (ICSE'13), IEEE, San Francisco, CA, USA, pp 842–851

Herzig K, Zeller A (2013) The impact of tangled code changes. In: Proceedings of the 10th IEEE Working Conference on Mining Software Repositories (MSR'13), IEEE, San Francisco, CA, USA, pp 121–130

Kawrykow D, Robillard MP (2011) Non-essential changes in version histories. In: Proceedings of the 33rd IEEE/ACM International Conference on Software Engineering (ICSE,11), IEEE, Waikiki, HI, USA, pp 351–360

Kim M, Lee E (2019) A novel approach to automatic query reformulation for ir-based bug localization. In: Proc. of the 34th ACM/SIGAPP Symposium on Applied Computing, ACM, New York, NY, SAC '19, pp 1752–1759

Kochhar PS, Tian Y, Lo D (2014) Potential biases in bug localization: Do they matter? In: Proc. of the 29th ACM/IEEE International Conf. on Automated Software Engineering (ASE'14), ACM, Vasteras, Sweden, pp 803–814

Lawrie D, Binkley D (2018) On the value of bug reports for retrieval-based bug localization. In: 2018 IEEE International Conference on Software Maintenance and Evolution (ICSME), IEEE, pp 524–528

Linstead E, Bajracharya S, Ngo T, Rigor P, Lopes C, Baldi P (2008) Sourcerer: Mining and searching internet-scale software repositories. Data Mining and Knowledge Discovery 18(2):300–336

Lukins SK, Kraft NA, Etzkorn LH (2008) Source code retrieval for bug localization using latent dirichlet allocation. In: Proceedings of the 15th IEEE Working Conference on Reverse Engineering (WCRE'08), Koblenz-Landau, Germany, pp 155–164

Lukins SK, Kraft NA, Etzkorn LH (2010) Bug localization using latent dirichlet allocation. Information and Software Technology 52(9):972–990

Marcus A, Antoniol G (2012) On the use of text retrieval techniques in software engineering. In: Proc. of the 34th IEEE/ACM International Conf. on Software Engineering (ICSE'12), Technical Briefing, IEEE, Zurich, Switzerland

Marcus A, Sergeyev A, Rajlich V, Maletic JI (2004) An information retrieval approach to concept location in source code. In: Proceedings of the 11th IEEE Working Conference on Reverse Engineering (WCRE'04), IEEE, Delft, The Netherlands, pp 214–223

McMillan C, Grechanik M, Poshyvanyk D, Xie Q, Fu C (2011) Portfolio: Finding relevant functions and their usage. In: Proceedings of the 33rd ACM/IEEE International Conference on Software Engineering (ICSE'11), ACM, Waikiki, HI, USA, pp 111–120

Mills C (2019) Replication package. URL `http://www.cs.fsu.edu/~serene/mills2019-emse-bugs/`

Mills C, Bavota G, Haiduc S, Oliveto R, Marcus A, Lucia AD (2017) Predicting query quality for applications of text retrieval to software engineering tasks. ACM Transactions on Software Engineering and Methodology (TOSEM) 26(1):3

Mills C, Pantiuchina J, Parra E, Bavota G, Haiduc S (2018) Are bug reports enough for text retrieval-based bug localization? In: 2018 IEEE International Conference on Software Maintenance and Evolution, ICSME 2018, Madrid, Spain, September 23-29, 2018, pp 381–392

Mitchell M (1998) An Introduction to Genetic Algorithms. MIT Press, Cambridge, MA, USA

Poshyvanyk D, Marcus A, Dong Y, Sergeyev A (2005) Iriss-a source code exploration tool. In: Proceedings of the 21st IEEE International Conference on Software Maintenance (ICSM'05), IEEE, Budapest, Hungary,, pp 25 – 30

Rahman MM, Roy CK (2017) Improved query reformulation for concept location using coderank and document structures. In: Proceedings of the 32nd IEEE/ACM International Conference on Automated Software Engineering, IEEE Press, pp 428–439

Rahman MM, Roy CK (2017) Strict: Information retrieval based search term identification for concept location. In: Proceedings of the 24th IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER'17), pp 79–90, DOI 10.1109/SANER.2017.7884611

Rahman MM, Roy CK (2018) Improving ir-based bug localization with context-aware query reformulation. In: Proc. of the 26th ACM Joint Meeting on European Software Engineering Conf. and Symposium on the Foundations of Software Engineering, ACM, New York, NY, USA, pp 621–632

Rao S, Kak A (2011) Retrieval from software libraries for bug localization: A comparative study of generic and composite text models. In: Proceedings of the 8th IEEE Working Conference on Mining Software Repositories (MSR'11), ACM, Waikiki, HI, USA, pp 43–52

Razzaq A, Wasala A, Exton C, Buckley J (2018) The state of empirical evaluation in static feature location. ACM Transactions on Software Engineering and Methodology (TOSEM) 28(1):2

Roldan-vega M, Mallet G, Hill E, Fails JA (2013) Conquer: A tool for nl-based query refinement and contextualizing source code search results. In: Proceedings of the 29th IEEE International Conference on Software Maintenance (ICSM'13), IEEE, Eindhoven, The Netherlands, pp 512 — 515

Saha RK, Lease M, Khurshid S, Perry DE (2013) Improving bug localization using structured information retrieval. In: Proceedings of the 28th IEEE International Conference on Automated Software Engineering (ASE'13), IEEE, Palo Alto, CA, USA, pp 345–355

Salton G, Wong A, Yang CS (1975) A vector space model for information retrieval. Communications of the ACM 18(11):613–620

Savage T, Revelle M, Poshyvanyk D (2010) Flat3: Feature location and textual tracing tool. In: Proceedings of the 32nd IEEE/ACM International Conference on Software Engineering (ICSE'10), IEEE, Cape Town, South Africa, vol 2, pp 255–258

Shepherd D, Fry Z, Gibson E, Pollock L, Vijay-Shanker K (2007) Using natural language program analysis to locate and understand action-oriented concerns. In: Proc. of the 6th International Conf. on Aspect Oriented Software Development, ACM, Vancouver, Canada, pp 212–224

Shepherd D, Damevski K, Ropski B, Fritz T (2012) Sando: An extensible local code search framework. In: Proceedings of the 20th ACM SIGSOFT International Symposium on the Foundations of Software Engineering (FSE'12), ACM, Cary, NC, USA, pp 15:1–15:2

Strasser H, Weber C (1999) On the asymptotic theory of permutation statistics. Mathematical Methods of Statistics 2

Wang Q, Parnin C, Orso A (2015) Evaluating the usefulness of ir-based fault localization techniques. In: Proc. of the 24th International Symposium on Software Testing and Analysis, ACM, Baltimore, MD, USA, pp 1–11

Wang S, Lo D (2014) Version history, similar report, and structure: Putting them together for improved bug localization. In: Proceedings of the 22nd IEEE International Conference on Program Comprehension (ICPC'14), IEEE, Hyderabad, India, pp 53–63

Wang S, Lo D, Lawall J (2014) Compositional vector space models for improved bug localization. In: Proceeedings of the 30th IEEE International Conf. on Software Maintenance and Evolution, IEEE, Victoria, Canada, pp 171–180

Wong SKM, Ziarko W, Wong PCN (1985) Generalized vector spaces model in information retrieval. In: Proceedings of the 8th Annual International ACM SIGIR Conference on Research and Development in Information Retrieval, ACM, New York, NY, USA, SIGIR '85, pp 18–25

Ye X, Bunescu R, Liu C (2015) Mapping bug reports to relevant files: A ranking model, a fine-grained benchmark, and feature evaluation. IEEE Transactions on Software Engineering 42(4):379–402

Youm KC, Ahn J, Lee E (2017) Improved bug localization based on code change histories and bug reports. Information and Software Technology 82:177–192

Zhao W, Zhang L, Liu Y, Sun J, Yang F (2006) SNIAFL: Towards a static non-interactive approach to feature location. ACM Transactions on Software Engineering and Methodologies 15(2):195–226

Zhou J, Zhang H, Lo D (2012) Where should the bugs be fixed?-more accurate information retrieval-based bug localization based on bug reports. In: Proceedings of the 34th IEEE International Conference on Software Engineering (ICSE'12), IEEE, Zurich, Switzerland, pp 14–24