Noname manuscript No.
(will be inserted by the editor)

The Forgotten Role of Search Queries in IR-based Bug Localization: An Empirical Study

Mohammad M. Rahman · Foutse Khomh ·
Shamima Yeasmin · Chanchal K. Roy

the date of receipt and acceptance should be inserted later

Abstract Being light-weight and cost-effective, IR-based approaches for bug localization have shown promise in finding software bugs. However, the accuracy of these approaches heavily depends on their used bug reports. A significant number of bug reports contain only plain natural language texts. According to existing studies, IR-based approaches cannot perform well when they use these bug reports as search queries. On the other hand, there is a piece of recent evidence that suggests that even these natural language-only reports contain enough good keywords that could help localize the bugs successfully. On one hand, these findings suggest that natural language-only bug reports might be a sufficient source for good query keywords. On the other hand, they cast serious doubt on the query selection practices in the IR-based bug localization. In this article, we attempted to clear the sky on this aspect by conducting an in-depth empirical study that critically examines the state-of-the-art query selection practices in IR-based bug localization. In particular, we use a dataset of 2,320 bug reports, employ ten existing approaches from the literature, exploit the Genetic Algorithm-based approach to construct optimal, near-optimal search queries from these bug reports, and then answer three research questions. We confirmed that the state-of-the-art query construction approaches are indeed not sufficient for constructing appropriate queries (for bug localization) from certain natural language-only bug reports. However, these bug reports indeed contain high-quality search keywords in their texts even though they might not contain explicit hints for localizing bugs (e.g., stack traces). We also demonstrate that optimal queries and non-optimal queries chosen from bug report texts are significantly different in terms of several keyword characteristics (e.g., frequency, entropy, position, part of speech). Such an analysis has led us to four actionable insights on how to choose appropriate keywords from a bug report. Furthermore, we demonstrate 27%–34% improvement in the performance of non-optimal queries through the application of our actionable insights to them. Finally, we summarize our study findings with future research directions (e.g., machine intelligence in keyword selection).

Mohammad Masudur Rahman[†], Foutse Khomh[$], Shamima Yeasmin[⋆], Chanchal K. Roy[⋆]
Dalhousie University[†], Polytechnique Montreal[$], University of Saskatchewan[⋆], Canada
E-mail: masud.rahman@dal.ca, foutse.khomh@polymtl.ca, shamima.yeasmin@usask.ca, chanchal.roy@usask.ca

Keywords Debugging automation, bug localization, information retrieval, natural language processing, query construction, keyword selection, genetic algorithm, optimal search query, poor search query, empirical study

1 Introduction

Software bugs and errors could lead to massive fatalities (e.g., Boeing-737 MAX 8 crashes[1], Therac-25 accidents[2]) and loss of trillions of dollars every year [1, 3, 86]. Finding and fixing these bugs and errors (a.k.a., software debugging) cost about 50% of the development time and efforts [3]. One of the most challenging and time consuming steps of software debugging is *bug localization* (i.e., locating the bugs within the code) [37, 51, 86]. Over the last two decades, Information Retrieval (IR) methods have been widely adopted in bug localization [47, 65, 73, 85]. The IR-based localization leverages the lexical similarity between bug reports (queries) and source code documents to find out the bugs [85]. Such a localization is reported to be light-weight, cost-effective, and also as accurate as spectra-based techniques which analyse software execution traces for the localization [62, 72]. However, recent findings [36, 59, 72] raise a major concern about the effectiveness of the IR-based localization. About 55% of the bug reports contain plain natural language texts without providing any explicit hints for the bug localization (e.g., program elements, stack traces) [59, 72]. Based on a developer study, Wang et al. [72] report that the IR-based approaches (e.g., BugLocator [85]) cannot perform well when they use these bug reports as search queries. Such *natural language-only* reports are also referred to as *non-localized* bug reports [36] or bug reports *without identifiable information* [72]. The findings and conclusions of the developer study mentioned above have also been partially supported by another empirical study [59].

One of the key challenges of any IR-based solution is to select an appropriate search query that reflects the information need. In the case of bug localization, an appropriate query should capture the right keywords (e.g., characteristics of a bug) that can pinpoint the faulty code. Since non-localized bug reports lack explicit localization hints (e.g., program elements), many existing approaches [47, 65, 73, 74, 85] might fail to construct appropriate queries from them. Thus, the empirical and developer studies above [36, 72] might have used sub-optimal search queries, which possibly led to their poor performance in the IR-based localization. However, a more recent study [46] suggests that bug reports alone contain sufficient keywords (other than the localization hints) which (1) could form the appropriate search queries and (2) thus in turn could deliver optimal performance in localizing the bugs. That is, even the *natural language-only* bug reports might contain high-quality search keywords. In other words, natural language-only bug reports might be a source for good query keywords, which in essence contradicts the findings of the past empirical studies [36, 72]. This recent result showing that bug reports often contain optimal queries, and the fact that existing IR-based localization approaches often do not perform well prompts us to question the effectiveness of the query selection practices in these IR-based localization approaches. A better understanding of the strengths and weaknesses of current query construction tech-

[1] https://goo.gl/GwXv6H
[2] https://bit.ly/2KU9IR2

niques and the characteristics of optimal queries could help us build more effective, reliable, and practical tools for the IR-based bug localization.

Software developers often use a few important keywords from a bug report as a search query for bug localization [70]. Unfortunately, even the experienced developers struggle to choose the right keywords [18, 32, 41]. In fact, Kevic and Fritz [32] reports that developers might fail to choose the right keywords from the bug reports up to 88% of time. Several existing studies attempt to (1) deliver appropriate queries from bug reports [15, 32, 57, 58] and (2) reformulate the poor queries chosen by developers [19, 25, 55, 56, 70]. A few other studies attempt to find out the best document retrieval algorithm [48, 50] for a given query. Unfortunately, in the literature, there is a marked lack of research that (1) investigates the hidden variables behind the issues of *natural language-only* bug reports, and (2) critically examines the state-of-the-art practices for query selection in the IR-based bug localization. By examining these important aspects, one might gain valuable insights on whether these bug reports can deliver good queries for accurately localizing bugs or not (as alluded by the past studies [36, 58, 72]).

In this article, we conduct an empirical study using 2,320 bug reports (939 natural language-only + 1,381 natural language texts and localization hints), and *ten* existing approaches on search query construction [13, 31, 32, 56, 57, 63, 64, 70], and critically examine the state-of-the-art query construction practices in the IR-based bug localization. We perform three different analyses in our empirical study. First, we compare between two major keyword selection methodologies (*frequency-based* [25, 31, 32, 70] and *graph-based* [56, 57]) that are widely used to construct search queries for localizing bugs with Information Retrieval. Second, we employ a Genetic Algorithm-based approach [46] to construct optimal, near-optimal search queries from the bug report texts. We define an *optimal query* as a set of search keywords that achieve the optimal performance in bug localization, i.e., retrieves the first buggy document at the topmost position of the ranked result list. We demonstrate how optimal or near-optimal performance could have been achieved in localizing bugs if the appropriate search queries were provided. Third, we contrast between optimal and non-optimal search queries constructed from the bug reports using a detailed analysis involving machine learning, data mining, and manual inspection. Our analysis demonstrates that optimal and non-optimal queries are significantly different from each other in several aspects, which indicates that there is room for improvement in query formulation techniques (e.g., application of machine intelligence in recognizing optimal keywords). In particular, we answer three research questions in this empirical study as follows.

(a) **$\mathbf{RQ_1}$: How do the state-of-the-art approaches perform in identifying appropriate search keywords from software artifacts (e.g., bug reports) for IR-based bug localization?**
We investigate two major keyword selection methodologies –*frequency-based* and *graph-based*, and compare their strengths and weaknesses. We first divide our bug reports into four subsets based on their baseline queries and localization hints, and then extract the queries from each of them using the existing approaches. Queries from these approaches achieve a maximum of 68% Hit@10, 43% mean average precision and 0.43 mean reciprocal rank, which are comparable to baseline measures (Table 9). Graph-based approaches were found more promising than the frequency-based ones. However, as a whole, the state-

of-the-art approaches were not sufficient to make appropriate queries from the bug reports that lead to poor baseline queries. We also found that some bug reports could lead to poor search queries even if they contain the potential hints for localizing software bugs (e.g., program elements, stack traces) in their texts (Table 6), which was intriguing.

(b) **$\mathbf{RQ_2}$: Can optimal, near-optimal search queries be constructed from the bug reports that lack bug localization hints or simply contain natural language only texts? How do these search queries perform in the IR-based bug localization?**
Optimal search queries (that deliver optimal performance) could be constructed from **50%–81%** of the bug reports that might neither contain any explicit localization hints (e.g., program elements, stack traces) nor lead to any good baseline queries. We use Genetic Algorithm (GA) and ground truth information to construct these queries. It should be noted that this GA-based query construction approach cannot be used in a practical setting where the ground truth is not known beforehand. Our analysis also suggests that **78%–93%** Hit@10, **57%–86%** mean average precision and **0.58–0.85** mean reciprocal rank could be achieved by the near-optimal queries constructed from some bug reports whereas their baseline queries (i.e., preprocessed bug reports) simply fail to retrieve any relevant documents within their Top-10 result positions (e.g., **0.00%** Hit@10) (Table 13).

(c) **$\mathbf{RQ_3}$: How optimal, near-optimal, and non-optimal search queries differ from each other in their characteristics and performances?**
We compare optimal and near-optimal search queries with non-optimal queries collected from each of our selected bug reports. We conduct a multi-modal comparative analysis that involves machine learning, data mining, and careful manual inspection. Our analysis reveals several major insights. First, optimal search keywords are relatively less frequent than non-optimal ones in a bug report. Second, they are more specific (i.e., less ambiguous) than those in the non-optimal queries. Third, optimal keywords are more prevalent within the *description* section in a bug report. Fourth, they are more likely to be nouns than of other part of speech. We employed these actionable insights to baseline queries and the queries from the state-of-the-art technique [57] (i.e., non-optimal queries) and then significantly improve them. Although the poor baseline queries fail to retrieve any results within their Top-10 positions, our insight-based query expansion improved their Hit@10 by up to **27%–34%**, which clearly demonstrates the significant benefits of our derived insights.

Novelty in contribution: Our work makes significant contributions to the literature in several aspects. First, we demonstrate that even the state-of-the-art IR-based localization approaches fail to construct the right search queries from the bug reports that lead to poor baseline queries ($\mathbf{RQ}_1$). Bug reports could provide high-quality or poor search queries irrespective of their localization hints (e.g., program elements, stack traces). Second, graph-based approaches perform consistently higher than frequency-based approaches in selecting queries both from bug reports and from source code ($\mathbf{RQ}_1$). Third, although the state-of-the-art approaches fail, natural language-only bug reports actually contain high-quality search keywords in their texts, which could be identified using a Genetic Algorithm

Table 1: An Example Natural Language-Only Bug Report (#229380, eclipse.jdt.debug)

Field	**Content**
Title	Up/Down buttons incorrectly enabled on classpath tab
Description	(1) Open a Java Launch Configuration. (2) Go to the classpath tab. (3) Ensure there is at least one bootstrap entry and one user entry. (4) Select the bottom bootstrap entry, the DOWN button is enabled, but pressing it does nothing. (5) Select the top user entry, the UP button is enabled, but pressing it does nothing. We should be able to update the button to reflect whether moving it is possible.

Table 2: Example Queries from the Bug Report of Table 1

Technique	**Search Query**	**QE**
Baseline-I	{*title*}	138
Baseline-II	{*description*}	43
Baseline	{*title* + *description*}	25
Kevic and Fritz [32]	{**button** enabled tab entry classpath bootstrap}	93
TF-IDF [31]	{**button** entry bootstrap enabled incorrectly moving}	177
Rahman and Roy [57]	{tab classpath enabled **buttons** user entry}	86
NrOptimal$_{\textbf{GA}}$	{*open reflect tab bottom entry classpath*}	**01**
	Impact of Insight-Based Noise Filtration	
Kevic and Fritz	{enabled tab entry classpath bootstrap}	**70**
TF-IDF	{entry bootstrap enabled incorrectly moving}	**141**
Rahman and Roy	{tab classpath enabled user entry}	**76**

QE = Rank of the first buggy result returned by a query

(GA) and ground truth information (**RQ**$_2$). This result strengthens the earlier finding of Mills et al. [46]. Fourth, although the GA-based approach might not be applied in a real-world bug localization scenario, it allows identifying insights that can be leveraged to improve keyword selection. We found that optimal search queries are significantly different from non-optimal ones in several characteristics (e.g., keyword position, frequency, ambiguity), which were previously unknown. Furthermore, these insights were also found actionable and effective according to our investigation where we improved the non-optimal baseline and state-of-the-art queries by applying our insights (**RQ**$_3$).

Structure of the article: The rest of the article is organized as follows – Section 2 motivates our research challenges, Section 3 provides a background overview, and Section 4 discusses our study set up. Sections 5, 6, 7 answer our three research questions, and Section 8 summarizes our findings from the empirical and manual analysis. Section 9 discusses threats to the validity, Section 10 focuses on related work, and finally Section 11 concludes the article with future work.

2 Motivating Example

In order to contrast among baseline, state-of-the-art and optimal search queries, we provide a motivating example. Table 1 shows a natural language-only bug report that does not contain any hints for localizing bugs (e.g., program elements, stack traces). Table 2 shows multiple search queries constructed from this bug report. We see that the baseline queries from *title*, *description* or *title* + *description* fields do not perform well. They return their first buggy documents at the 138th, 43rd and 25th positions respectively within their result lists. On the contrary, an optimal query selected by the Genetic Algorithm (NrOptimal$_{GA}$) from the bug report returns the same buggy document at the **topmost position**, which is the best possible outcome. Finally, the state-of-the-art approaches in query construction [31, 32, 57] achieve the best rank of 86, which is far from ideal.

We compare optimal queries with non-optimal queries and found that the optimal search keywords are often found within the *description* section of a bug report and might always not be *frequent* (see details in Table 20, Fig. 7). Such insights might explain why one of the frequent keywords of the bug report – *"button"* – is not a part of the optimal query. Furthermore, when this keyword was removed from the above queries, we notice a performance improvement (i.e., lower QE) in each of the three existing approaches– Kevic and Fritz [32], TF-IDF [31] and Rahman and Roy [57], which suggests the benefit of this insight.

To summarize, even the state-of-the-art approach fails to provide an appropriate search query from a natural language-only bug report. Thus, existing IR-based localization approaches used in the past empirical studies [36, 72] were possibly not provided with the *optimal* search queries from bug reports, which possibly led to their low performance in the bug localization. All these findings presented above clearly point out a gap in the literature (concerning search query construction) that warrants further researches and investigations.

3 Background

3.1 Term Weighting

Determining the relative importance of a term within a textual document is commonly known as *term weighting* [6, 26]. Although the underlying concepts and algorithms were introduced by the IR community, term weighting has been frequently used to develop IR-based solutions for Software Engineering problems (e.g., code search, bug localization) [8, 25, 58, 70]. Two term weighting methodologies are frequently adopted in Software Engineering contexts – *frequency based* and *graph-based*– as follows.

(a) Frequency-Based Term Weighting: TF-IDF is a frequency-based method that is widely adopted by the literature for term weighting. TF-IDF stands for Term Frequency (TF) × Inverse Document Frequency (IDF), and it can be calculated using the following equation.

$$\text{TF-IDF (t,d)} = (1 + log(f_{t,d})) \times log(\frac{|D|}{n_t} + 0.01) \tag{1}$$

Here $f_{t,d}$ refers to the frequency of a term t in the document d, n_t refers to the number of documents containing the term t, and D is the set of all documents in the corpus. That is, if a term is frequent within a document but not so frequent in other documents across the corpus, then this term is considered to be *important* (e.g., search keyword) with respect to the target document. TF-IDF adopts the notion of *term independence.* That is, it does not consider the impact of contexts (e.g., surrounding terms) upon a given term in determining the importance of the term [39]. However, contexts play a major role in determining the term semantics

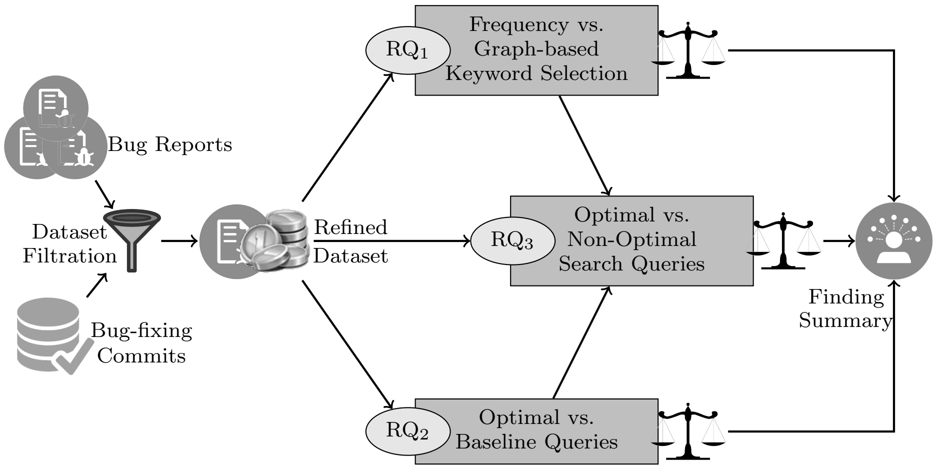
Frequency vs. Graph-based Keyword Selection
RQ$_1$
Bug Reports
Dataset Filtration
Refined Dataset
Optimal vs. Non-Optimal Search Queries
RQ$_3$
Finding Summary
Bug-fixing Commits
Optimal vs. Baseline Queries
RQ$_2$

Fig. 1: Schematic diagram of our empirical study

and hence, in the term importance as well [44, 82]. Thus, TF-IDF could be limited in the term importance estimation [10, 43]. However, due to its simplicity and low costs, several studies [25, 32] adopt TF-IDF in term weighting for Software Engineering problems (e.g., bug localization, concept location).

(b) Graph-Based Term Weighting: Unlike TF-IDF, several studies capture dependencies among the terms within a body of texts (e.g., bug report) using term co-occurrences [43] and syntactic relations [10]. First, they transform each text document into a graph where the nodes represent the distinct words and the connecting edges refer to the dependencies among the words from the document. Second, they adapt Google's PageRank algorithm [11] for natural language texts, and determine the relative weight $S(v_i)$ of each of the words v_i using the recursive score computation as follows:

$$S(v_i) = (1-\phi) + \phi \times \sum_{j \epsilon V(v_i)} \frac{S(v_j)}{|V(v_j)|} \quad (0 \leq \phi \leq 1) \tag{2}$$

Here, $V(v_j)$ refers to the list of nodes that are connected to v_i, and ϕ is the damping factor. The connecting edges among the nodes could be uni-directional or bi-directional. In the context of web link analysis, Brin and Page [11] define ϕ as the probability of a random surfer of staying on a web page, and $1-\phi$ as the probability of jumping off the page. PageRank relies on a voting mechanism, and computes score of each node based on the scores from connected nodes within the graph. Several studies in Software Engineering [56, 57, 58] use PageRank for term weighting and keyword selection.

In our empirical study, we compare between these two major term weighting approaches above, and analyse their relative strengths and weaknesses in the query construction for IR-based bug localization.

Table 3: Study Dataset (Subject Systems & Bug Reports)

Subject System	**#BR_{H-}**	**#BR_{H+}**	**#Files**	**#Methods**	**#AGT**
`ecf`	58	141	2,802	21.5K	1.74
`eclipse.jdt.core`	53	95	5,908	66.3K	2.03
`eclipse.jdt.debug`	141	186	1,532	15.7K	1.76
`eclipse.jdt.ui`	293	286	10,927	57.4K	1.78
`eclipse.pde.ui`	332	188	5,334	31.8K	1.97
`tomcat70`	62	485	1,841	23.8K	1.27
	939	**1,381**			
Total Bug Reports: 2,320					

BR_{H+}=Bug reports with hints, **BR_{H-}**=Bug reports without hints, **AGT**=Average number of ground truth files

3.2 Genetic Algorithm (GA)

Genetic Algorithms are a class of evolutionary search that is inspired by biological operations such as *natural selection*, *cross-over* and *mutation*. These algorithms are widely used to solve complex optimization problems in various research domains including Software Engineering (e.g., automatic patch generation [21]). The whole texts (i.e., *title* + *description*) of a bug report are often verbose and non-optimal as a search query [14]. Thus, carefully chosen keywords from the report texts could form an optimal or near-optimal search query for IR-based bug localization [46]. In other words, query construction could be considered as a kind of optimization problem. Genetic Algorithms equipped with an appropriate fitness function are a great fit for this optimization task. In our empirical study, we employ Genetic Algorithm with Query Effectiveness (QE) as a fitness function [46], and construct near-optimal search queries from the bug reports.

4 Study Setup

Fig. 1 shows the schematic diagram of our empirical study. We collect dataset from an existing benchmark [58], refine it by discarding the noisy items (e.g., tangled commits, false-positive bug reports), and then answer three research questions through comparative analyses. In this section, we discuss our dataset construction, noise filtration, and study setup as follows.

4.1 Dataset Preparation

Bug Report Selection: Table 3 shows the dataset for our empirical study. We first collect a total of 1,546 bug reports from six open source, Java-based subject systems for our study. They are taken from a publicly available *benchmark dataset* [4, 58, 59]. Several steps were taken by the benchmark authors to construct this dataset. First, all the bug reports marked as RESOLVED were extracted from the bug repository (e.g., BugZilla, JIRA) of each system. Second, they consulted with the version control history of each system at GitHub, and collected the bug-fixing commits (i.e., commits solving the bugs) using a set of traditional heuristics [7, 85]. In particular, a set of regular expressions (e.g.,`(B|b)ug\s+\d+|\=\d+| <Repo>-\d+`)

were employed for identifying these commits. For example, `(B|b)ug\s+` recognizes this commit[3] as the bug-fixing commit of the showcase bug report in Table 1 (e.g., Bug 229380). Then only such bug reports were retained that have corresponding bug-fixing commits. Third, they also detected the presence of structured entities namely *bug localization hints* (e.g., stack traces, method invocations) using regular expressions in the report texts [47, 58], and retained such bug reports that contain only unstructured regular texts and no structured entities.

Regular expression for stack traces:

```
(.*)?(.+)\.(.+)(\((.+)\.java:\d+\)|\(Unknown Source\)|\(Native Method\))
```

Regular expression for method invocations:

```
((\w+)?\.[\s\n\r]*[\w]+)[\s\n\r]*(?=\(.*\))|([A-Z][a-z0-9]+){2,}
```

According to Wang et al. [72], developers often struggle to find appropriate search keywords from the reports containing only regular texts and no structured entities. Fourth, they also discarded such bug reports (1) for which no source code documents (e.g., Java classes) were changed, or (2) that their changed documents were missing in the current snapshot of the subject system.

Construction of Ground Truth: We collect the changed source documents (a.k.a., *change set*) from each of the bug-fixing commits, and use them as the *ground truth* for corresponding bug reports. When multiple commits are made for a single bug report, their changed document lists are merged together to construct the final ground truth. Thus, all three items – bug reports, their corresponding ground truths and the subject systems – are collected from the benchmark dataset.

Dataset Cleansing with Manual Analysis: Recent findings [35, 36] suggest that existing dataset used for IR-based bug localization could be noisy due to misclassified bug reports (a.k.a., feature requests). In order to mitigate such a threat, we conduct an extensive manual analysis and perform further data cleansing and noise removal. First, two authors individually go through *title* and *description* of each bug report, and annotate whether it indicates a software bug or a new feature [35, 36]. Second, we perform an agreement analysis between the two authors for all six subject systems. We reached an average agreement level of 74%. While 26% (100%-74%) is a significant disagreement between the two authors, such disagreement in our classification might also be explained. Bug reports collected from the benchmark mostly contain natural language texts rather than structured elements (e.g., stack traces). Both authors carefully analyse the textual contents to understand their hidden semantics and then classify them as either bug reports or feature requests. Since the same words/texts could be explained differently by different people based on their subjective perceptions, such disagreement might be expected. In fact, similar disagreements among the human experts were also observed in various other text processing tasks such as search keyword selection [18] or textual summary generation [22, 49]. We also performed retrospective analysis and resolved the disagreements between the two authors through cross-examination, further manual analysis, and extensive discussions. During this analysis, we toggled the class labels of 192 bug reports/feature requests. Finally,

[3] https://bit.ly/2RnIAPK

we retained only such documents that were labelled as *bug reports* by both authors, and discarded the rest from our dataset.

According to recent findings [27, 79], the ground truth of bug reports could be bloated due to *tangled commits*. In particular, the bug-fixing commits might contain changes irrelevant to the fixed bugs. Such bug-fixing commits are called *tangled commits*. In order to mitigate such a threat, we conduct an extensive manual analysis on the collected ground truth. First, we manually analyse only the large commits containing more than *five* changed source documents. Out of 1,546 bug reports, we manually analysed the bug-fixing commits of 333 (22%) bug reports. In particular, we manually check for (1) changes irrelevant to the bug and (2) trivial changes (e.g., updating code comments), and discard them from the dataset. We identified 78 commits as *"tangled"* which were discarded from further analysis. In order to mitigate the threat in the remaining commits containing five or less changed source documents, we conducted further analysis. After discarding false-positive bug reports (i.e., feature request), we identified 1,149 bug reports with five or less ground truth documents. Since manually analyzing all of them could be very costly, we (1) selected 10% bug reports from each of the six subject systems using stratified random sampling, and (2) manually analyzed 112 bug-fixing commits. Out of these 112 cases, we found only 5 tangled commits which are only ≈5%. Thus, the impact of tangled commits on our findings might be negligible. An earlier study [36] has also reported negligible impact of the tangled commits on bug localization performance. Furthermore, to assess the potential impact of tangled commits on our results, we replicated the analysis on a manually validated dataset, and the detailed analysis results can be found in Table 5.

Once both the filtration steps above were completed, we ended up with a refined dataset of **939** bug reports and their ground truths for our study. We spent **60+** man-hours in conducting our manual analysis.

Dataset Extension: Although the above analysis provides a total of 939 natural language-only bug reports, we further extend our dataset with such bug reports that contain explicit localization hints (e.g., stack traces, method invocations). We use the same benchmark dataset [58] as above. Our goal was to extend our experiments and generalize our findings. In particular, we collect such *localized bug reports* (i.e., containing localization hints) that have a single ground truth source file. We believe that commits modifying only one source file are less likely to contain *irrelevant changes*. Our idea was to avoid the threat of *tangled* commits. Thus, we extend our dataset with an additional 1,381 bug reports from the six subject systems, which provides a total of **2,320** bug reports for the study.

Replication Package: Our dataset, replicated techniques and other associated materials are *publicly available* [2] for the replication and third-party reuse.

4.2 Selection of Baseline Search Queries

Developers often use *title* and *description* fields from a bug report as ad hoc search queries for locating the buggy code within a software system. Existing approaches [25, 47, 58, 70] also employ them as the baseline queries for their evaluation and validation. We thus construct three baseline queries using these fields – **Baseline-I** (i.e., *title*), **Baseline-II** (i.e., *description*) and **Baseline** (i.e., *title* + *description*) – for our empirical study. In particular, we perform standard natural language preprocessing (e.g., removal of stop words and punctuations, token splitting, camel case splitting) on these fields, and use their preprocessed versions as the baseline search queries for our study. We avoid stemming due to its mixed findings, as reported in the literature [29, 32]. Stemming might improve recall but often reduces the precision of search queries, which is important in our case. It should be noted that the same pre-processing steps were used for both bug reports and the source code documents in the corpus. Table 7 shows the statistics on keyword counts in the baseline queries.

4.3 Code Search Engine

In order to evaluate the performance of our search queries, we use *Apache Lucene* [5], a popular document search engine that has been widely adopted by both academia [25, 47, 48, 57] and industry (e.g., ElasticSearch, Stack Overflow). We capture each source code document of a subject system, perform limited natural language preprocessing (e.g., removal of stop words, keywords and punctuations, token splitting, camel case splitting) on these documents, and then use them to construct our corpus. Lucene then develops a document index against each subject system by analysing its preprocessed corpus documents. Once a search query is issued, Lucene (1) first selects a list of candidate results using Boolean Search Model, and then (2) delivers a ranked list of relevant results using a Vector Space Model (VSM)-based search algorithm (e.g., BM25 [68]).

4.4 Performance Metrics

We employ four state-of-the-art performance metrics for evaluating the search queries in our empirical study. Since these metrics have been widely adopted earlier by relevant literature on IR-based bug localization [47, 58, 65] and query reformulation [25, 56, 57], their use is also justified for this study. We define the four performance metrics as follows.

Hit@K: It calculates the fraction of search queries (e.g., bug reports) for each of which at least one ground truth is retrieved within the Top-K results. The higher the Hit@K value is, the better the search queries are. It is also called Recall @Top K in the literature [65].

Query Effectiveness (QE): It approximates a developer's effort in locating the reported bug within the source code of a software system [47]. In practice, the metric returns the rank of the first result that matches with the ground truth, within the ranked list. The underlying idea is to provide an accurate starting point to the developer that deals with the bug discussed in the search query. The lower the effectiveness value is, the better a query is since the developer then can locate the buggy source code more quickly with less manual efforts.

Mean Reciprocal Rank (MRR): Reciprocal Rank (RR) refers to the multiplicative inverse of the rank of the first buggy source code document correctly returned within the ranked result list. In practice, only Top-K results (e.g., $K = 10$) are often analysed for each query. Mean Reciprocal Rank (MRR) averages the RR measures for all search queries (Q) of a subject system. It can be defined as follows:

$$\text{MRR(Q)} = \frac{1}{|Q|}\sum_{q\in Q}\frac{1}{firstRank(q)}$$

Here, $firstRank(q)$ provides the rank of the first correctly retrieved buggy document. MRR can take a maximum value of 1.00 when the correct result is found at the top-most position and a minimum value of 0.00 when no correct results are found within the whole result list. The bigger the MRR value is, the better the search query is.

Mean Average Precision (MAP): Precision@K calculates the precision at the occurrence of every single buggy source document within the ranked list. Average Precision@K (AP) averages the Precision@K for all buggy documents in the list for a search query. Thus, Mean Average Precision (MAP) is calculated from the mean of Average Precision@K (AP) for all the queries (Q) of a system as follows:

$$\text{AP} = \frac{\sum_{k=1}^{D} P_k \times buggy(k)}{|S|},\ \text{MAP} = \frac{\sum_{q\epsilon Q}\text{AP(q)}}{|Q|}$$

Here, $buggy(k)$ determines whether k^{th} result in the ranked list is buggy (i.e., matches ground truth) or not (i.e., does not match ground truth). It returns either 0 (false positive) or 1 (true positive). P_k denotes the precision calculated at the k^{th} position, and D refers to the number of total results. S refers to the true positive instances retrieved by a query, and Q is the set of all queries (e.g., bug reports).

4.5 Bug Report Clustering

Our dataset contains a total of 2,320 bug reports where 939 of them are natural language only and 1,381 of them contain explicit bug localization hints (e.g., program elements, stack traces). Several studies [36, 72] suggest a significant impact of localization hints in the bug reports upon IR-based bug localization methods. For our empirical study, we further divide them into subsets based on the baseline queries they produce through the pre-processing steps (Section 4.2). In particular, if a baseline query (e.g., **Baseline**) achieves the query effectiveness (QE) between 1 and 10, then the query is already good. The corresponding bug report is considered as *bug report leading to high-quality queries* (**HQ**). On the other hand, a bug report with a poor baseline query (i.e., QE>10) is considered as *bug report leading to poor queries* (**LQ**). Thus, based on two different dimensions – baseline query performance and presence of localization hints (e.g., method invocations, stack traces), we divide our bug report collection into four different subsets. Dividing the bug reports into smaller categories like these can help us zoom in the issues of IR-based bug localization. Our idea was to check how the existing approaches perform with these four subsets and to derive further insights on how to make appropriate search queries from bug reports. Table 4 shows the statistics on bug report from each subset.

Table 4: Four Subsets of Bug Report Collection

Bug Report Category	**#BR**
Bug reports with high-quality baseline queries and without localization hints ($HQ_{\mathbf{H}-}$)	567
Bug reports with high-quality baseline queries and with localization hints ($HQ_{\mathbf{H}+}$)	954
Bug reports with low-quality baseline queries and without localization hints ($LQ_{\mathbf{H}-}$)	372
Bug reports with low-quality baseline queries and with localization hints ($LQ_{\mathbf{H}+}$)	427
Total	**2,320**

BR = Bug reports

5 Answering $\mathbf{RQ_1}$: Frequency-Based vs. Graph-Based Search Keyword Selection for IR-Based Bug Localization

5.1 Determination of State-of-the-Art on Keyword Selection

The underlying idea of any query construction/reformulation task is to choose appropriate search keywords from the suitable information sources (e.g., bug report, source code). According to existing literature [19, 25, 56, 57, 70], two keyword selection methodologies –*frequency-based* and *graph-based*– have been widely adopted for query construction in the context of concept/bug localization. We choose ten different approaches from these two major methodologies for our study as follows.

Frequency-Based Keyword Selection Methods: TF-IDF [31] has been a popular term weighting method in Information Retrieval for the last five decades. It is also widely adopted in Software Engineering contexts. Several existing studies [25, 32, 33, 46, 48] employ TF-IDF and its variants to identify important keywords from a body of texts (e.g., bug report, source code). Kevic and Fritz [32] use TF-IDF and three heuristics (e.g., POS tags, notation, position) to identify the important search terms from a bug report. Gay et al. [19] employ Rocchio's expansion [64] where they use TF-IDF in reformulating search queries for concept location. Haiduc et al. [25] later employ three frequency-based term weighting methods –Rocchio [64], RSV [63] and Dice [13], and deliver the best performing query keywords from the source code for concept location using machine learning. Sisman and Kak [70] leverage *spatial proximity* between query keywords and candidate keywords within the source code (a.k.a., spatial code proximity (SCP)), and return such candidates that frequently co-occur with the query. To the best of our knowledge, these are the state-of-the-art studies that adopt *frequency-based* keyword selection for IR-based concept/bug localization.

The above studies employ two different sources –*bug report* and *source code document*– for query construction. Thus, we use Kevic and Fritz [32] to select keywords from the bug reports, and Haiduc et al. [25] and Sisman and Kak [70] for selecting keywords from the source code. It should be noted that Haiduc et al. [25] combine three term weighting approaches –Rocchio, RSV and Dice. Thus, we select a total of *eight* frequency-based approaches (Tables 5, 9) in our study that use TF-IDF and its variants for keyword selection. Since the authors' implementations were not publicly available, we carefully implement each of these approaches in our working environment with their recommended settings and the best performing parameters (e.g., regression coefficients [32]). Please check our replication package for further details on the used settings and parameters.

Graph-Based Keyword Selection Methods: Although TF-IDF has been a widely adopted method, it fails to capture a term's contexts which is a major limitation [10, 43]. Several existing studies [56, 57, 58] attempt to overcome this issue by transforming a document into a graph representation. In particular, they represent the individual terms and the dependencies among them using nodes and connecting edges respectively within a graph. Rahman and Roy [57] capture semantic and syntactic dependencies among the words, employ PageRank algorithm [11] on the constructed graph, and then deliver appropriate search keywords from a bug report for concept location. While their approach (STRICT) is suitable for regular texts, bug reports could also be noisy; containing too many structured elements (e.g., stack traces, test cases). Thus, keywords from bug reports should be chosen carefully. Recently, Rahman and Roy leverage reporting quality dynamics, graph-based term weighting, and deliver search keywords even from the noisy bug reports [58]. Despite these attempts, constructing appropriate queries could be challenging since the bug reports might always not contain the necessary information. Thus, Rahman and Roy also make use of source document structures and their contexts, employ graph-based term weighting, and then suggest appropriate search keywords from the relevant source code [56]. To the best of our knowledge, these are the state-of-the-art studies that adopt *graph-based* keyword selection for IR-based concept location and IR-based bug localization.

Since we consider two keyword sources in our study, we use STRICT [57] to select keywords from bug reports and ACER [56] for selecting keywords from the source code. The authors' implementations for these approaches were publicly available. We thus use their prototypes for our empirical study.

5.2 Search Keyword Selection from Bug Reports

Tables 5, 6, 7, 8, and Fig. 2 summarize our comparative analysis between frequency-based keyword selection and graph-based keyword selection from the bug reports. Table 5 shows the bug localization performance of various queries constructed from a total of 2,320 bug reports. We see that the existing approaches are not sufficient enough for delivering appropriate keywords from the bug report texts. Simple TF-based keywords achieve 56% Hit@10 with 34% mean average precision and a mean reciprocal rank of 0.34. That is, term frequency (TF) might work only half of the time in returning a relevant result within the top 10 positions. TF-IDF-based search queries achieve 60% Hit@10, 37% precision and 0.37 reciprocal rank, which are marginally higher. However, they still cannot outperform the best performing baseline measures (e.g., 67% Hit@10, 43% MAP, and 0.43 MRR). Kevic and Fritz construct a regression model for keyword selection using TF-IDF and three popular heuristics concerning the position, notation and part of speech of the keywords. Unfortunately, their model also fails to deliver the right keywords and thus performs poorly. On the contrary, Rahman and Roy [57] propose a graph-based technique namely STRICT where they leverage co-occurrences and syntactic dependencies among the keywords for search keyword selection. Queries delivered by STRICT achieve 13% higher Hit@10 and 5% higher Hit@10 than that of the TF and TF-IDF approaches respectively. Thus, graph-based keyword selec-

Table 5: Frequency-Based vs. Graph-Based Keyword Selection from Bug Reports (Using Top-10 Results Only)

Technique	**Genre**	**Hit@1**	**Hit@5**	**Hit@10**	**MRR**	**MAP**
Retrieval performance using all bug reports (2,320)						
Baseline		**31.98%**	**57.96%**	**66.50%**	**0.43**	**42.69%**
Baseline-I	–	22.37%	46.73%	57.58%	0.33	32.82%
Baseline-II		29.01%	51.16%	60.43%	0.38	38.21%
TF		24.39%	46.23%	55.58%	0.34	33.94%
IDF	Frequency	24.77%	43.91%	52.79%	0.33	32.94%
TF-IDF		**27.02%**	49.92%	59.80%	**0.37**	36.76%
Kevic and Fritz		23.36%	44.03%	52.92%	0.32	31.98%
STRICT	Graph	25.82%	**52.28%**	**63.02%**	**0.37**	**37.31%**
Retrieval performance using manually analysed bug reports (175)						
Baseline		**29.65%**	**58.77%**	**63.77%**	**0.41**	**38.08%**
Baseline-I	–	21.45%	48.65%	60.00%	0.34	32.87%
Baseline-II		23.62%	45.79%	56.07%	0.33	30.40%
TF		20.75%	44.45%	56.13%	0.32	28.72%
IDF	Frequency	17.61%	38.90%	45.93%	0.26	24.63%
TF-IDF		24.37%	47.33%	55.98%	0.35	31.57%
Kevic and Fritz		26.17%	49.03%	**58.40%**	0.36	31.89%
STRICT	Graph	**26.60%**	**50.87%**	58.01%	**0.37**	**34.28%**
Retrieval performance using single-release bug reports (360)						
Baseline		**36.99%**	**58.55%**	**66.66%**	**0.46**	**46.29%**
Baseline-I	–	27.42%	48.67%	59.56%	0.37	36.40%
Baseline-II		35.28%	50.77%	57.00%	0.42	42.05%
TF		24.89%	50.74%	58.03%	0.35	34.60%
IDF	Frequency	29.56%	48.41%	54.26%	0.38	37.79%
TF-IDF		**30.06%**	**55.12%**	62.97%	**0.41**	**40.68%**
Kevic and Fritz		22.37%	44.46%	52.39%	0.31	29.82%
STRICT	Graph	28.54%	50.31%	**66.18%**	0.39	39.14%

Emboldened=Baseline and state-of-the-art performance measures

tion is marginally better than frequency-based keyword selection according to our experiment. Unfortunately, none of these existing approaches above offers a better alternative than the baseline query (i.e., pre-processed version of a bug report), which is unexpected. It should be noted that, in our study, a baseline query contains almost all keywords (except stop words) from a bug report whereas a search query suggested by the existing approaches contains only Top-K (e.g., $K = 10$) chosen keywords.

Since the existing approaches cannot outperform the baseline, we conduct further investigation to gain more insights. In particular, we divide our bug report collection into four different subsets based on the quality of their baseline queries and the presence of localization hints (e.g., stack traces) in the report texts (check Section 4.5 for details). Table 6 shows the performance of existing approaches for these four different subsets. We see that some bug reports could make good queries regardless of whether they contain any localization hints (e.g., HQ_{H+}) or not (e.g., HQ_{H-}). Baseline queries from these bug reports achieve 42%–50% Hit@1 and 100% Hit@10, which are highly promising and interesting. That is, in this case, the preprocessed version of a bug report (a.k.a., Baseline) is already a good search query for the bug localization. The existing approaches choose only Top-10 keywords from each bug report, and their performance is comparatively lower than the baseline. However, they also achieve ≈85% Hit@10 using their top ten search keywords, which is a high performance.

Table 6: Impact of Bug Report Quality and Localization Hints on Search Keyword Selection (Using Top-10 Results Only)

Technique	Genre	Hit@1	Hit@5	Hit@10	MRR	MAP
Bug reports with good baseline queries and no localization hints (**567**) (HQ$_{H-}$)						
Baseline	–	**41.96%**	**86.18%**	**100.00%**	**0.60**	**58.15%**
TF	Frequency	35.08%	68.87%	81.63%	0.49	48.61%
IDF		26.59%	53.17%	62.89%	0.38	35.99%
TF-IDF		33.84%	68.00%	80.67%	0.48	46.67%
Kevic and Fritz		35.22%	67.67%	79.82%	0.49	46.71%
STRICT	Graph	**36.14%**	**72.55%**	**84.83%**	**0.51**	**50.29%**
Bug reports with good baseline queries and with localization hints (**954**) (HQ$_{H+}$)						
Baseline	–	**50.01%**	**86.75%**	**100.00%**	**0.66**	**66.63%**
TF	Frequency	35.37%	66.61%	76.87%	0.49	49.53%
IDF		38.22%	63.12%	74.68%	0.49	50.05%
TF-IDF		**41.59%**	**70.80%**	**83.55%**	**0.54**	**55.15%**
Kevic and Fritz		33.19%	58.93%	69.14%	0.44	44.91%
STRICT	Graph	36.53%	69.86%	82.41%	0.51	51.85%
Bug reports with poor baseline queries and no localization hints (**372**) (LQ$_{H-}$)						
Baseline	–	**0.00%**	**0.00%**	**0.00%**	**0.00**	**0.00%**
TF	Frequency	1.22%	2.99%	7.97%	0.02	2.31%
IDF		1.38%	4.49%	6.94%	0.03	2.52%
TF-IDF		1.36%	6.00%	9.70%	0.03	3.18%
Kevic and Fritz		1.00%	6.24%	11.42%	0.03	3.23%
STRICT	Graph	**1.56%**	**9.54%**	**16.89%**	**0.05**	**5.12%**
Bug reports with poor baseline queries and with localization hints (**427**) (LQ$_{H+}$)						
Baseline	–	**0.00%**	**0.00%**	**0.00%**	**0.00**	**0.00%**
TF	Frequency	0.38%	2.86%	7.02%	0.02	1.64%
IDF		3.86%	15.11%	22.28%	0.09	8.74%
TF-IDF		1.56%	9.31%	16.80%	0.05	5.11%
Kevic and Fritz		1.12%	8.02%	12.93%	0.04	4.01%
STRICT	Graph	**4.91%**	**18.86%**	**25.77%**	**0.11**	**10.66%**

Emboldened=Baseline and state-of-the-art performance measures

Table 7: Statistics on Keyword Counts from the Search Queries

Query	Keyword Source	Min	Median	Mean	Max
Baseline	{*title*+*description*}	03	32	49	406
Baseline-I	{*title*}	01	07	07	24
Baseline-II	{*description*}	01	28	46	404
TF-IDF	{*title*+*description*}	03	10	10	10
STRICT	{*title*+*description*}	03	10	10	10
SCP	{*title*+*description*+ *source code*}	03	40	56	406
ACER	{*title*+*description*+ *source code*}	13	40	56	375

From Table 6, we also see another interesting finding. Our dataset contains a total of 799 (372 LQ$_{H-}$+427 LQ$_{H+}$) bug reports with their baseline queries that fail to retrieve any buggy source document within their Top-10 results. That is, they deliver 0.00% Hit@10 in the bug localization. It should be noted that 427 of these bug reports contain explicit localization hints (e.g., stack traces, program elements). Unfortunately, in this case, the baseline search queries were clearly not sufficient. On the contrary, the existing approaches achieve up to 17% Hit@10 and

26% Hit@10 for these bug reports with no localization hints (i.e., LQ_{H-}) and with localization hints (i.e., LQ_{H+}). Although 26% Hit@10 could still be considered as low, it is clearly a better alternative than the baseline for these 799 bug reports which constitute 34% of our dataset. We also note that graph-based approaches (e.g., STRICT) perform better than frequency-based approaches (e.g., TF-IDF) in delivering appropriate search keywords from these types of bug reports.

According to our analysis in Table 5, baseline approach outperforms each of the five existing techniques on keyword-selection (e.g., TF, STRICT). Further investigation in Table 6 demonstrates that the baseline method can only outperform the existing approaches when they are provided with the bug reports containing good baseline queries. Besides this corner case, the above findings clearly question the benefits of existing sophisticated techniques. However, it should be noted that our baseline queries (*title + description*) contain a total of 49 keywords on average (median: 32), whereas each query from the existing techniques contains at most 10 keywords (details in Table 7). Thus, the comparison between them was not head-to-head using same settings. We thus further wanted to see whether the suggested keywords are any better than 10 randomly selected keywords from the bug report texts (a.k.a., random baseline, Baseline_R), which could show the benefits of the existing keyword selection algorithms (e.g., TF, TF-IDF, STRICT). We thus repeat the process of random keyword selection five times, evaluate their performance in finding buggy documents, and then report their average performance. Fig. 2 demonstrates the comparison between random baseline (Baseline_R) and the best performing existing techniques for each of the four subsets of bug reports. Interestingly, we found that the suggested keywords (by existing techniques) are indeed better than the randomly chosen baseline keywords. They achieve higher Hit@K than that of the random baseline method in all circumstances. In particular, STRICT, the graph-based approach, performs consistently higher than not only the random baseline but also the competing frequency-based techniques. We conducted non-parametric statistical tests (e.g., *Wilcoxon Signed Rank*, *Cliff's delta*), and found that STRICT outperforms the random baseline (Baseline_R) with significant margin (*p-value*$\leq$0.01) and large effect size ($0.54 \leq \delta \leq 0.90$) with each of the four subsets of bug reports (Fig. 2-(a), (b), (c), (d)).

To determine the potential impact of *tangled commits* in our ground truth (e.g., $\approx$5%, details in Section 4.1), we conduct a limited experiment using 175 bug reports. We randomly choose these bug reports and ensure that their ground truths do not contain any tangled commits through a careful manual analysis. From Table 5, we see that the performance of baseline queries and the queries from the existing approaches reduce marginally with this limited dataset. However, the overall findings did not change. The queries from state-of-the-art approaches still cannot outperform the baseline queries in bug localization according to multiple performance metrics (e.g., Hit@K, MAP, MRR).

According to our investigation, the dataset might have contained bug reports from multiple versions of a software system, which could pose a threat to our reported findings. However, to mitigate this threat, we conducted additional experiments. In particular, we identified the project version at our disposal and selected all the bug reports from the dataset that targeted this version, which left us with 360 bug reports from six subject systems. The associated artifacts are included in the replication package. From our experiments in Tables 5, 8, we see that our major findings did not change. The performance of the search queries from each existing method increases to some extent, which aligns to an earlier finding in the similar scenario [38]. However, the queries from state-of-the-art approaches [56, 57] still cannot outperform the baseline search queries, which was one of our major findings from RQ1. Furthermore, as shown in Table 8, graph-based approaches perform consistently higher than frequency-based approaches in improving the baseline queries for IR-based bug localization. Thus, our major findings remain the same even with the bug reports from a single software release.

While the above analyses considered only Top-10 results, we further contrast between frequency-based and graph-based approaches by analysing all the results retrieved by their queries. In particular, we compare each suggested query with its baseline counterpart in terms of their Query Effectiveness (i.e., position of the first ground truth, defined in Section 4.4). If the suggested query achieves a higher rank than the baseline, existing literature [25, 48] defines it as *query improvement*, otherwise it is considered to be *query worsening*. If both queries achieve the same rank, then it is called *query preserving*. From Table 8, we see that TF-IDF performs the best among the four frequency-based approaches. It improves 21% and preserves 31% of the baseline queries. Unfortunately, TF-IDF based approach also worsens 49% of the queries. On the contrary, STRICT, a graph-based approach, improves 29% and worsens 44% of the queries which are 40% higher and 11% lower respectively. When this experiment is repeated using the 175 manually selected bug reports and 360 single-release bug reports above, our findings also remain aligned (Table 8). However, as shown above, none of these existing techniques improves more baseline queries than it worsens. We thus further investigate this issue using different subsets of bug reports. From Fig. 3, we see that STRICT can actually improve more baseline queries than it worsens when the technique is provided with the bug reports containing poor baseline queries (LQ_{H-}, LQ_{H+}). Unfortunately, it fails to do so when the bug reports contain good baseline queries (i.e., Query Effectiveness $\leq$10) in their texts.

5.3 Search Keyword Selection from Source Code Document

Our analyses show that (1) bug reports could make poor baseline queries and (2) existing techniques for selecting keywords from them might not be sufficient. Thus, several approaches [13, 56, 63, 64, 70] attempt to complement the baseline queries with appropriate keywords chosen from apparently relevant source code documents. That is, each search query from these approaches is a combination of baseline query (from the bug report) and suggested keywords from the source code. Tables 8, 9, 10, and Fig. 3 summarize our comparative analysis between frequency-based and graph-based keyword selection from source code documents. From Table 9, we see that all four frequency-based approaches – Rocchio [64], RSV [63], Dice [25] and SCP [70] – perform almost equally. They achieve a maximum of 66% Hit@10, 42% precision and a reciprocal rank of 0.42 which are marginally lower than the baseline measures. Thus, TF-IDF and its variants might not be enough to select keywords neither from the bug reports (e.g., Table 6) nor from the source code. On the other hand, ACER, a graph-based approach, achieves 68% Hit@10 with 43% precision and 0.43 reciprocal rank which are marginally higher than the baseline measures. We also further investigate the performances of baseline and existing techniques using four subsets of bug reports. From Table 10, we see

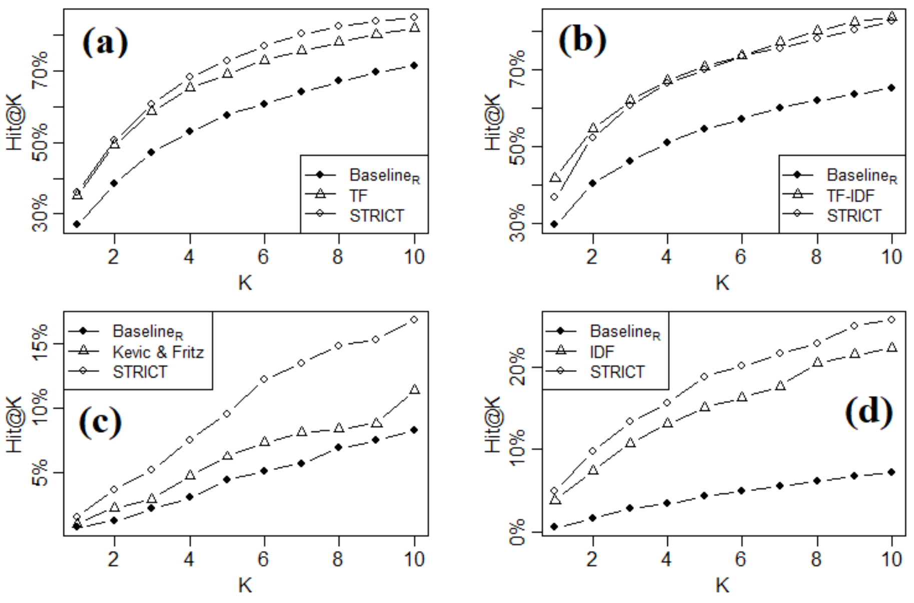

Fig. 2: Comparison between existing techniques and random baseline (Baseline$_R$) for (a) bug reports with good baseline queries and no localization hints (HQ$_{H-}$), (b) bug reports with good baseline queries and with localization hints (HQ$_{H+}$), (c) bug reports with poor baseline queries and no localization hints (LQ$_{H-}$), and (d) bug reports with poor baseline queries and with localization hints (LQ$_{H+}$)

that the keywords collected from the source code improve the baseline queries marginally when they are already good (e.g., HQ$_{H-}$, HQ$_{H+}$). However, these keywords from source code can significantly improve the baseline queries when they are poor (e.g., LQ$_{H-}$, LQ$_{H+}$). In particular, the best performing approach– ACER– achieves 9%–12% Hit@10 for the bug reports with poor baseline queries, which is 0.00% for the baseline method. It should be noted that these accuracy measures (Tables 9, 10) are based on Top-10 results only. We also note that the graph-based approaches are more effective than their frequency-based counterparts in delivering appropriate search keywords from the source code documents.

We also compare between graph-based and frequency-based approaches based on how much they can improve the baseline queries through query expansion. From Table 8, we see that SCP [70] was found more effective than other frequency-based approaches in delivering appropriate keywords from the source code. It improves 27% and preserves 38% of the baseline queries. However, this approach also worsens 35% of the queries. On the contrary, ACER improves 35% and worsens 24% of the baseline queries which are 29% higher and 31% lower respectively. That is, the graph-based approach achieves $\approx$ 10% *net improvement* (i.e., improvement − worsening) in the baseline queries. When this experiment is repeated using 175 manually selected and 360 single-release bug reports, our findings also remain aligned. We also further investigate using four different bug report groups. From Fig. 3, we see that ACER can actually improve more baseline queries than it worsens. It also should be noted that while the net improvement is marginal for the

Table 8: Improvement & Worsening of Baseline Queries by Existing Techniques

Query	**Genre**	**Improvement**	**Worsening**	**Preserving**
Query Construction using Bug Reports Only (2,320)				
TF	Frequency	434 (18.71%)	1,204 (51.90%)	682 (29.40%)
IDF		443 (19.10%)	1,281 (55.22%)	596 (25.69%)
TF-IDF		475 (20.47%)	1,135 (48.92%)	710 (30.60%)
Kevic and Fritz		496 (21.38%)	1,278 (55.09%)	546 (23.53%)
STRICT	Graph	**664 (28.62%)**	**1,008 (43.45%)**	**648 (27.93%)**
Query Construction using Bug Reports (2,320) + Source Code Documents				
Rocchio	Frequency	639 (27.54%)	887 (38.23%)	794 (34.22%)
RSV		650 (28.02%)	869 (37.46%)	801 (34.53%)
Dice		575 (24.78%)	827 (35.65%)	918 (39.57%)
SCP		626 (26.98%)	814 (35.09%)	880 (37.93%)
ACER	Graph	**805 (34.70%)**	**563 (24.27%)**	**952 (41.03%)**
Query construction using manually selected bug reports (175)				
TF	Frequency	34 (19.43%)	81 (46.29%)	60 (34.29%)
IDF		27 (15.43%)	113 (64.57%)	35 (20.00%)
TF-IDF		36 (20.57%)	92 (52.57%)	47 (26.86%)
Kevic and Fritz		36 (20.57%)	93 (53.14%)	46 (26.29%)
STRICT	Graph	**47 (26.86%)**	**75 (42.86%)**	**53 (30.29%)**
ACER		**57 (32.57%)**	**55 (31.43%)**	**63 (36.00%)**
Query construction using single-release bug reports (360)				
TF	Frequency	65 (18.06%)	199 (55.28%)	96 (26.67%)
IDF		75 (20.83%)	203 (56.39%)	82 (22.78%)
TF-IDF		81 (22.50%)	181 (50.28%)	98 (27.22%)
Kevic and Fritz		76 (21.11%)	213 (59.17%)	71 (19.72%)
STRICT	Graph	**106 (29.44%)**	**168 (46.67%)**	**86 (23.89%)**
ACER		**133 (36.94%)**	**87 (24.17%)**	**140 (38.89%)**

Emboldened=State-of-the-art performance measures

Table 9: Frequency-Based vs. Graph-Based Keyword Selection from Source Code Document (Using Top-10 Results Only)

Technique	**Genre**	**Hit@1**	**Hit@5**	**Hit@10**	**MRR**	**MAP**
Baseline	–	31.98%	57.96%	66.50%	0.43	42.69%
Baseline-I		22.37%	46.73%	57.58%	0.33	32.82%
Baseline-II		29.01%	51.16%	60.43%	0.38	38.21%
Rocchio	Frequency	30.17%	55.19%	65.03%	0.41	40.76%
RSV		30.37%	55.75%	65.00%	0.41	41.19%
Dice		31.15%	56.99%	66.10%	0.42	41.86%
SCP		31.34%	56.03%	66.20%	0.42	41.95%
ACER	Graph	**32.21%**	**58.88%**	**67.45%**	**0.43**	**43.34%**

Emboldened=State-of-the-art performance measures

bug reports with good baseline queries, it is significantly large (e.g., 26%) for the bug reports with poor baseline queries. All these empirical findings above suggest that graph-based approaches are a better choice for search keyword selection from the source code documents in the context of IR-based bug localization.

Summary of RQ$_1$: Bug reports could lead to poor search queries despite having explicit bug localization hints (e.g., program elements, stack traces) and to good search queries despite lacking such hints, which challenges the conventional wisdom. Queries from existing approaches achieve a maximum of 68% Hit@10,

Table 10: Impact of Bug Report Quality and Localization Hints on Query Expansion (Using Top-10 Results Only)

Technique	**Genre**	**Hit@1**	**Hit@5**	**Hit@10**	**MRR**	**MAP**
Bug reports with good baseline queries and no localization hints (**567**) (HQ_{H-})						
Baseline	–	**41.96%**	**86.18%**	**100.00%**	**0.60**	**58.15%**
Rocchio	Frequency	38.92%	79.32%	92.20%	0.56	54.45%
RSV		38.24%	80.81%	92.56%	0.56	55.29%
Dice		**43.32%**	84.13%	**95.74%**	0.60	58.17%
SCP		41.36%	81.09%	92.68%	0.58	56.46%
ACER	Graph	42.55%	**85.62%**	94.87%	**0.61**	**58.90%**
Bug reports with good baseline queries and with localization hints (**954**) (HQ_{H+})						
Baseline	–	**50.01%**	**86.75%**	**100.00%**	**0.66**	**66.63%**
Rocchio	Frequency	47.92%	83.69%	94.46%	0.63	63.71%
RSV		48.61%	84.04%	94.58%	0.63	64.40%
Dice		47.88%	85.47%	96.08%	0.63	64.46%
SCP		48.02%	83.71%	95.80%	0.63	64.22%
ACER	Graph	**50.20%**	**88.95%**	**97.43%**	**0.66**	**67.06%**
Bug reports with poor baseline queries and no localization hints (**372**) (LQ_{H-})						
Baseline	–	**0.00%**	**0.00%**	**0.00%**	**0.00**	**0.00%**
Rocchio		0.00%	2.68%	9.99%	0.02	1.51%
RSV		0.00%	2.20%	10.71%	0.02	1.62%
Dice		0.00%	0.00%	4.77%	0.01	1.00%
SCP	Graph	0.00%	1.92%	11.39%	2.05%	2.09%
ACER		0.00%	2.36%	**11.78%**	1.76%	1.81%
Bug reports with poor baseline queries and with localization hints (**427**) (LQ_{H+})						
Baseline	–	**0.00%**	**0.00%**	**0.00%**	**0.00**	**0.00%**
Rocchio		0.00%	1.00%	5.44%	0.01	1.00%
RSV		0.00%	1.00%	4.91%	0.01	1.00%
Dice		0.00%	1.01%	6.65%	0.01	1.00%
SCP		0.00%	1.82%	5.83%	0.01	1.09%
ACER	Graph	0.00%	1.00%	**8.78%**	0.01	1.15%

Emboldened=Baseline and state-of-the-art performance measures

43% MAP and 0.43 MRR in bug localization, which are comparable to the baseline measures. However, neither existing techniques nor baseline technique might be able to deliver appropriate search queries from a significant number of bug reports (e.g., 34%), which warrants further investigation. According to our experiments, graph-based approaches consistently perform higher than the best performing frequency-based approaches in suggesting appropriate keywords both from bug reports and from source code documents.

6 Answering $\mathbf{RQ_2}$: Comparing Optimal, Near-Optimal Search Queries with Baseline and State-of-the-Art Queries in IR-Based Bug Localization

6.1 Genetic Algorithm (GA)-Based Keyword Selection

According to $\mathbf{RQ_1}$, existing state-of-the-art approaches might not be enough for delivering appropriate search queries from the bug reports that lead to poor baseline queries (LQ_{H+}, LQ_{H-}). Their queries achieve a maximum of 26% Hit@10 that

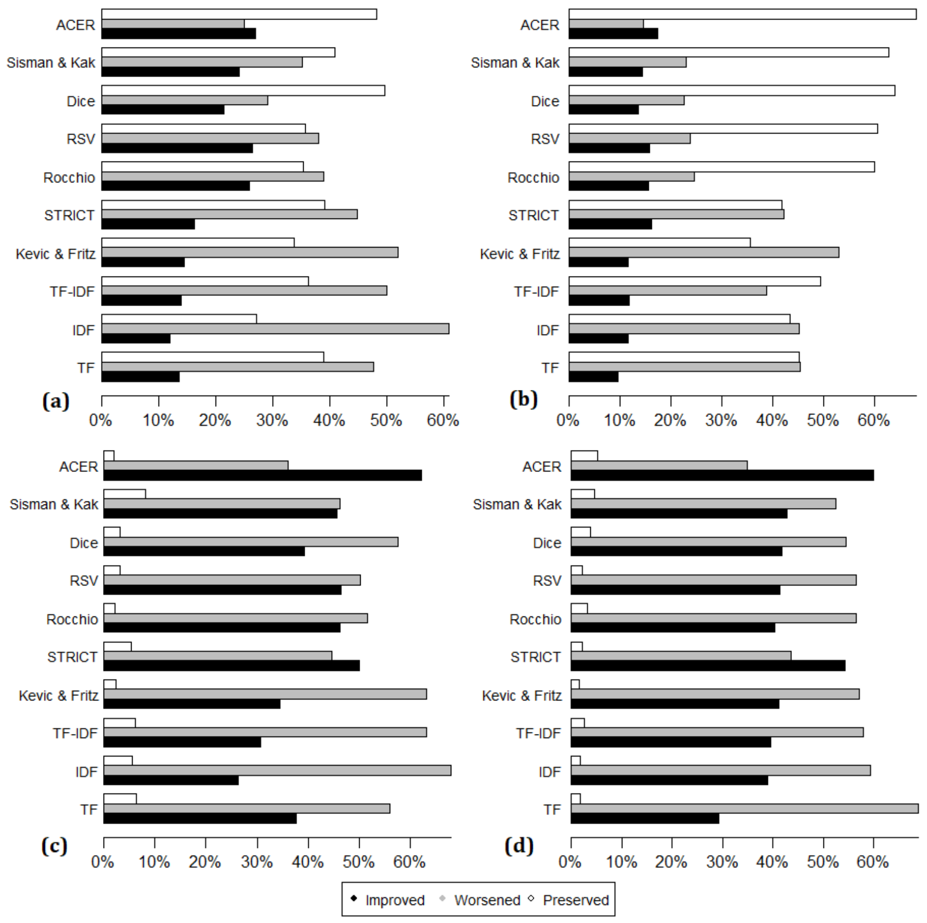

Fig. 3: Improvement and worsening of baseline queries by existing techniques for (a) bug reports with good baseline queries and no localization hints, (b) bug reports with good baseline queries and with localization hints, (c) bug reports with poor baseline queries and no localization hints, and (d) bug reports with poor baseline queries and with localization hints

outperforms baseline Hit@10 (0.00%) when only Top-10 results are analyzed. However, this still indicates a low performance. We thus wanted to determine whether the these bug reports actually contain any meaningful keywords that could form any optimal or near-optimal search queries. We define an *optimal search query* as a set of keywords that can retrieve the first faulty source document at the top-most position of the result list (Table 12). On the other hand, *near-optimal queries* return their first faulty document at the 2^{nd} to 10^{th} position of their result list.

We employ a Genetic Algorithm (GA)-based approach for identifying optimal and near-optimal search keywords from each bug report, as done by an earlier study [46]. Algorithms 1 and 2 show the pseudo code of our GA-based approach, **NrOptimal$_{\mathbf{GA}}$**. It should be noted that this **NrOptimal$_{GA}$ might not be applied in a real-world bug localization scenario where the ground truth is not**

known [46]. Although NrOptimal$_{GA}$ could not be used in practice, it could help gather useful insights (e.g., characteristics of optimal keywords) to improve other existing techniques (e.g., Tables 21, 22). We thus attempt to construct an optimal query from each bug report and if not possible, a near-optimal query from each bug report using five different steps inspired by biological evolution as follows.

(a) Initial Population Generation: The first step of a Genetic Algorithm is the initialization of solution candidates (a.k.a., *population*) where each candidate (or chromosome) is characterized with a fixed set of genes (Lines 4–6, Algorithm 1). In the context of our query construction problem, we initialize a population *pop* of M candidate queries. Each candidate comprises of K terms (or genes) randomly taken from all terms (or keywords) R of a bug report. It should be noted that we choose K = 5, 10, 15, 20 terms as the initial length of each solution candidate. Population from each iteration of the algorithm is also called a *generation*.

(b) Fitness Evaluation: Since Genetic Algorithms attempt to find out an optimal (or near-optimal) solution to a problem, each solution candidate from each generation is evaluated with a *fitness function*. In this study, we design our GA-based approach with Query Effectiveness (Section 4.4) as the fitness function. That is, NrOptimal$_{GA}$ attempts to find out a candidate query that retrieves the ground truth as its topmost search result (i.e., QE=1) (Lines 8–9, 17, Algorithm 1). Although the Query Effectiveness (QE) has been primarily used in our algorithm, other performance metrics (e.g., MAP) could be easily employed as the fitness function. Fitness function is frequently invoked during the selection step.

(c) Selection: The third step of a Genetic Algorithm is the population selection. The idea behind natural selection is to select the fittest individuals and let them pass their genes to the next generation. In the context our query construction problem, we select two fittest individuals (i.e., high performing candidate queries) from a population (generation) as parents and send them forward for further evolution. In particular, we arrange two tournaments by randomly selecting two subsets (N candidates each) of a population *pop*, and identify the two fittest candidates as parents (Lines 9–12, Algorithm 2).

(d) Crossover: Chromosomal *crossover* involves reconstruction of a new chromosome from two given chromosomes through gene overlapping. In our problem context, we choose two fittest candidate queries from the selection step above and generate a new candidate by randomly swapping their keywords (or genes). We adopt a *uniform crossover* strategy for generating the new chromosome. That is, for each gene index within the chromosome, the new gene (keyword) is taken from either parent chromosome (i.e., candidate query) with an equal probability (i.e., 0.5). We repeat the process $M-1$ times and generate a new population *newPop* using crossover and elitism strategy (Lines 9–15, Algorithm 2). Elitism ensures that the fittest candidate ($fittest$) from a previous generation joins the new generation without any crossover operation [21] (Lines 5–8, Algorithm 2).

(e) Mutation: Once a new population is created, *mutation* modifies each candidate (chromosome) by randomly *flipping* genes in random positions within a chromosome [21]. Similarly, we randomly *replace* the keywords with other keywords within each candidate query and mutate all candidates of a population (Lines 16–21, Algorithm 2). In particular, we use a non-uniform, single point mutation strategy with a low mutation rate where each point of mutation is chosen with a small probability of 0.015. We implement a restrictive mutation policy to preserve

Algorithm 1 Query Construction from a Bug Report using Genetic Algorithm

```
procedure NrOptimal_GA(R)
    ▷ R: all terms from a bug report
    ▷ Q: near optimal search query
    ▷ population selection
    pop ←generatePopulation(R, M, K)
    generation++
    while pop.getFittest().fitness<MAX_FITNESS do
        ▷ fitness evaluation
        Individual fittest ← pop.getFittest()
        ▷ crossover and mutation
        pop ←evolvePopulation(fittest, pop)
        if generation==MAX_GENERATION then
            break
        end if
    end while
    ▷ suggesting the near-optimal query
    Q ← pop.getFittest().genes
    return Q
end procedure
```

Table 11: A Working Example of Query Construction with NrOptimal$_{GA}$

Step	**Candidate Queries**	**QE**
All keywords	{bug, change, dialogs, primitive, object, accessible, command, variables, expressions, view, contexts}	105
Initialize population	C$_1$: {variables, command, variables, variables, contexts}	∞
	C$_2$: {variables, dialogs, change, object, object}	167
	C$_3$: {expressions, change, command, command, primitive}	233
	C$_4$: {dialogs, accessible, accessible, expressions, change}	121
	C$_5$: {dialogs, contexts, object, primitive, expressions}	13
	C$_6$: {contexts, view, view, dialogs, command}	07
	C$_7$: {dialogs, dialogs, contexts, Bug, accessible}	26
	....	-
	....	-
	C$_M$: {contexts, command, object, accessible, contexts}	619
Selection	P$_1$: {contexts, view, view, dialogs, command}	07
	P$_2$: {dialogs, contexts, object, primitive, expressions}	13
Crossover	Child: {dialogs, view, view, primitive, command}	**05**
Mutation	MC: {view, primitive, command, dialogs}	**04**
After Three Generation of Evolutions		
Duplicate keywords removal		
Solution	**{primitive, dialogs}**	**01**

QE = Query Effectiveness, **C**$_i$ = Chromosome, **P**$_i$ = Parent chromosome, **MC** = Mutated chromosome

the diversity of the population that was created in the crossover step above. Once completed, we finalize the new population for the next round of evolution.

Four genetic operations above – selection, fitness calculation, crossover, mutation – are performed iteratively on each generation until a candidate query with the *maximum fitness* (e.g., QE = 1, ground truth as the topmost result) is found

Algorithm 2 Evolve the Population of Candidate Queries

```
1:  procedure evolvePopulation($pop$,$fittest$)
2:      ▷ $pop$: current population of candidate queries
3:      ▷ $fittest$: fittest individual
4:      $newPop \leftarrow \{\}$
5:      ▷ retaining the fittest individual
6:      if $elitismEnabled$ then
7:          $newPop[0] \leftarrow fittest$
8:      end if
9:      ▷ evolve through selection and crossover operations
10:     for $i \in pop.\text{index}-1$ do
11:         $indiv1 \leftarrow$tournamentSelection($pop$)
12:         $indiv2 \leftarrow$tournamentSelection($pop$)
13:         $newIndiv \leftarrow$crossover($indiv1$, $indiv2$)
14:         $newPop[i] \leftarrow newIndiv$
15:     end for
16:     ▷ evolve through mutation operation
17:     for $i \in newPop.\text{index}$ do
18:         $indiv \leftarrow newPop[i]$
19:         $muIndiv \leftarrow$mutate($indiv$)
20:         $newPop[i] \leftarrow muIndiv$
21:     end for
22:     ▷ returning the evolved population
23:     return $newPop$
24: end procedure
```

or the generation count reaches the maximum threshold (e.g., 100). The crossover and mutation operations above might sometimes lead to *duplicate* keywords (genes) in a candidate query, which are discarded as they do not improve a candidate's fitness according to our investigation. We initialize each of the candidate queries (chromosome) with K unique keywords where $5 \leq K \leq 20$. Further justification on this can be found in Table 15. Since Genetic Algorithms involve randomness, we execute our approach *three* times on our dataset and report the performance for a combined set of optimal and near-optimal search queries.

6.2 An Example Workflow of NrOptimal$_{GA}$ Algorithm

Table 11 shows a working example of query construction from a given bug report (e.g., Bug #189012, `eclipse.jdt.debug`). We first collect all unique keywords from the *title* and *description* of the report and initialize a population of M solution candidates (or chromosomes). As shown in the Table 11, each candidate consists of $K = 5$ search keywords that determine the candidate's fitness to be a solution. We select the two fittest candidates (e.g., P_1 and P_2) from this population and generate a new candidate query from them using crossover operation (i.e., keyword overlapping). The operation also led to duplicate keywords, which were discarded. It should be noted that the crossover step improves the candidate's fitness. We repeat crossover operation M times and generate a new population of solution

Table 12: Query Definitions

Query	**Description**
Baseline query	The pre-processed version of the title and description texts from a bug report.
Optimal query	The search query that returns the first faulty source document at the *topmost* position of its result list.
Near-optimal query	The search query that returns the first faulty source document at the 2^{nd} to 10^{th} position of its result list.
Non-optimal query	The search query that returns the first faulty source document below the 10^{th} position of its result list.

Table 13: Baseline vs. Near-Optimal Search Queries from Bug Reports (Using Top-10 Results Only)

Query	**Hit@1**	**Hit@5**	**Hit@10**	**MRR**	**MAP**
Performance with all bug reports (**2,320**)					
Baseline	**31.98%**	**57.96%**	**66.50%**	**0.43**	**42.69%**
Baseline-I	22.37%	46.73%	57.58%	0.33	32.82%
Baseline-II	29.01%	51.16%	60.43%	0.38	38.21%
NrOptimal$_{\mathbf{GA}}$	**87.41%%**	**93.94%**	**95.74%**	**0.90**	**90.00%**
Bug reports with good baseline queries and no localization hints (**567**) (HQ$_{H-}$)					
Baseline	41.96%	86.18%	100.00%	0.60	58.15%
Baseline-I	32.26%	64.94%	79.72%	0.46	45.57%
Baseline-II	35.10%	69.96%	81.96%	0.50	47.65%
NrOptimal$_{\mathbf{GA}}$	**95.93%**	**100.00%**	**100.00%**	**0.98**	**93.51%**
Bug reports with good baseline queries and with localization hints (**954**)(HQ$_{H+}$)					
Baseline	50.01%	86.75%	100.00%	0.66	66.63%
Baseline-I	29.97%	61.20%	71.16%	0.43	43.79%
Baseline-II	46.11%	79.18%	92.07%	0.60	60.90%
NrOptimal$_{\mathbf{GA}}$	**98.04%**	**99.91%**	**100.00%**	**0.99**	**100.00%**
Bug reports with poor baseline queries and no localization hints (**372**)(LQ$_{H-}$)					
Baseline	0.00%	0.00%	0.00%	0.00	0.00%
Baseline-I	2.44%	8.40%	16.17%	0.05	5.05%
Baseline-II	0.00%	1.00%	5.55%	0.01	1.00%
NrOptimal$_{\mathbf{GA}}$	**50.04%**	**69.68%**	**77.96%**	**0.58**	**56.47%**
Bug reports with poor baseline queries and with localization hints (**427**)(LQ$_{H+}$)					
Baseline	0.00%	0.00%	0.00%	0.00	0.00%
Baseline-I	5.01%	18.99%	28.20%	0.11	10.98%
Baseline-II	0.00%	0.00%	0.02%	0.00	0.00%
NrOptimal$_{\mathbf{GA}}$	**80.70%**	**91.00%**	**93.37%**	**0.85**	**86.19%**

Emboldened=Baseline and NrOptimal$_{GA}$ performance measures

candidates. Then each of these candidates goes through a mutation step that helps them further improve their fitness. After three generation of evolution (selection, crossover and mutation), our algorithm, NrOptimal$_{GA}$ delivers the an optimal search query –{*primitive, dialogs*}–that achieves the best possible outcome in the IR-based bug localization (i.e., **QE=1**).

6.3 Comparison between Optimal or Near-Optimal Query and Baseline Query

Once optimal or near-optimal search queries are constructed from each of the bug reports, we compare them with baseline queries to determine their potential in IR-based bug localization. Tables 13, 14 and Figures 4, 5 summarize our comparative analyses. From Table 13, we see that the baseline queries (defined in Section 4.2) achieve a maximum of 67% Hit@10 with 43% mean average precision and a mean reciprocal rank of 0.43. On the contrary, the near-optimal search queries (NrOptimal$_{GA}$) achieve 96% Hit@10 with 90% MAP and 0.90 MRR, which are 44%, 111% and 109% higher respectively. These queries also achieve 94% Hit@5 which is 62% higher than the baseline counterpart. Furthermore, 87% of the bug reports contain optimal queries that can deliver the ground truth results at the topmost position of their result list. Thus, all the above findings clearly suggest that appropriate search keywords indeed exist in the bug report texts and they are highly effective for localizing bugs using Information Retrieval.

Although the above findings are promising, we further investigate whether they hold for all four subsets of bug reports. From Table 13, we see that search queries from NrOptimal$_{GA}$ achieve 98% Hit@1 and 100% Hit@5 when bug reports leading to good baseline queries (HQ$_{H-}$, HQ$_{H+}$) are considered. That is, the GA-based approach can always deliver optimal or near-optimal queries when the baseline queries are already good, which is expected and explainable. However, we also notice an interesting phenomenon for the bug reports leading to poor baseline queries (LQ$_{H-}$, LQ$_{H+}$). Baseline queries from these bug reports fail to retrieve any ground truth documents within their Top-10 results (i.e., 0.00% Hit@10). On the contrary, near-optimal queries constructed (by NrOptimal$_{GA}$) from these bug reports achieve 78%–94% Hit@10, which is both surprising and interesting. More interestingly, these queries can return the buggy source documents at the topmost position for 50%–81% of cases, which is highly promising. Thus, optimal and near-optimal query keywords are present even in the bug reports that make poor baseline queries (i.e., QE>10). Unfortunately, given our experimental findings (e.g., Tables 6, 13), the baseline and existing approaches from literature [31, 32, 57] are simply not effective enough to identify these keywords accurately.

We also contrast between near-optimal and baseline queries by analysing their top 1 to 10 search results. From Fig. 4, we see that near-optimal queries outperform the best performing baseline (i.e., Baseline) with a significant margin (i.e., *p-value*$\leq$0.01, 0.82$\leq\delta\leq$1.00) for each of four subsets of bug reports. Similar conclusions can also be drawn for their precision and reciprocal rank measures.

While the above analyses use Top-10 results only, we further contrast between near-optimal and baseline search queries using all the results retrieved by each query. In particular, we determine how many near-optimal queries improve upon the baseline alternatives. From Table 14, we see that near-optimal queries improve upon the baseline for 67%–77% of the queries and preserve 23%–33% of the queries, which are highly promising according to existing literature [25, 48]. That is, $\approx$72% of the baseline queries could have been improved either by choosing appropriate keywords from them or by discarding the noisy ones from them. Fig. 5 further demonstrates how near-optimal queries could improve upon $\approx$72% of the baseline queries from each of the six subject systems. It should be noted that NrOptimal$_{GA}$ does not worsen any of the baseline queries. We also investigate whether this finding holds for all four subsets of bug reports.

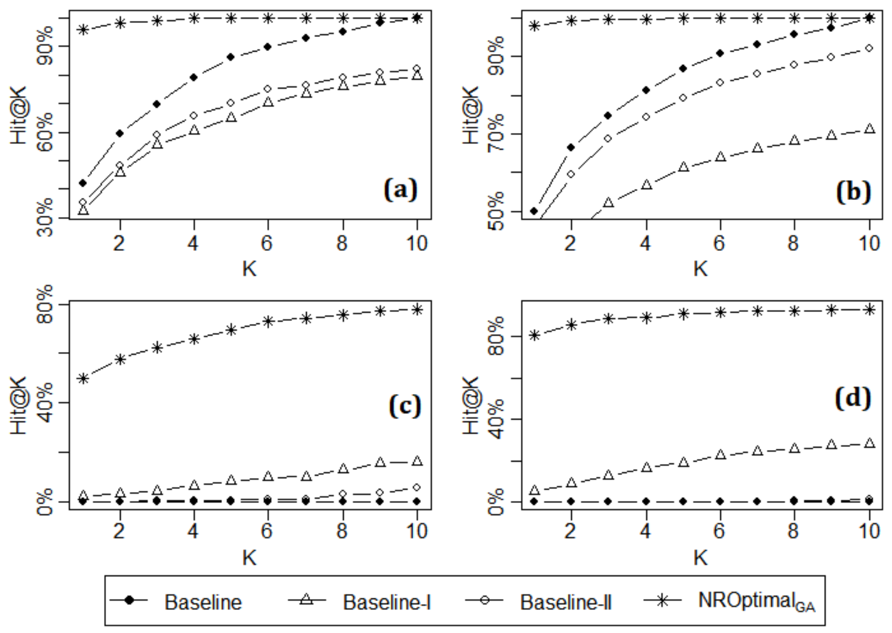

Fig. 4: Comparison of Hit@K between near-optimal and baseline queries from (a) bug reports with good baseline queries and no localization hints, (b) bug reports with good baseline queries and with localization hints, (c) bug reports with poor baseline queries and no localization hints, and (d) bug reports with poor baseline queries and with localization hints

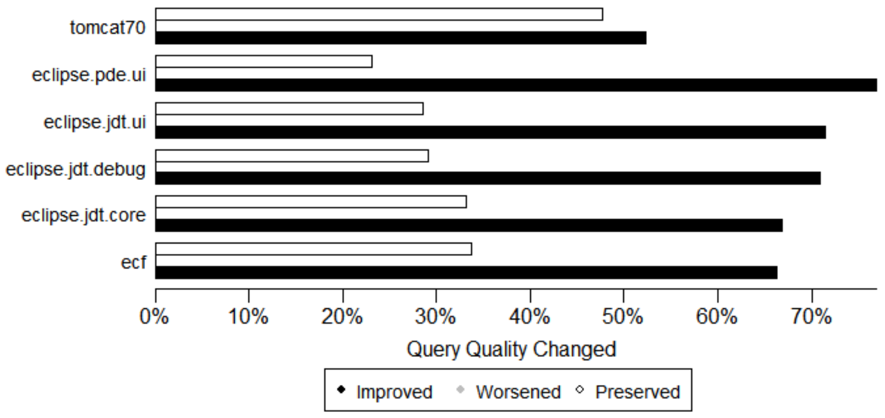

Fig. 5: Query improvement upon the Baseline alternatives by near-optimal search queries from the bug reports

From Table 14, we see that near-optimal queries are better than 46%–58% of 1,521 (567 + 954) baseline queries that were constructed from the bug reports leading to good baseline queries (HQ_{H-}, HQ_{H+}). On the contrary, NrOptimal$_{GA}$ can deliver better queries than 99% of 799 (372 + 427) baseline queries that were constructed from the bug reports leading to poor baseline queries (LQ_{H-}, LQ_{H+}).

Table 14: Query Improvement & Worsening against Baseline (Keywords from Bug Report)

Query Pair	**Improvement**	**Worsening**	**Preserving**
Performance for all Bug Reports (**2,320**)			
NrOptimal$_{GA}$ - Baseline-I	1,789 (77.11%)	0 (0.00%)	531 (22.89%)
NrOptimal$_{GA}$ - Baseline-II	1,643 (70.82%)	0 (0.00%)	677 (29.18%)
NrOptimal$_{GA}$ - Baseline	**1,563 (67.37%)**	0 (0.00%)	**757 (32.63%)**
Bug reports with good baseline queries and no localization hints (**567**)			
NrOptimal$_{GA}$ - Baseline-I	398 (70.19%)	0 (0.00%)	169 (29.81%)
NrOptimal$_{GA}$ - Baseline-II	373 (65.78%)	0 (0.00%)	194 (34.22%)
NrOptimal$_{GA}$ - Baseline	**331 (58.38%)**	0 (0.00%)	**236 (41.62%)**
Bug reports with good baseline queries and with localization hints (**954**)			
NrOptimal$_{GA}$ - Baseline-I	630 (66.04%)	0 (0.00%)	324 (33.96%)
NrOptimal$_{GA}$ - Baseline-II	478 (50.11%)	0 (0.00%)	476 (49.90%)
NrOptimal$_{GA}$ - Baseline	**439 (46.02%)**	0 (0.00%)	**515 (53.98%)**
Bug reports with poor baseline queries and no localization hints (**372**)			
NrOptimal$_{GA}$ - Baseline-I	361 (97.04%)	0 (0.00%)	11 (2.96%)
NrOptimal$_{GA}$ - Baseline-II	368 (98.93%)	0 (0.00%)	4 (1.08%)
NrOptimal$_{GA}$ - Baseline	**369 (99.19%)**	0 (0.00%)	**3 (0.01%)**
Bug reports with poor baseline queries and with localization hints (**427**)			
NrOptimal$_{GA}$ - Baseline-I	400 (93.68%)	0 (0.00%)	27 (6.32%)
NrOptimal$_{GA}$ - Baseline-II	424 (99.30%)	0 (0.00%)	3 (0.01%)
NrOptimal$_{GA}$ - Baseline	**424 (99.30%)**	0 (0.00%)	**3 (0.01%)**
Manually selected bug reports (**175**)			
NrOptimal$_{GA}$ - Baseline-I	137 (78.29%)	0 (0.00%)	38 (21.71%)
NrOptimal$_{GA}$ - Baseline-II	126 (72.00%)	0 (0.00%)	49 (28.00%)
NrOptimal$_{GA}$ - Baseline	**117 (66.86%)**	0 (0.00%)	**58 (33.14%)**
Single-release bug reports (**360**)			
NrOptimal$_{GA}$ - Baseline-I	283 (78.61%)	0 (0.00%)	77 (21.39%)
NrOptimal$_{GA}$ - Baseline-II	261 (72.50%)	0 (0.00%)	99 (27.50%)
NrOptimal$_{GA}$ - Baseline	**248 (68.89%)**	0 (0.00%)	**112 (31.11%)**

Emboldened=Improvement and preserving of baseline search queries by the NrOptimal$_{GA}$ approach

Table 15: Impact of Candidate Query Length and Fitness Function on Near-Optimal Queries (Using Top-10 Results Only)

Search Query		**Hit@1**	**Hit@5**	**Hit@10**	**MRR**	**MAP**
Impact of Candidate Search Query Length						
NrOptimal$_{GA}$	5-Keywords	85.93%	93.74%	95.65%	0.89	89.16%
NrOptimal$_{GA}$	10-Keywords	86.89%	93.73%	95.34%	0.90	89.70%
NrOptimal$_{GA}$	15-Keywords	87.26%	93.69%	95.34%	0.90	89.57%
NrOptimal$_{GA}$	20-Keywords	86.91%	93.65%	95.20%	0.90	89.43%
Impact of Fitness Function						
NrOptimal$_{GA}$	QE	86.89%	93.73%	95.34%	0.90	89.70%
NrOptimal$_{GA}$	MAP	86.83%	93.49%	95.12%	0.90	91.06%

We also repeat our experiments using 175 manually selected and 360 single-release bug reports (see details in Table 14), and our findings remain aligned with the above (e.g., ≈70% query improvement). All these empirical findings above clearly suggest that optimal and near-optimal queries exist in bug report texts and they are a far better alternative than the baselines.

Table 16: State-of-the-Art vs. Near-Optimal Search Queries from Bug Reports (Using Top-10 Results Only)

Technique	**Genre**	**Hit@1**	**Hit@5**	**Hit@10**	**MRR**	**MAP**
Bug reports with good baseline queries and no localization hints (**567**) (**HQ**$_{H-}$)						
Baseline	–	41.96%	86.18%	100.00%	0.60	58.15%
STRICT	Graph	36.14%	72.55%	84.83%	0.51	50.29%
ACER	Graph	42.55%	85.62%	94.87%	0.61	58.90%
NrOptimal$_{\mathbf{GA}}$	GA	95.93%	100.00%	100.00%	0.98	93.51%
Bug reports with good baseline queries and with localization hints (**954**) (**HQ**$_{H+}$)						
Baseline	–	50.01%	86.75%	100.00%	0.66	66.63%
TF-IDF	Frequency	41.59%	70.80%	83.55%	0.54	55.15%
ACER	Graph	50.20%	88.95%	97.43%	0.66	67.06%
NrOptimal$_{\mathbf{GA}}$	GA	98.04%	99.91%	100.00%	0.99	100.00%
Bug reports with poor baseline queries and no localization hints (**372**) (**LQ**$_{H-}$)						
Baseline	–	0.00%	0.00%	0.00%	0.00	0.00%
STRICT	Graph	1.56%	9.54%	16.89%	0.05	5.12%
NrOptimal$_{\mathbf{GA}}$	GA	50.04%	69.68%	77.96%	0.58	56.47%
Bug reports with poor baseline queries and with localization hints (**427**) (**LQ**$_{H+}$)						
Baseline	–	0.00%	0.00%	0.00%	0.00	0.00%
STRICT	Graph	4.91%	18.86%	25.77%	0.11	10.66%
NrOptimal$_{\mathbf{GA}}$	GA	80.70%	91.00%	93.37%	0.85	86.19%

We also investigate the impact of two important parameters –*candidate query length* and *fitness function* – upon the GA-based algorithm (NrOptimal$_{GA}$) used in our study. Table 15 summarizes our investigation details. In NrOptimal$_{GA}$, we initialize our solution candidates (chromosomes) with four different query lengths (e.g., 5, 10, 15, 20) and collect the optimal and near-optimal search queries generated from them. However, we did not find any significant difference in the performance of search queries generated from these four query lengths. According to our analysis, although the solution candidates started with many keywords (e.g., 20), the algorithm, NrOptimal$_{GA}$, eventually found out a subset of these keywords with the maximum fitness (e.g., QE = 1) through evolution and removal of duplicate keywords. However, construction of optimal and near-optimal queries from the long candidate queries is more time-consuming than with the short ones. We also conduct experiments using two different fitness functions – Query Effectiveness (QE) and Mean Average Precision (MAP) (defined in Section 4.4), and collect the optimal and near-optimal queries for them. We did not find any significant difference in the performance of search queries generated using these two fitness functions (Table 15). However, we found that MAP-based query construction is more time-consuming than QE-based one.

6.4 Comparison between Optimal or Near-Optimal Search Query and State-of-the-art Search Query

We compare optimal and near-optimal search queries with the queries from existing approaches [31, 56, 57]. From Table 16, we see that both GA-based approach and existing approaches were able to provide good search queries (e.g., 84%–97% Hit@10) from the bug reports that already make good baseline queries (i.e., HQ$_{H-}$ and HQ$_{H+}$). However, the best-performing existing approach (STRICT [57] according to Table 6) often struggles to provide appropriate queries from the bug reports that make poor baseline queries (i.e., LQ_{H-} and LQ_{H+}). It should be noted that bug reports could make poor baseline queries despite having explicit hints for bug localization (e.g., stack traces). From Table 16, we see that the queries from STRICT, the state-of-the-art technique, achieve 17%–26% Hit@10 for 799 (372 + 427) bug reports with poor baseline. On the contrary, search queries from the GA-based approach ($NrOptimal_{GA}$) for the same dataset achieve 78%–93% Hit@10, which are three to five times higher. Similar conclusions can also be drawn for other performance metrics (e.g., MAP, MRR). Furthermore, this approach was able to deliver optimal search queries from 50%–81% of the bug reports containing poor baseline queries (i.e., 50%–81% Hit@1).

According to our investigation, about 65.56% (567 $HQ_{H}-$ + 954 $HQ_{H}+$) bug reports inherently contain more useful keywords for localizing the bugs than others (372 $LQ_{H}-$ + 427 $LQ_{H}+$). We annotate these bug reports as the bug reports leading to good baseline queries and the bug reports leading to poor baseline queries. Besides, different existing approaches have their own strengths and weaknesses. Given these two dimensions, we see significant differences in techniques' performance across the bug report clusters (Table 6).

One can argue about the novelty of our findings in RQ_2 compared to that of Mills et al. [46]. However, we not only strengthen the earlier finding but also make further contributions in this work. First, we conduct a more extensive experiment using 2,320 bug reports (as opposed to 620 by Mills et al.) that are categorized into four different subsets and demonstrate that optimal search queries are present even in the bug reports that make poor baseline queries. Second, we show how effective the optimal and near-optimal search queries are compared to the baseline and state-of-the-art queries. Third, we demonstrate the impact of candidate query length and fitness function on the optimal or near-optimal search queries constructed by the genetic algorithm. Fourth, we also provide a detailed pseudo-code (Algorithms 1, 2) and an example workflow of the GA-based query construction, which were not provided by Mills et al. [46].

Summary of RQ_2: Bug reports could lead to poor baseline queries or could miss the explicit hints for localizing bugs (e.g., program elements, stack traces) in their texts. However, they generally **contain sufficient keywords** to construct optimal or near-optimal search queries, as demonstrated by the GA-based approach. These queries can deliver optimal performance in bug localization for **50%–81%** of bug reports. They also **outperform** the baseline and state-of-the-art search queries in all *four* performance measures (e.g., Hit@K, MAP, MRR, QE) by a **large margin**.

7 Answering RQ_3: Optimal vs. Non-Optimal Search Queries for IR-Based Bug Localization

According to RQ_1, state-of-the-art approaches [32, 57] might not be able to deliver appropriate search queries from many bug reports despite their texts containing explicit hints for bug localization (e.g., program elements) (Table 6). However, these bug reports actually contain optimal and near-optimal search queries in their texts as shown in RQ_2. Table 16 also demonstrates that near-optimal search queries from these bug reports perform at least *three* to *five* times higher than the state-of-the-art queries (STRICT [57]) in terms of Hit@10 and eight to ten times higher in terms of MAP and MRR. All these findings above beg an obvious question – *"What are the differences between optimal and non-optimal search queries from a bug report?"* A comprehensive understanding of these differences is essential since they can be leveraged to improve query reformulation approaches. We thus conduct a multi-modal analysis involving machine learning, data mining and manual inspection to answer this important question. Tables 16, 17, 18 and Figures 6, 7, 8 summarize our analysis details.

7.1 Optimal, Near-Optimal and Non-Optimal Search Queries

Since we wanted to differentiate among optimal, near-optimal and non-optimal search queries, we construct a dataset of 13,914 queries from 2,320 bug reports. Table 17 shows detailed dataset for our comparative analysis. We first collect queries from five existing approaches (TF, IDF, TF-IDF [31], Kevic and Fritz [32] and STRICT [57]) and the GA-based approach, NrOptimal$_{GA}$, and then carefully categorize them into optimal, near-optimal and non-optimal queries based on their effectiveness (Table 12). If a search query returns any ground truth result at the topmost position of its result list (i.e., QE=1), we consider it as an *optimal query.* If the query returns the ground truth between the 2nd and 10th position, we consider it as a *near-optimal query.* Developers often check only top 10 results before making another search with a reformulated query. Existing studies [25, 65, 85] often also analyze top 10 results for a query, which justifies our definition of near-optimal query. On the other hand, if the search query returns the ground truth result below the 10th position, it is considered as a *non-optimal query.* Thus, as shown in Table 17, we end up with 4,893 optimal, 3,857 near-optimal, and 5,164 non-optimal search queries in our dataset.

7.2 Search Query Characteristics

We select a total of 31 traditional metrics (Table 23) from five different dimensions to characterize each of the search queries. Our goal was to differentiate among optimal, near-optimal and non-optimal search queries using these metrics and then derive meaningful insights to improve the non-optimal queries. In particular, we choose the following five dimensions in our comparative analysis.

Frequency has been a popular proxy of keyword importance for decades [31]. It had been regularly used by the Information Retrieval community for the last 50 years. The underlying idea is that if a word frequently occurs within a document, it could serve as a potential query keyword for retrieving the document from a corpus. Since our optimal and non-optimal queries differ significantly in their performance (e.g., Query Effectiveness), it would be interesting to see whether they also differ significantly in terms of their frequency-based statistics. We thus use three frequency-based metrics (e.g., Term Frequency (TF), Inverse Document Frequency (IDF), Term Frequency $\times$ Inverse Document Frequency (TF-IDF)) in our comparative analysis. It should be noted that these metrics are often adopted by relevant literature [25, 45]. As shown in the Table 23, we capture four descriptive statistics (e.g., average, median, maximum, standard deviation) of each metric from each query where TF, IDF and TF-IDF are calculated for each query

Table 17: Optimal, Near-Optimal & Non-Optimal Search Queries

Techniques	**QS**	**Optimal**	**Near-Optimal**	**Non-Optimal**	**TQ**
TF, IDF, TF-IDF	Q-HQ$_{H-}$	1,415	1,313	674	3,402
Kevic and Fritz,	Q-HQ$_{H+}$	2,859	1,885	980	5,724
STRICT,	Q-LQ$_{H-}$	212	277	1,740	2,229
NrOptimal$_{GA}$	Q-LQ$_{H+}$	407	382	1,770	2,559
	Q-All	**4,893**	**3,857**	**5,164**	**13,914**

QS = Query set, **HQ**$_{H-}$ = Bug reports without hints leading to good baseline queries, **HQ**$_{H+}$ = Bug reports with hints leading to good baseline queries, **LQ**$_{H-}$ = Bug reports without hints leading to poor baseline queries, **LQ**$_{H+}$ = Bug reports with hints leading to poor baseline queries, **QE**=Rank of the first buggy document within the result list, **TQ** = Total queries.

keyword. In short, we collect 12 statistics to characterize each search query using their frequency-related properties.

Entropy has been a useful proxy for determining ambiguity or specificity of a given textual entity [12, 13]. It has been inspired by the concept of entropy from Information Theory domain [66]. That is, if a query contains keywords with high entropies (or ambiguities), the query could be hard to answer and thus might fail to retrieve any relevant documents. We were interested to see whether optimal queries are more specific (i.e., less ambiguous) than non-optimal queries. Such an insight could be helpful for selecting appropriate keywords from a textual entity (e.g., bug report). Thus, we use three entropy-based metrics from the existing literature – Term Entropy [45, 54], Jensen Shannon Divergence (JSD) [12], and Query Specificity Index (QSI) [24]– in our investigation. The entropy dimension provides six statistics for each query.

Mutual Information is another probabilistic metric that could be useful to differentiate between optimal and non-optimal search queries. It approximates the semantic dependency between two words based on their co-occurrences across multiple documents. That is, presence of keywords that share a lot of mutual information indicates the cohesiveness of a search query. We were interested to see whether the optimal queries are more *cohesive* than the non-optimal ones. We thus calculate Point-wise Mutual Information (PMI) for each of the keyword-pairs from a search query. Then we capture four descriptive statistics of this probabilistic metric and use them in our comparative analysis.

Part of Speech (POS) has been a popular and widely used heuristic for extracting keywords or phrases from a body of texts (e.g., bug report, Q&A threads) [57, 60, 77, 83]. In particular, nouns and verbs are preferred to other POS as keywords since they are intuitive and convey rich semantics. We were interested to see whether optimal and non-optimal queries differ from each other in the distribution of their noun or verb keywords. We thus consider four POS-based metrics (e.g., *nounRatio*, *verbRatio*, *nounVerbRatio* and *otherPOSRatio*) for our comparison.

Keyword Position could be another important dimension to compare optimal queries with non-optimal queries. Each bug report has two textual fields – *title* and *description*. We were interested to see how the keywords from optimal and non-optimal queries are distributed across these two fields. In particular, we wanted to see whether the optimal keywords are extracted from only title, only description or from both fields in a bug report. We thus use three position-related metrics (e.g., *titleKeywordRatio*, *bodyKeywordRatio*, *titleBodyKWRatio*) in our comparative analysis to differentiate between optimal and non-optimal queries.

Table 18: Performance of Query Classification Models

Model	#Queries	Precision	Recall	Precision	Recall	Accuracy
		Optimal		**Non-Optimal**		
RF-HQ	5,928	87.30%	89.40%	70.70%	66.40%	82.96%
RF-LQ	4,129	83.30%	81.40%	96.70%	97.10%	**94.77**%
RF-All	10,057	84.60%	79.80%	81.90%	86.20%	**83.12**%
		Near-Optimal		**Non-Optimal**		
RF-HQ	4,852	80.60%	68.40%	52.80%	68.30%	68.36%
RF-LQ	4,169	53.90%	43.60%	89.80%	93.00%	85.20%
RF-All	9,021	63.80%	70.70%	76.20%	70.00%	70.29%

RF = Random Forest, **RF-HQ** = Random Forest model based on bug reports leading to good baseline queries, **RF-LQ** = Random Forest model based on bug reports leading to poor baseline queries.

We also select two ad-hoc metrics (e.g., *groundTruthTermRatio*, *uniqueKeywords*) through a reverse engineering process. We wanted to see whether the keywords from optimal queries match with the ones collected from the ground truth file names. The idea is that if there exists a strong connection between these two keyword lists (i.e., high *groundTruthTermRatio*), one can focus on identifying the potential solution documents to make a better search query. We also consider the number of unique keywords in each query and attempt to understand the difference between optimal and non-optimal queries in this aspect.

Several earlier studies [25, 45, 56] employ dozens of pre-retrieval and post-retrieval query difficulty metrics to classify the *difficult* and *easy* queries. Although several of our adopted metrics overlap with theirs, we avoid several of them (e.g., Spatial Auto-Correlation, Query Scope). Our goals are to demonstrate how optimal queries are different from the non-optimal queries in terms of simple, intuitive, popular measures and also to derive meaningful insights in the process. Unfortunately, those metrics are computationally expensive, complex, and less intuitive, which makes them less than ideal for our comparative analysis.

7.3 Comparison between Optimal and Non-Optimal Search Queries using Machine Learning-Based Feature Importance Analysis

Optimal and non-optimal search queries might not be linearly separable across all characteristics discussed above. We thus attempt to separate them using a non-linear approach – machine learning algorithm. We use supervised machine learning to classify the search queries, identify such features that the classification algorithms found as *important*, and then analyze the distinctive characteristics of optimal and non-optimal queries as follows.

First, we make use of optimal (QE=1), near-optimal ($2 \leq QE \leq 10$), and non-optimal search queries (QE>10) (Table 17), calculate their metrics, and then train *six* classification models from them (Table 18). Since the queries were taken from various subsets of bug reports, the trained models were named accordingly. Given the imbalanced set of queries (Table 17), these models might suffer from over-fitting issues. We thus use RandomForest algorithm that was reported as robust against model over-fitting issues [45, 61], SMOTE-based oversampling [17] and 10-fold cross validation to mitigate the data-imbalance issue. Table 18 shows the performance of our trained models. We see that these models were able to classify the optimal and non-optimal queries with 83% to 95% accuracy, which is promising. While optimal and non-optimal queries differ significantly in terms of their retrieval performance (e.g., query effectiveness (QE)), this high classification accuracy of the models indicates that these queries can also be different in terms of their lexical or statistical characteristics (Section 7.2). That means, meaningful insights could be derived from these characteristics to improve poor or non-optimal search queries. However, as shown in the Table 18, near-optimal and non-optimal queries cannot be classified with a high accuracy. That is, they might not be very different in terms of their lexical or statistical characteristics.

Second, although non-linear, machine learning approach was able to separate optimal queries from non-optimal ones, the underlying decision trees could be difficult to interpret due to their sheer sizes. We thus rely on feature importance analysis and identify such features that the training algorithms found more important than others in separating the optimal queries from the non-optimal ones. Fig. 6 shows the relative importance of all features in terms of mean decrease accuracy, which indicates how much accuracy a model might lose if a certain feature is discarded from the model. We select the top 10 most important features including *groundTruthTermRatio*, *bodyKeywordRatio*, *stdTermEntropy*, *avgPMI* and *avgTF*, collect their corresponding numerical values, and then compare their distribution for optimal and non-optimal search queries. Then we shortlist four features for which the optimal and non-optimal queries demonstrate *significant differences* (i.e., *p-value*$\leq$0.05) in terms of non-parametric statistical tests (e.g., Mann-Whitney Wilcoxon, cliff's delta). In particular, we notice significant differences in the *distribution* of their term frequency, median term entropy, body keyword ratio, and noun keyword ratio (check Figures 7, 8, 9, 10 for details). It should be noted that *randomField*, a randomly generated feature, has been found as the least important feature for classification, which is expected and thus instills confidence in our feature importance estimation. Fig. 7 shows the probability distribution of the four features – average term frequency, median term entropy, percentage of keywords from body section, and percentage of noun keywords as follows.

From Fig. 7-(a), we see that the frequency of keywords in both optimal and non-optimal search queries has a lognormal distribution. Both queries have a high probability mass on a lower frequency (e.g., 1). However, non-optimal queries contain slightly more frequent keywords, especially the keywords that occur more than twice in a bug report. Box plots in Fig. 8-(a) also show a higher variance in keyword frequency for non-optimal search queries. We also repeated this experiment using bug reports that lead to poor baseline queries (LQ$_{H-}$, LQ$_{H+}$) and found that median frequency of optimal keywords is less than that of non-optimal keywords (Fig. 10-(a)). Traditionally, frequency has been considered as a popular proxy of keyword importance, which does not apply well to optimal search queries. One can argue about using TF-IDF as a proxy of keyword importance. However, according our investigation, non-optimal queries also contain keywords with significantly higher TF-IDF measures. From Fig. 7-(b), we also see that entropy measures of keywords from both queries have a mixed distribution. However, non-optimal search queries contain more keywords of high entropy. That is, optimal search queries contain less ambiguous keywords than non-optimal queries do, which aligns to conventional wisdom [12, 23]. Our statistical tests (e.g., Mann-Whitney Wilcoxon) also report a p-value of <0.001 with a *small* effect size (i.e., δ=0.30). Furthermore, this finding is confirmed by the box plots in Fig. 8-(b)

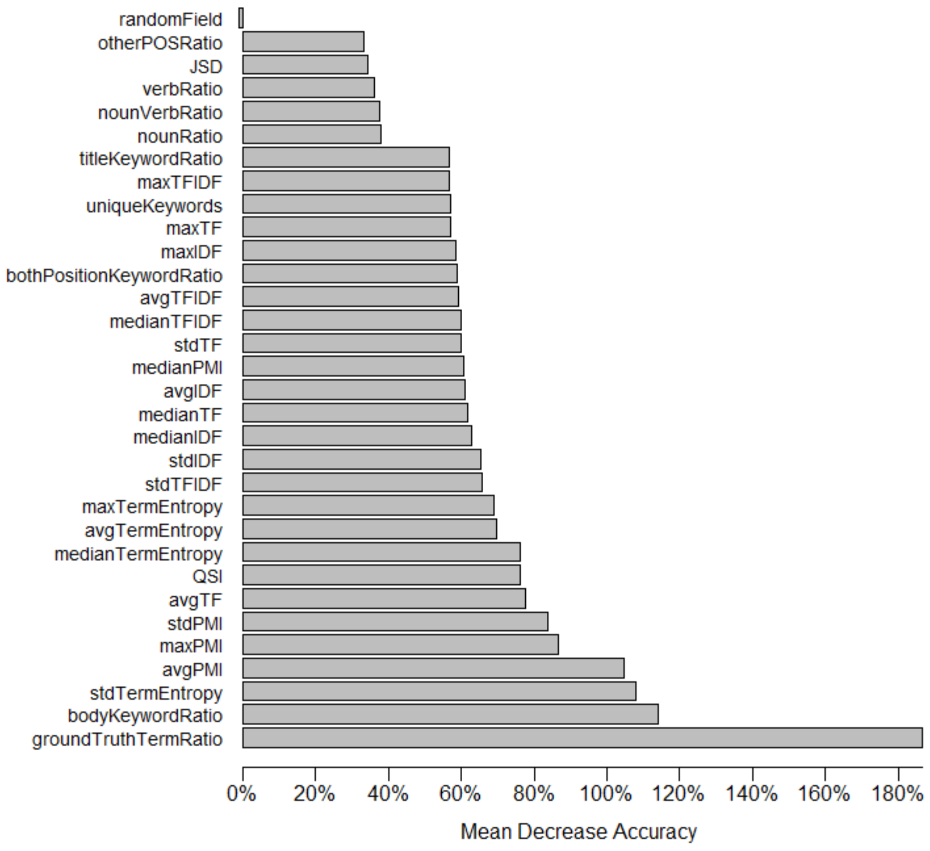

Fig. 6: Feature importance in the RandomForest model (RF-All)

where median entropy of optimal queries is lower (i.e., less ambiguous) than that of non-optimal queries. We also calculate the percentage of keywords in queries that were extracted from the *description* section (or body) of a bug report, and Fig. 7-(c) compares their distribution between two query types. We see that optimal queries are more likely to contain keywords from the description section than non-optimal ones. Although our statistical test reported a significant *p-value*, the effect size is negligible. Finally, Fig. 7-(d) shows the distribution of noun keywords in both optimal and non-optimal queries. While both distributions are irregular, we see that optimal queries are more likely to contain 100% noun keywords. This observation is further strengthened by the box plot analysis in Fig. 8-(d). We see that optimal queries are more likely to contain nouns as keywords. Our statistical tests also report a *p-value* of $\leq$0.001 with a small effect size (i.e., 0.20), which strengthens this finding. We also repeated each of these analysis above (e.g., Figures 7, 8) using optimal and non-optimal queries from the bug reports that lead to poor baseline queries (LQ_{H-}, LQ_{H+}). We were able to reproduce each finding as demonstrated in the Figures 9, 10, which further strengthens our findings on the four query features.

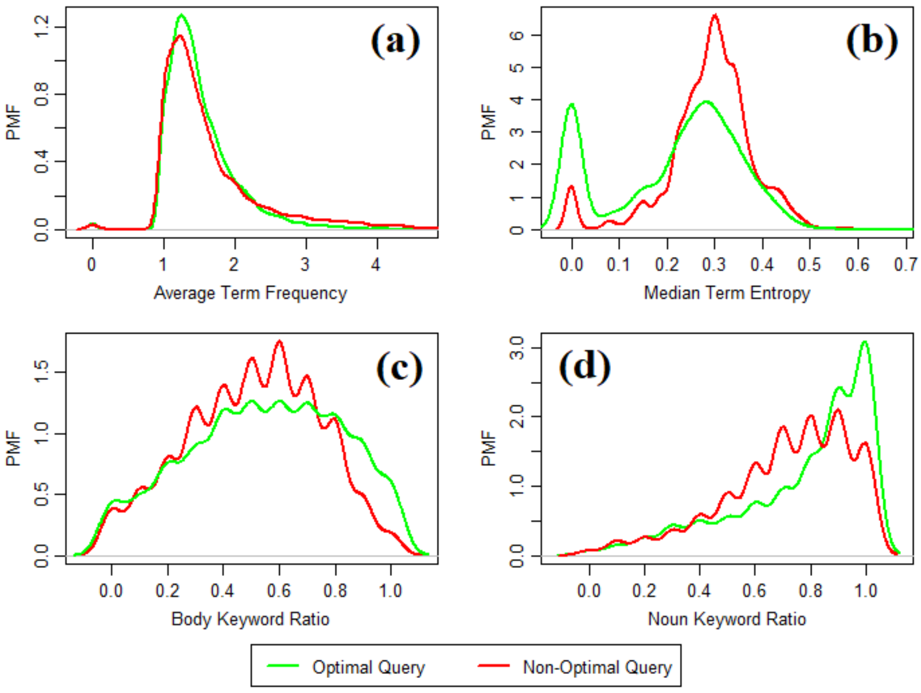

Fig. 7: Comparison between optimal and non-optimal keywords (from all bug reports) using their distributions (e.g., PMF=probability mass function) of (a) frequency, (b) entropy, (c) percentage from the body/description section of a bug report, and (d) percentage of nouns

We also perform regression analysis between the four query features discussed above and query performance. Table 19 summarizes our analysis using Logistic regression. We see that all four features are found statistically significant, which provides confidence in their coefficients. That is, for every unit increase in *nounRatio* feature, the odd of being a query to be *optimal* (i.e., QE=1) increases by 1.81. Similarly, every unit increase in *bodyKeywordRatio* increases the same odd ratio by 1.25. In short, nouns and keywords from body section are likely to improve the performance of a search query. On the other hand, every unit increase in *medianTF* and *medianTermEntropy* features changes the odd of being a query to be optimal by only 0.01 and 0.75 respectively. That is, frequent, ambiguous keywords are not likely to improve a search query. Thus, a Logistic regression-based analysis also leads us to the same conclusion regarding the four query features.

7.4 Actionable Insights

The above comparative analysis (Section 7.3, Figures 7, 8) has reported several interesting findings, which can be turned into actionable insights. Table 20 shows four actionable insights regarding frequency, ambiguity, position and part of speech of query keywords as follows.

Table 19: Regression Analysis between Query Features and Query Performance

Coefficient	**Estimate**	**OR**	**SE**	**z-value**	**p-value**
Intercept	1.13	3.10	0.10	-11.06	<2e-16***
medianTF	-0.29	0.75	0.03	8.68	<2e-16***
medianTermEntropy	-5.14	0.01	0.20	26.31	<2e-16***
bodyKeywordRatio	0.22	**1.25**	0.09	-2.44	0.02*
nounRatio	0.59	**1.81**	0.10	-6.07	1.28e-09***

OR=Odd Ratio, **SE**=Std. Error, *******=Strongly significant, *****=Significant
Emboldened=Odd ratios of strong features from the optimal search queries

Table 20: Actionable Insights from Feature Importance Analysis

No.	**Insight**
I_1	Optimal search keywords are less frequent within a bug report than the non-optimal ones.
I_2	Optimal search keywords are less ambiguous (i.e., have less entropy) than the non-optimal ones.
I_3	Optimal search keywords are more likely to be found in the *description* section of a bug report than the non-optimal ones.
I_4	Optimal search keywords are more likely to be *noun* than the non-optimal ones.

I_1: Frequency might not be a reliable proxy of keyword importance and highly frequent keywords should not be included in a query. This insight contradicts the conventional wisdom from existing literature [25, 32]. To implement this insight, we determine *median* frequency of keywords in a bug report and select the ones with less than median frequency for query expansion.

I_2: Term entropy is an important proxy of keyword importance and the keywords with less entropy, i.e., less ambiguous, should be included in a query. This insight is mostly aligned to earlier literature on query difficulty [12, 23, 45]. To implement this, we determine *median* entropy of keywords in a bug report and then select the ones with less than median term entropy for query expansion.

I_3: Keywords from the *description* section are more prevalent in optimal queries and thus they should be prioritized over others during query formulation. We determine the position of each keyword within a bug report and select the keywords that are only found in the description section for expanding a query.

I_4: Optimal queries contain more nouns than non-optimal queries and thus, noun keywords should be prioritized during query formulation. We determine the part of speech of each keyword in a bug report using Stanford POS tagger [71] and select the nouns for expanding a given query.

From Fig. 6, we also notice several ad hoc features (e.g., *uniqueKeywords*, *ground-TruthTermRatio*) that were found important for separating the optimal queries from the non-optimal ones. Although they are strong features for classification, they might not be actionable for keyword selection since they were derived through reverse-engineering. However, they could still help us better understand the optimal and non-optimal search queries.

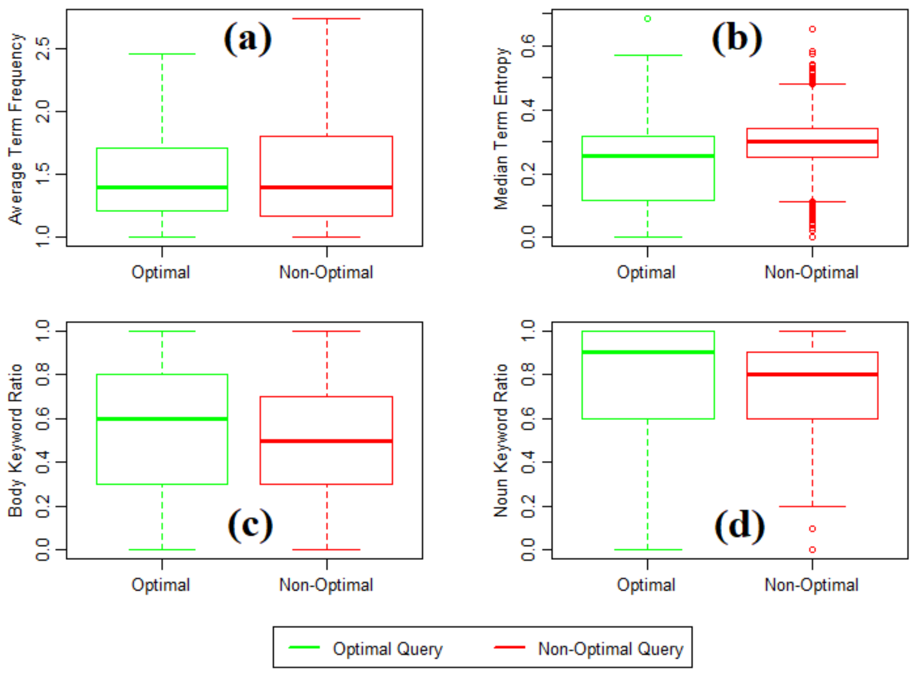

Fig. 8: Comparison between optimal and non-optimal keywords (from all bug reports) using their box plot of (a) frequency, (b) entropy, (c) percentage from the body/description section of a bug report, and (d) percentage of nouns

7.5 Improvement of Baseline Queries and State-of-the-art Search Queries with Actionable Insights

Our feature importance analysis above (Section 7.3) offers four actionable insights (Table 20). We apply these insights to both baseline and STRICT queries (i.e., state-of-the-art), expand them using appropriate keywords and determine their benefits. Tables 21, 22 summarize our experimental analysis as follows.

In Table 21, we investigate the benefits of four insights both in isolation and in combination by applying them to baseline queries. We see that expansion of baseline queries using less frequent keywords (i.e., I_1) improves their performance marginally. They get further improved when less ambiguous keywords are added to them (i.e., I_1+I_2). For example, they achieve a Hit@10 of 69% with 44% mean average precision which are 3% improvement over the baseline. However, a combination of three insights (I_1+I_2+I_4) leads to further improvement of baseline queries. That is, when less frequent, less ambiguous and noun keywords are added to 2,320 baseline queries from their corresponding bug reports, the extended queries achieve a 34% Hit@1, 69% Hit@10, 0.46 MRR and 45% MAP, which are 8%, 4%, 7% and 6% higher respectively, which are promising.

We also investigate the impact of our actionable insights upon the baseline queries from different subsets of bug reports. From Table 21, we see that they fail to improve the already good baseline queries, i.e., collected from HQ_{H-} and HQ_{H+}

Table 21: Improvement of Baseline Queries with Actionable Insights (Using Top-10 Results Only)

Technique	**Insight**	**Hit@1**	**Hit@5**	**Hit@10**	**MRR**	**MAP**
Performance with all bug reports (**2,320**)						
Baseline		**31.98%**	**57.96%**	**66.50%**	**0.43**	**42.69%**
Baseline	I_1	32.63%	58.33%	67.52%	0.44	43.76%
Baseline	I_2	28.82%	54.32%	64.00%	0.40	39.65%
Baseline	I_3	33.15%	56.99%	66.73%	0.43	43.31%
Baseline	I_4	28.95%	54.02%	63.43%	0.40	39.57%
Baseline	$\mathbf{I_1+I_2}$	32.82%	59.38%	68.69%	0.44	44.09%
Baseline	I_1+I_3	32.49%	56.32%	65.54%	0.43	42.69%
Baseline	I_2+I_3	32.16%	57.32%	66.45%	0.43	42.65%
Baseline	I_1+I_4	33.16%	58.81%	68.27%	0.44	43.97%
Baseline	$I_1+I_2+I_3$	32.94%	57.98%	66.31%	0.44	43.38%
Baseline	$I_2+I_3+I_4$	33.46%	59.07%	67.38%	0.44	43.88%
Baseline	$I_3+I_4+I_1$	33.39%	57.34%	67.13%	0.44	43.52%
Baseline	$\mathbf{I_1+I_2+I_4}$	**34.42%**	**60.30%**	**69.08%**	**0.46**	**45.44%**
Baseline	all	34.02%	58.86%	67.08%	0.46	44.44%
Bug reports with good baseline queries and no localization hints (**567**) ($\mathbf{HQ_{H-}}$)						
Baseline	–	41.96%	86.18%	100.00%	0.60	58.15%
Baseline	I_1+I_2	38.12%	76.51%	87.27%	0.54	51.68%
Baseline	I_1+I_4	41.35%	78.92%	89.64%	0.57	55.07%
Baseline	$I_1+I_2+I_4$	43.24%	79.39%	90.12%	0.58	55.71%
Bug reports with good baseline queries and with localization hints (**954**) ($\mathbf{HQ_{H+}}$)						
Baseline	–	50.01%	86.75%	100.00%	0.66	66.63%
Baseline	I_1+I_2	50.44%	83.02%	92.01%	0.65	65.67%
Baseline	I_1+I_4	50.63%	81.29%	90.53%	0.64	64.84%
Baseline	$I_1+I_2+I_4$	52.71%	83.83%	92.21%	0.66	67.61%
Bug reports with poor baseline queries and no localization hints (**372**) ($\mathbf{LQ_{H-}}$)						
Baseline	–	0.00%	0.00%	0.00%	0.00	0.00%
Baseline	I_1+I_2	2.05%	7.74%	15.51%	0.05	5.19%
Baseline	$\mathbf{I_1+I_4}$	1.28%	7.37%	**16.09%**	0.04	4.50%
Baseline	$I_1+I_2+I_4$	2.23%	7.33%	15.01%	0.05	4.79%
Bug reports with poor baseline queries and with localization hints (**427**) ($\mathbf{LQ_{H+}}$)						
Baseline	–	0.00%	0.00%	0.00%	0.00	0.00%
Baseline	$\mathbf{I_1+I_2}$	6.79%	**23.06%**	**34.29%**	**0.14**	**13.79%**
Baseline	I_1+I_4	**7.01%**	20.04%	31.25%	0.13	12.80%
Baseline	$I_1+I_2+I_4$	6.96%	22.41%	32.95%	0.14	13.70%

Emboldened: Original version of Baseline and the best-performing combinations of actionable insights

bug reports. Although the query expansion based on our actionable insights did not help, the expanded queries still remained of high-quality (e.g., 90% Hit@10). However, such an expansion improved the poor baseline queries, i.e., collected from LQ_{H-} and LQ_{H+} bug reports, with a significant margin. For example, the poor baseline queries fail to return any ground truth within their Top-10 results (i.e., 0.00% Hit@10). When more keywords were added to them using two actionable insights on frequency and entropy (i.e., I_1+I_2), the expanded versions achieve up to 23% Hit@5, 34% Hit@10, 0.14 MRR and 14% MAP, which are highly promising.

In Table 22, we investigate the effectiveness of four insights both in isolation and in combination by applying them to the queries of STRICT technique [57]. We see that expansion of STRICT queries with less frequent keywords (i.e., I_1)

Table 22: Improvement of STRICT Queries with Actionable Insights (Using Top-10 Results Only)

Technique	**Insight**	**Hit@1**	**Hit@5**	**Hit@10**	**MRR**	**MAP**
Performance with all bug reports (**2,320**)						
STRICT	–	**25.82%**	**52.28%**	**63.02%**	**0.37**	**37.31%**
STRICT	**I_1**	33.97%	59.56%	68.60%	0.45	44.85%
STRICT	I_2	30.75%	56.43%	67.38%	0.42	42.09%
STRICT	I_3	33.53%	57.74%	67.10%	0.44	43.86 %
STRICT	I_4	30.35%	56.59%	65.57%	0.41	41.45%
STRICT	I_1+I_2	35.40%	61.02%	69.92%	0.46	46.19%
STRICT	I_1+I_3	35.06%	59.81%	68.75%	0.46	45.48%
STRICT	I_2+I_3	34.46%	58.85%	67.45%	0.45	44.72%
STRICT	**I_1+I_4**	34.87%	62.02%	70.41%	0.46	46.04%
STRICT	I_2+I_4	34.43%	61.06%	70.34%	0.46	45.70%
STRICT	I_3+I_4	34.09%	60.11%	69.50%	0.45	45.12%
STRICT	$I_1+I_2+I_3$	**35.50%**	60.59%	69.43%	0.46	46.03%
STRICT	$I_2+I_3+I_4$	34.76%	61.35%	69.99%	0.46	45.79%
STRICT	$I_3+I_4+I_1$	35.04%	62.14%	70.28%	0.46	46.05%
STRICT	**$I_1+I_2+I_4$**	35.35%	**62.58%**	**70.90%**	**0.47**	**46.52%**
STRICT	all	35.41%	62.06%	70.63%	0.46	46.31%
Bug reports with good baseline queries and no localization hints (567) (**$\mathbf{HQ}_{H-}$**)						
STRICT	-	36.14%	72.55%	84.83%	0.51	50.29%
STRICT	**$I_1+I_2+I_4$**	**41.90%**	**82.74%**	**92.12%**	**0.58**	**56.03%**
Bug reports with good baseline queries and with localization hints (954) (**$\mathbf{HQ}_{H+}$**)						
STRICT	–	36.53%	69.86%	82.41%	0.51	51.85%
STRICT	**$I_1+I_2+I_4$**	**55.17%**	**88.00%**	**95.26%**	**0.69**	**70.09%**
Bug reports with poor baseline queries and no localization hints (372) (**$\mathbf{LQ}_{H+}$**)						
STRICT	–	1.56%	9.54%	16.89%	0.05	5.12%
STRICT	$I_1+I_2+I_4$	2.66%	7.18%	15.18%	0.05	5.09%
Bug reports with poor baseline queries and with localization hints (427) (**$\mathbf{LQ}_{H+}$**)						
STRICT	–	4.91%	18.86%	25.77%	0.11	10.66%
STRICT	**$I_1+I_2+I_4$**	**6.80%**	**23.31%**	**32.83%**	**0.13**	**13.39%**

Emboldened: Original version of STRICT and the best-performing combinations of actionable insights

improve Hit@1, Hit@10, MRR and MAP by 32%, 9%, 22%, and 20% respectively. Addition of nouns from bug reports (i.e., I_1+I_4) further improves these queries and they achieve 70% Hit@10 with 46% MAP, which are 12% and 24% higher than original performance measures. However, a combination of three insights on frequency, entropy and part of speech of keywords (i.e., $I_1+I_2+I_4$) leads to the maximum improvement of 2,320 queries from STRICT – 71% Hit@10, 0.47 MRR and 47% MAP. It should be noted that these performance metrics are 7%, 9%, and 9% higher than their baseline counterparts (Table 21).

We also investigate the impact of our actionable insights upon the STRICT queries from different subsets of bug reports. From Table 22, we see that they improve queries from three subsets of bug reports - HQ_{H-}, HQ_{H+} and LQ_{H+}. STRICT achieves 70%–73% Hit@5, 82%–85% Hit@10, 0.51 MRR and 50%–52% MAP when queries are constructed from the bug reports that lead to good baseline queries (HQ_{H-}, HQ_{H+}). Expansion of these queries applying the actionable insights on keyword frequency, entropy and part of speech ($I_1+I_2+I_4$) leads to 83%–88% Hit@5, 92%–95% Hit@10, 0.58–0.69 MRR, and 56%–70% MAP, which clearly demonstrates the positive impacts of our derived insights. We also notice improvement in query performance due to query expansion, especially for STRICT queries from the bug reports with poor baseline (LQ_{H+}). The original queries from STRICT technique achieves 26% Hit@10 with 11% MAP. Our expanded version of STRICT queries achieve 33% Hit@10 with 13% MAP, which are 27% and 18% higher respectively. All these empirical findings clearly suggest the benefits of our derived actionable insights (Table 20).

According to our investigation, simple keyword removal strategies and existing techniques (e.g., STRICT) perform comparably when dealing with bug reports that lead to poor baseline queries. For example, simply removing the duplicate keywords from a baseline query can lead to 25% Hit@10 for LQ_{H+} bug reports, which is comparable to STRICT's original performance (Table 6). Combination of unique keywords from the title and body sections of a bug report also leads to 34% Hit@10 (Table 21). Thus, further investigation is warranted in this context. Our queries constructed using all 15 possible combinations (i.e., ${}^4C_1 + {}^4C_2 + {}^4C_3 + {}^4C_4$) of the four actionable insights can also be found in the replication package [2] for further investigation and reuse.

7.6 Manual Analysis of Bug Reports Leading to Poor Baseline Queries

Given the feature importance analysis, actionable insights and query improvements above, we further analyze the bug reports that lead to poor baseline queries to gain a deeper understanding of them. Since analyzing all of them could be impractical, we choose a random subset of 120 bug reports. In particular, we randomly select 20 bug reports (10 with localization hints + 10 without localization hints) from each of the six subject systems for our analysis. We go through the *title* and *description* texts of each bug report and attempt to find out important trends using Grounded Theory (GT) approach [20]. In our GT approach, we perform three levels of coding namely *open coding*, *axial coding* and *selective coding*. During open coding, we keep an open mind and attempt to describe a bug report with a set of appropriate key phrases. During axial coding, we put these key phrases on a spreadsheet document and connect the semantically related key phrases. That is, we use the same colour to represent similar concepts. In our selective coding step, we gather the low-level categories from the axial coding and derive high-level theories and meaningful insights from them. The artifacts from our Grounded Theory-based investigation can be found in the replication package [2]. We spent $\approx$20 man-hours for this manual inspection. Based on our analysis, we make several important observations about the bug reports that lead to poor baseline queries (i.e., QE>10) as follows.

First, bug reports leading to poor baseline queries often contain generic, verbose, and noisy textual contents that might not be helpful to understand a reported bug. Although our Genetic Algorithm-based approach was able to extract the optimal and near-optimal queries from them, their keywords might always not be intuitive enough. However, interestingly, many of them match with the keywords from ground truth file names. Furthermore, the percentage of this match has been found as one of the most important features for separating between optimal and non-optimal queries (see Fig. 6 for details).

Second, many of these bug reports (LQ_{H+}) contain noisy stack traces with hundreds of trace lines, configuration information and complex memory dump.

Although these contents could be useful for human developers with relevant experience, they are not helpful enough for constructing appropriate queries using the existing automated tool supports.

Third, many of these bug reports were submitted with screenshots, videos, stack traces and data dump in the attachment. Therefore, the submitter did not feel the necessity of writing a detailed bug report using texts. As a result, sufficient textual contents were not available to the existing techniques for constructing appropriate queries. Thus, their queries were not effective for localizing the bugs.

Fourth, several of these bug reports were submitted with patched code. Developers often identify software bugs in an application, fix themselves and then submit both the bug reports and fixed code. Since the bug is already fixed, they often either provide a very generic explanation of the bug or submit the placeholder texts. Since existing approaches mostly rely on textual contents, they fail to construct appropriate search queries from these bug reports.

Fifth, many bug reports contain simple code examples as a part of explaining their bugs. However, these examples might always not point towards actual buggy elements within the software code. Thus, despite containing apparent localization hints, these bug reports might not be found useful to construct appropriate queries for IR-based bug localization.

Summary of RQ$_3$: Optimal search queries are significantly different from the non-optimal ones not only in performance but also in their characteristics. The optimal keywords are *less frequent* than the non-optimal ones within a bug report. They are *more specific* (less ambiguous) than non-optimal keywords. Optimal keywords are also more likely to be found within the *description* section of the report. They are more likely to be *nouns* than the non-optimal ones. When these actionable insights are applied, baseline queries and state-of-the-art queries can achieve up to **34**% higher Hit@10 and **27**% higher Hit@10 respectively, which are highly promising. Our manual analysis with Grounded Theory approach also provides further insights on why the traditional approaches might fail to construct right queries from the bug reports that lead to poor baseline queries.

8 Findings Summary & Discussions

In our empirical study, we conduct three comparative analyses and answer three important research questions. In the following section, we summarize our important findings, actionable insights and also point out the potential gaps in the literature of IR-based bug localization.

(a) **Bug reports could produce poor baseline queries despite containing explicit localization hints.** About 34% (799) of our collected bug reports lead to poor baseline queries (i.e., QE>10) where 18% (427) reports contain explicit hints for localizing the bugs (e.g., program elements, stack traces). Thus, the idea of using localization hints might not be sufficient to extract appropriate search keywords. Through the application of actionable insights, we were able to improve the queries from these bug reports significantly (e.g., 27%–34% improvement in Hit@10), which demonstrates the benefits of our insights.

(b) **State-of-the-art approaches are not enough.** The proxies adopted by state-of-the-art approaches for keyword importance (e.g., TF, TF-IDF, TextRank [57]) might not be sufficient to capture appropriate keywords from the bug reports that lead to poor baseline queries. Although these reports might sometimes contain explicit hints for localizing bugs, on average, their textual contents are generic, verbose, and noisy. Thus, the state-of-the-art approaches often fail to choose the right queries from these reports and as a result, could perform poorly in the IR-based bug localization (Tables 6, 10).

(c) **Graph-based approaches have more potential for search keyword selection.** According to RQ_1, graph-based approaches (e.g., STRICT, ACER) provide better search queries than frequency-based counterparts regardless of the bug report clusters (e.g., Table 6 Section 4.5) or the origin of their keywords (e.g., bug reports, source code) (Section 5). Graph-based approaches were able to extract important keywords even from the bug reports that lead to poor baseline queries, which demonstrates their promising aspect.

(d) **Optimal search keywords exist even in the bug reports leading to poor baseline queries.** Contrary to the popular beliefs [36, 72], our investigation (RQ_2) suggests that bug reports that lead to poor baseline queries contain sufficient keywords to form optimal or near-optimal search queries. Our finding thus strengthens a similar observation of Mills et al. [46]. Optimal and near-optimal queries from these bug reports achieve 78%–93% Hit@10, which are three to five times higher than the state-of-the-art Hit@10 (Table 16).

(e) **Optimal queries are different from non-optimal queries.** Optimal keywords are less frequent than the non-optimal ones within a bug report. They are more specific (less ambiguous) than the non-optimal keywords. Unlike keywords chosen by existing approaches [32, 57], optimal keywords are more likely to be found within the *description* section of a bug report. They are also more likely to be nouns than the non-optimal keywords. Thus, optimal keywords are significantly different from the non-optimal ones in several aspects.

(f) **Optimal search keywords are limited in number.** According to our investigation, the average length of an optimal query does not vary across subject systems. When candidate queries with 5 keywords were provided, our GA-based algorithm returned optimal queries with an average length of 5 keywords. However, the candidates with 10, 15, 20 keywords ended up with the optimal queries containing 9, 12, 14 keywords, on average. We also performed keyword overlap analysis, and found that the overlap ratio ranged from 40% to 60%. That is, out of 5 optimal keywords, 2-3 keywords overlap with a 9-keywords optimal query, on average. When we repeat this process between 12-keywords and 14-keywords optimal queries, we notice an overlap of 6 keywords, which also indicates 50% overlap (6/12). Thus, optimal queries overlap significantly although they might have different initial candidates. That is, regardless of various initial keywords (from a bug report) in candidate queries, optimal candidates are likely to converge to a small number of important keywords (a.k.a., optimal keywords) through evolution.

(g) **Simple keyword removal strategies work!** According to our investigation, simple keyword removal strategies and existing techniques (e.g., STRICT) perform comparably when dealing with bug reports that lead to poor baseline queries. For example, simply removing the duplicate keywords from a baseline query can lead to 25% Hit@10 for LQ_{H+} bug reports, which is comparable to STRICT's original performance (Table 6). Combination of unique keywords from the title and body sections of a bug report also leads to 34% Hit@10 (Table 21). Thus, although these simple strategies might not lead to a few important keywords (e.g., GA-based approach, STRICT), further investigation involving these strategies is warranted.

(h) **Machine intelligence could be effective in recognizing optimal keywords.** About 47% of 799 bug reports that lead to poor baseline queries do not contain any explicit hints for localizing bugs (e.g., stack traces, program elements). According to an earlier study [72], developers often fail to recognize appropriate keywords from these reports. In this study, we perform feature importance analysis (Section 7.3) and attempt to better separate optimal queries from non-optimal ones using machine learning (e.g., Table 18). Our investigation has also resulted into a few actionable insights that were found effective for improving the queries (e.g., Tables 20, 21, 22). Future studies can invest more efforts in recognizing the right search keywords from a bug report and machine learning could be a feasible option.

(i) **Practical fitness function for Genetic Algorithm-based query construction is warranted.** We identify optimal and near-optimal search queries from a bug report using ground truth information (RQ_2) which might be unknown in practice. That is, the GA-based approach discussed in this paper might not be practical. Thus, a fitness function is warranted that can reliably evaluate the fitness of a candidate query (Section 6.1) without needing the ground truth. Such a function has high potential for improving GA-based query construction from any textual entities (e.g., bug reports, change requests) and thus will improve several IR-based software engineering tasks such as bug localization, concept location or code search.

9 Threats to Validity

We identify a few threats to the validity of our findings. In this section, we discuss these threats and necessary steps taken to mitigate them as follows.

9.1 Threats to Internal Validity

Threats to *internal validity* relate to experimental errors and human bias [82]. Construction of the dataset and replication of the existing approaches are two potential sources of these threats. However, we collect the dataset from an existing benchmark [58], and carefully discard the tangled commits and false-positive bug reports (a.k.a., feature requests) through a manual analysis of 60+ man-hours. We employ independent coding of the bug reports by two authors, perform agreement analysis and conflict resolution, and finalized our dataset rigorously. We also use the authors' implementation for two approaches [56, 57] and carefully re-implemented the rest (i.e., unavailable prototypes) with their best settings and parameters (e.g., regression coefficients [32]). Our dataset, implementation of existing approaches and other associated resources are available in the replication package [2] for third party reuse. In this way, we mitigate the threats to internal validity.

According to our analysis, the dataset contains bug reports from multiple versions of a software system whereas the corpus is made of a single snapshot of the codebase, which could be a source of threat. That is, due to this potential misuse of software versions, the bug localization might take place against the fixed code rather than the buggy code. However, we investigate this issue with a limited experiment using 360 bug reports and careful manual analysis (Section 5.2), and demonstrate that our core findings remain the same. While this threat has been mitigated, we nevertheless recommend a consistent use of the bug reports and their software versions for the relevant future studies.

9.2 Threats to External Validity

Threats to *external validity* relate to the generalizability of the findings [82]. The findings based on single benchmark might always not generalize to others. We thus conduct a parallel experiment using 332 bug reports from another benchmark dataset (e.g., SWT, ZXing, AspectJ) [85], and was able to partially replicate our findings (RQ_1 and RQ_2). In particular, we reproduced that (1) graph-based approaches indeed provide higher quality keywords than the frequency-based ones, and (2) near-optimal performance could be achieved for 65% of the bug reports. NrOptimal_{GA} achieves 8% higher performance on natural language-only bug reports from our dataset (Table 13, RQ_2) which could fall within the margin of error. Thus, the threats to external validity might also be mitigated.

9.3 Threats to Construct Validity

Threats to construct validity are associated with the appropriateness of the performance metrics adopted in a study. We use four performance metrics (Section 4.4) that are widely adopted by the existing studies including the state-of-the-art on IR-based concept/bug localization [56, 57, 58, 65, 74, 85]. Thus, such threats are also mitigated.

9.4 Threats to Conclusion Validity

Threats to *conclusion validity* arise when conclusions about relationships between two given variables are not reasonable [40]. We answer three research questions using 2,320 bug reports, ten existing approaches including the state-of-the-art and a GA-based approach. We also use statistical tests (e.g., *Wilcoxon Signed Rank*, *Cliff's delta*) and report the test details (e.g., *p-value*, δ) before drawing any conclusion. Thus, such threats are also mitigated.

10 Related Work

10.1 IR-Based Bug Localization

Automatic localization of software bugs has been an active area of research for the last five decades [78]. While traditionally static and dynamic analyses are used [78, 84], Information Retrieval has been adopted in the bug localization for more than a decade. There have been several attempts using complex models such as Latent Semantic Indexing [42, 53] or Latent Dirichlet Allocation (LDA) [62]. However, existing findings [9, 62] suggest that simpler models (e.g., VSM) often perform higher than the complex ones in bug localization. Towards this end, there have been a number of IR-based approaches [65, 69, 73, 74, 75, 76, 81, 85]. Zhou et al. [85] first use revised Vector Space Model (rVSM) and past bug reports for bug localization using Information Retrieval. Saha et al. [65] leverage the structures of both bug reports and source code documents to improve the bug localization. Several studies [69, 73, 74, 75] make use of code change history, and combine them with Information Retrieval. A few studies [47, 58, 76] leverage the structured items (e.g., stack traces) found in the bug report to boost up the localization performance. However, recent findings [34, 36, 59, 72] suggest that IR-based localization could be limited due to poor queries from bug reports. In particular, the existing approaches might not perform well without the presence of localization hints (e.g., program elements) in the bug reports (queries). We *investigate* this *limitation* of IR-based localization through an empirical study, and attempt to better understand the underlying causes. Appropriate query construction is a major step of any IR-based text retrieval [25, 70]. Our investigation suggests that appropriate queries might not have been used by many of the existing approaches.

10.2 Query Reformulation in IR-Based Bug Localization

There have been a number of studies on query construction/reformulation that support the developers during concept location [19, 25, 32, 55, 56, 57, 64], feature/concern location [28, 52, 67, 83] and bug localization [14, 15, 58, 70]. One key challenge of any automated query construction task is – *appropriate keyword selection.* Existing studies can be classified into two broad categories based on their keyword selection – *frequency based* and *graph-based.* TF-IDF [31] has been a popular frequency-based term weighting method for the last five decades. Several existing studies [25, 32, 33, 46, 48] employ TF-IDF and its variants to identify the important keywords from a body of texts (e.g., bug report, source code). Kevic and Fritz [32] use TF-IDF and three heuristics to identify the important keywords from a bug report. Gay et al. [19] employ Rocchio's expansion [64] where they use TF-IDF for keyword selection from the source code. Haiduc et al. [25] later employ three frequency-based term weighting methods –Rocchio [64], RSV [63] and Dice [13], and deliver the best performing query keywords from the source code using machine learning. Sisman and Kak [70] leverage *spatial code proximity*, and suggest such terms that frequently co-occur with the query keywords within the source code. We select a total of *eight* frequency-based approaches (including six above [13, 31, 32, 63, 64, 70]) for our study. Although TF-IDF has been popular, it fails to capture a term's contexts (e.g., surrounding terms) during term weighting, which is a major issue [10, 43]. Rahman and Roy [57] first capture semantic and syntactic dependencies among the words, employ PageRank algorithm [11] on the constructed graph, and then deliver appropriate search keywords from a bug report. They later adopt similar methodologies, and suggest search keywords from source code documents [56] and noisy bug reports containing stack traces [58]. To the best of our knowledge, these are the state-of-the-art approaches for graph-based keyword selection from bug reports and source code. We thus employ two of the above approaches [56, 57] for our empirical study. The detailed comparison between frequency-based and graph-based approaches can be found in Section 5.

In terms of methodology and research goals, our work is closely related to Mills et al. [46]. Like ours, they also adopt a Genetic Algorithm to construct near-optimal search queries, and suggest that bug reports are often sufficient for constructing the right queries that can localize the bugs using Information Retrieval. Besides confirming and strengthening their finding, we conduct further investigation and report several major findings. First, bug reports could produce good or poor baseline queries regardless of the presence of explicit bug localization hints (e.g., stack traces, program elements) in their texts (RQ_1). Second, state-of-the-art algorithms are not sufficient enough for constructing appropriate queries from the bug reports that lead to poor baseline queries (i.e., QE>10) (RQ_2). Third, optimal or near-optimal search queries might not be badly affected by the length of their candidate queries (Table 15). Both Query Effectiveness and Mean Average Precision are found suitable as a fitness function for GA-based query construction (RQ_2). Fourth, optimal keywords are less frequent and less ambiguous than non-optimal ones. They are more likely to be found within the description section of a bug report and also more likely to be nouns(RQ_3). Fifth, our derived insights are actionable and they were able to improve both baseline and state-of-the-art queries significantly (Tables 21, 22) (RQ_3). Such comprehensive investigations were not reported by the earlier studies, which makes our work *novel.*

There also exist a few other studies that employ query construction/reformulation in the bug localization [14, 15], concern/concept location [28, 30, 67, 80] and duplicate bug report detection [16]. However, they construct their queries through natural language discourse analysis or software repository mining, which could be hard to replicate in an empirical study setting. We thus do not include them in our experiments although they are relevant. Besides, many of them were outperformed by the later studies. However, our study performs a detailed empirical study that examines the query construction practices in IR-based bug localization and also delivers several actionable insights, which could inspire future investigations.

11 Conclusion & Future Work

IR-based bug localization approaches have been widely used because of being light-weight and cost-effective. However, quality of their search queries might impact their localization accuracies. In fact, recent studies report mixed findings on the performance of IR-based localization. While most studies suggest that IR-based approaches perform poorly with natural language-only bug reports as queries, there is one study suggesting that even these bug reports may have sufficient keywords for successfully detecting the bugs. These findings could potentially make one believe that natural language-only bug reports are a sufficient source of good queries. In this study, we attempted to shed light on this pressing controversial issue by conducting an empirical study using 2,320 bug reports (939 natural language-only + 1,381 with localization hints), and ten existing approaches including the state-of-the-art on query construction for IR-based bug localization. We also employ a Genetic Algorithm-based approach for constructing optimal and near-optimal search queries from the bug reports. We conduct three comparative analyses, and answer three research questions. Our study reports several major findings. First, bug reports might lead to poor search queries despite containing explicit hints for localizing bugs (e.g., stack traces). On the other hand, bug reports containing only natural language texts might provide high-quality search queries for IR-based bug localization. Second, even the state-of-the-art approaches are not sufficient enough to deliver appropriate queries from the bug reports that lead to poor baseline queries. Third, graph-based approaches are better than frequency-based ones in selecting query keywords. Fourth, optimal search queries from bug reports are significantly different from the non-optimal queries in several aspects (e.g., frequency, entropy, keyword position). Fifth, the comparison between optimal and non-optimal queries led us to several actionable insights that were found useful to significantly improve existing search queries. Our study inspires several future research opportunities as follows.

- Machine intelligence might be leveraged to recognize the right keywords from the natural language-only bug reports since they do not contain any explicit hints for localizing the bugs. Ours is a first attempt towards this direction that compares between optimal and non-optimal queries through feature importance analysis and machine learning.
- Genetic Algorithms (GA) might also be used to identify the optimal or near-optimal search keywords from a bug report given that an appropriate fitness function is available. Our fitness function discussed in the paper might not be feasible for practical use since it relies on ground truth information. Thus, future work can also focus on developing a practical fitness function for GA-based query construction that does not rely on the ground truth.

Acknowledgement

This research was supported by Tenure-track startup grant, Dalhousie University, Saskatchewan Innovation & Opportunity Scholarship (2017–2018), and the Natural Sciences and Engineering Research Council of Canada (NSERC).

A Query Attributes for the Comparative Analysis

Table 23: Query Attributes Used for the Comparative Analysis

Attribute	**Description**	**Formula**
avgTF	Average frequency of all keywords from the search query Q within the texts of a bug report	$\frac{1}{\|Q\|}\sum_{q\in Q} TF(q)$
medianTF	Median frequency of all keywords from the query Q	$\underset{q\epsilon Q}{median}\ (TF(q))$
maxTF	Maximum frequency of all keywords from the query Q	$\underset{q\epsilon Q}{max}\ (TF(q))$
stdTF	Standard deviation of all keyword frequencies from the query Q	$\sqrt{\frac{1}{\|Q\|}\sum_{q\in Q}(TF(q)-\overline{TF})^2}$
avgIDF	Average Inverse Document Frequency of all keywords from the query Q within the corpus	$\frac{1}{\|Q\|}\sum_{q\in Q} IDF(q)$
medianIDF	Median Inverse Document Frequency of all keywords from the query Q	$\underset{q\epsilon Q}{median}\ (IDF(q))$
maxIDF	Maximum Inverse Document Frequency of all keywords from the query Q	$\underset{q\epsilon Q}{max}\ (IDF(q))$
stdIDF	Standard deviation of Inverse Document Frequencies of all the keywords from the query Q	$\sqrt{\frac{1}{\|Q\|}\sum_{q\in Q}(IDF(q)-\overline{IDF})^2}$
avgTFIDF	Average TF-IDF score of all keywords from the query Q	$\frac{1}{\|Q\|}\sum_{q\in Q} TF(q)\times IDF(q)$
medianTFIDF	Median TF-IDF score of all keywords from the query Q	$\underset{q\epsilon Q}{median}\ (TFIDF(q))$
maxTFIDF	Maximum TF-IDF score of all keywords from the query Q	$\underset{q\epsilon Q}{max}\ (TFIDF(q))$
stdTFIDF	Standard deviation of TF-IDF scores of all the keywords from the query Q	$\sqrt{\frac{1}{\|Q\|}\sum_{q\in Q}(TFIDF(q)-\overline{TFIDF})^2}$
avgEntropy	Average entropy of all keywords from the query Q	$\frac{1}{\|Q\|}\sum_{q\in Q} entropy(q)$
medianEntropy	Median entropy of all keywords from the query Q	$\underset{q\epsilon Q}{median}\ (entropy(q))$
maxEntropy	Maximum entropy of all keywords from the query Q	$\underset{q\epsilon Q}{max}\ (entropy(q))$
stdEntropy	Standard deviation of entropy measures of all the keywords from the query Q	$\sqrt{\frac{1}{\|Q\|}\sum_{q\in Q}(entropy(q)-\overline{entropy})^2}$
JSD	Jensen Shannon Divergence is a symmetric version of Kullback-Leibler Divergence between two different probability distributions P and Q	$\frac{1}{2}(KLD(P\|\|M)+KLD(Q\|\|M))$
QSI	Query Specificity Index [24]	1 - $\underset{q\epsilon Q}{median}\ (entropy(q))$
nounRatio	Fraction of all noun keywords N (from the bug report) that are found in the query Q	$\frac{1}{\|N\|}\sum_{q\in Q} isNoun(q)$
verbRatio	Fraction of all verb keywords V (from the bug report) that are found in the query Q	$\frac{1}{\|V\|}\sum_{q\in Q} isVerb(q)$

nounVerbRatio	Fraction of all the noun and verb keywords NV (from the bug report) that are found in the query Q	$\frac{1}{\|NV\|}\sum_{q\in Q} isNounOrVerb(q)$
otherPOSRatio	Fraction of all the non-noun and non-verb keywords	$1 - nounVerbRatio$
titleKeywordRatio	Fraction of query keywords that are only found within the title of a bug report	$\frac{1}{\|Q\|}\sum_{q\in Q} inTitle(q)$
bodyKeywordRatio	Fraction of query keywords that are only found within the body of a bug report	$\frac{1}{\|Q\|}\sum_{q\in Q} inBody(q)$
titleBodyKWRatio	Fraction of query keywords that are found both in title and in body	$\frac{1}{\|Q\|}\sum_{q\in Q} inTitleBody(q)$
uniqueKeywords	Number of unique keywords in the query	$\|removeDuplicates(Q)\|$
gtTermRatio	Fraction of query keywords that are found in ground truth class names	$\frac{1}{\|Q\|}\sum_{q\in Q} inGroundTruth(q)$
avgPMI	Average Point-wise Mutual Information over all pairs of query keywords	$\frac{2(\|Q\|-1)!}{(\|Q\|)!}\sum_{q1,q2\in Q} PMI(q1,q2)$
medianPMI	Median PMI measure over all pairs of query keywords	$\underset{q1,q2\epsilon Q}{median}\ (PMI(q1,q2))$
maxPMI	Maximum PMI measure over all pairs of query keywords	$\underset{q1,q2\epsilon Q}{max}\ (PMI(q1,q2))$
stdPMI	Standard deviation of PMI measures over all pairs of query keywords	$\sqrt{\frac{1}{\|Q\|}\sum_{q1,q2\in Q}(PMI(q1,q2)-\overline{PMI})^2}$

$$\mathbf{M(w)} = \frac{1}{2}(P(w)+Q(w)), \quad \mathbf{entropy(q)} = \sum_{d\in D}\frac{TF(q,d)}{TF(q,D)} \times \log_D \frac{TF(q,d)}{TF(q,D)}$$

$$\mathbf{PMI(q1,q2)} = log\frac{P_{q1,q2}(D)}{P_{q1}(D),P_{q2}(D)},\ P_{q1,q2}(D) = \frac{|D_{q1}\cap D_{q2}|}{|D|},\ P_q(D) = \frac{|D_q|}{|D|}$$

B Comparison of Query Feature Distribution

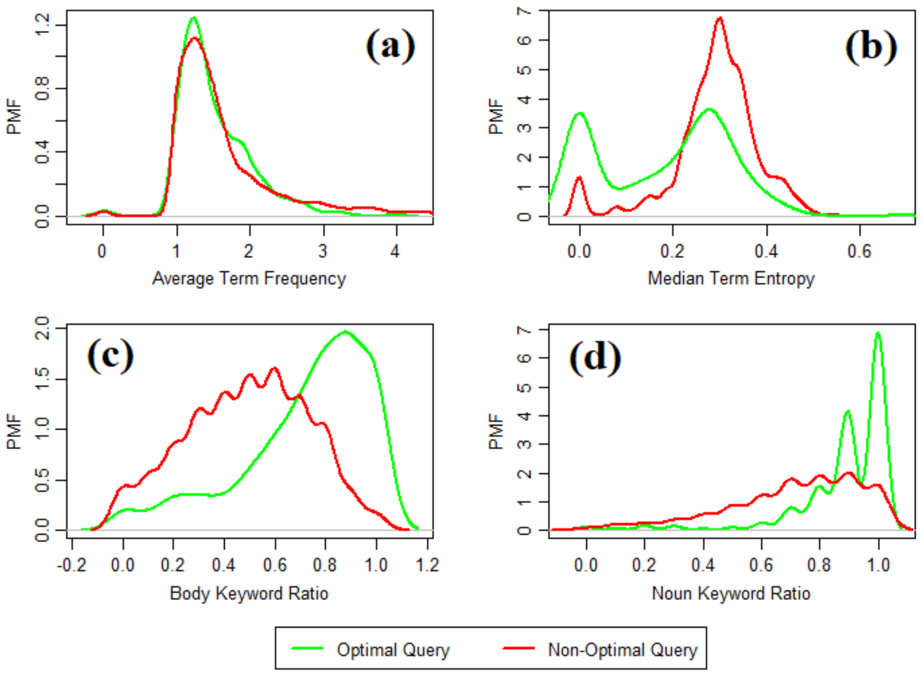

Fig. 9: Comparison between optimal and non-optimal keywords (from bug reports leading to poor baseline queries) using their distributions (e.g., PMF=probability mass function) of (a) frequency, (b) entropy, (c) percentage from the body/description section of a bug report, and (d) percentage of nouns

C Comparison of Query Features using Box plots

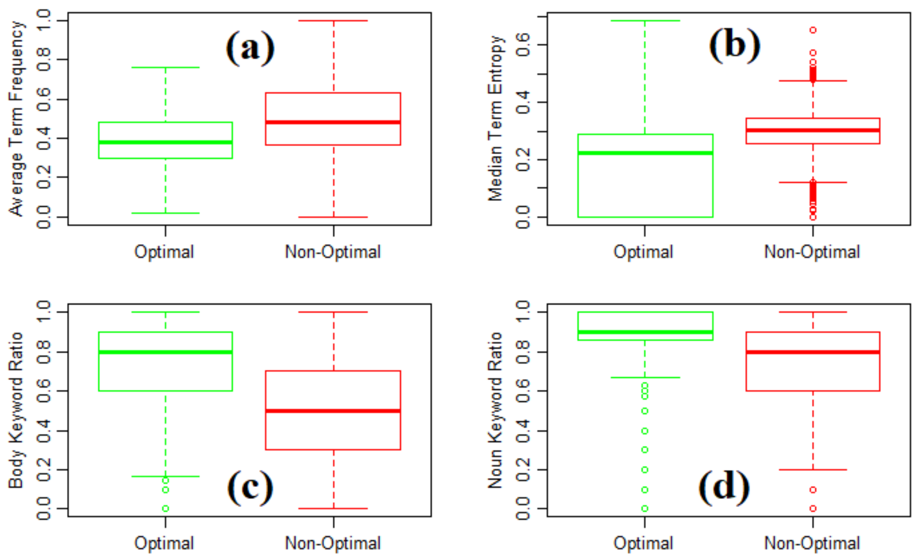

Fig. 10: Comparison between optimal and non-optimal keywords (from bug reports leading to poor baseline queries) using their box plot of (a) frequency, (b) entropy, (c) percentage from the body/description section of a bug report, and (d) percentage of nouns

References

1. Report: Software failure caused $1.7 trillion in financial losses in 2017. URL `https://tek.io/2FBNl2i`.
2. EMSE 2019 replication package. URL `https://github.com/masud-technope/EMSE-2019-Replication-Package`.
3. Cost of software debugging, 2019. URL `https://goo.gl/okoj21`.
4. Blizzard-experimental data, 2019. URL `https://goo.gl/toCZrs`.
5. Apache Lucene Core, 2019. URL `https://lucene.apache.org/core`.
6. A Arif, M M Rahman, and S Y Mukta. Information Retrieval by Modified Term Weighting Method Using Random Walk Model with Query Term Position Ranking. In *Proc. ICSPS*, pages 526–530, 2009.
7. A Bachmann and A Bernstein. Software Process Data Quality and Characteristics: A Historical View on Open and Closed Source Projects. In *Proc. IWPSE*, pages 119–128, 2009.
8. S. Bajracharya, T. Ngo, E. Linstead, Y. Dou, P. Rigor, P. Baldi, and C. Lopes. Sourcerer: A search engine for open source code supporting structure-based search. In *Proc. OOPSLA-C*, pages 681–682, 2006.

9. G Bavota, A De Lucia, R Oliveto, A Panichella, F Ricci, and G Tortora. The Role of Artefact Corpus in LSI-based Traceability Recovery. In *Proc. TEFSE*, pages 83–89, 2013.
10. R Blanco and C Lioma. Graph-based Term Weighting for Information Retrieval. *Inf. Retr.*, 15(1):54–92, 2012.
11. S Brin and L Page. The Anatomy of a Large-Scale Hypertextual Web Search Engine. *Comput. Netw. ISDN Syst.*, 30(1-7):107–117, 1998.
12. D Carmel, E Yom-Tov, A Darlow, and D Pelleg. What Makes a Query Difficult? In *Proc. SIGIR*, pages 390–397, 2006.
13. C Carpineto and G Romano. A Survey of Automatic Query Expansion in Information Retrieval. *ACM Comput. Surv.*, 44(1):1:1–1:50, 2012.
14. O Chaparro and A Marcus. On the Reduction of Verbose Queries in Text Retrieval Based Software Maintenance. In *Proc. ICSE-C*, pages 716–718, 2016.
15. O Chaparro, J M Florez, and A Marcus. Using Observed Behavior to Reformulate Queries during Text Retrieval-based Bug Localization. In *Proc. ICSME*, pages 376–387, 2017.
16. O. Chaparro, J. M. Florez, U. Singh, and A. Marcus. Reformulating queries for duplicate bug report detection. In *Proc. SANER*, page 12, 2019.
17. N. V. Chawla, K. W. Bowyer, L. O. Hall, and W. P. Kegelmeyer. Smote: Synthetic minority over-sampling technique. *J. Artif. Int. Res.*, 16(1):321–357, 2002.
18. G W Furnas, T K Landauer, L M Gomez, and S T Dumais. The Vocabulary Problem in Human-system Communication. *Commun. ACM*, 30(11):964–971, 1987.
19. G Gay, S Haiduc, A Marcus, and T Menzies. On the Use of Relevance Feedback in IR-based Concept Location. In *Proc. ICSM*, pages 351–360, 2009.
20. B. G. Glaser and A. L. Strauss. *The discovery of grounded theory : strategies for qualitative research.* Chicago : Aldine Publishing, 1967.
21. C Le Goues, T Nguyen, S Forrest, and W Weimer. GenProg: A Generic Method for Automatic Software Repair. *TSE*, 38(1):54–72, 2012.
22. S. Haiduc, J. Aponte, and A. Marcus. Supporting program comprehension with source code summarization. In *Proc. ICSE*, volume 2, pages 223–226, 2010.
23. S Haiduc, G Bavota, R Oliveto, A De Lucia, and A Marcus. Automatic Query Performance Assessment During the Retrieval of Software Artifacts. In *Proc. ASE*, pages 90–99, 2012.
24. S Haiduc, G Bavota, R Oliveto, A Marcus, and A De Lucia. Evaluating the Specificity of Text Retrieval Queries to Support Software Engineering Tasks. In *Proc. ICSE*, pages 1273–1276, 2012.
25. S Haiduc, G Bavota, A Marcus, R Oliveto, A De Lucia, and T Menzies. Automatic Query Reformulations for Text Retrieval in Software Engineering. In *Proc. ICSE*, pages 842–851, 2013.
26. S Hassan, R Mihalcea, and C Banea. Random-Walk Term Weighting for Improved Text Classification. In *Proc. ICSC*, pages 242–249, 2007.
27. K. Herzig and A. Zeller. The impact of tangled code changes. In *Proc. MSR*, pages 121–130, 2013.
28. E Hill, L Pollock, and K Vijay-Shanker. Automatically Capturing Source Code Context of NL-queries for Software Maintenance and Reuse. In *Proc. ICSE*, pages 232–242, 2009.

29. E Hill, S Rao, and A Kak. On the Use of Stemming for Concern Location and Bug Localization in Java. In *Proc. SCAM*, pages 184–193, 2012.
30. M J Howard, S Gupta, L Pollock, and K Vijay-Shanker. Automatically Mining Software-based, Semantically-Similar Words from Comment-Code Mappings. In *Proc. MSR*, pages 377–386, 2013.
31. K. S. Jones. A statistical interpretation of term specificity and its application in retrieval. *J. Doc.*, 28(1):11–21, 1972.
32. K Kevic and T Fritz. Automatic Search Term Identification for Change Tasks. In *Proc. ICSE*, pages 468–471, 2014.
33. K Kevic and T Fritz. A Dictionary to Translate Change Tasks to Source Code. In *Proc. MSR*, pages 320–323, 2014.
34. M. Kim and E. Lee. Are information retrieval-based bug localization techniques trustworthy? In *Proc. ICSE*, pages 248–249, 2018.
35. P. S. Kochhar, T. B. Le, and D. Lo. It's not a bug, it's a feature: Does misclassification affect bug localization? In *Proc. MSR*, pages 296–299, 2014.
36. P. S. Kochhar, Y. Tian, and D. Lo. Potential biases in bug localization: Do they matter? In *Proc. ASE*, pages 803–814, 2014.
37. Tien-Duy B Le, R J Oentaryo, and D Lo. Information Retrieval and Spectrum Based Bug Localization: Better Together. In *Proc. ESEC/FSE*, pages 579–590, 2015.
38. J. Lee, D. Kim, Tegawendé F. Bissyandé, W. Jung, and Y. Le Traon. Bench4bl: Reproducibility study on the performance of ir-based bug localization. In *Proc. ISSTA*, page 61–72, 2018.
39. J. Lin and G. C. Murray. Assessing the term independence assumption in blind relevance feedback. In *Proc. SIGIR*, pages 635–636, 2005.
40. M Linares-Vásquez, G Bavota, M Di Penta, R Oliveto, and D Poshyvanyk. How Do API Changes Trigger Stack Overflow Discussions? A Study on the Android SDK. In *Proc. ICPC*, pages 83–94, 2014.
41. D Liu, A Marcus, D Poshyvanyk, and V Rajlich. Feature Location via Information Retrieval Based Filtering of a Single Scenario Execution Trace. In *Proc. ASE*, pages 234–243, 2007.
42. A Marcus, A Sergeyev, V Rajlich, and J I Maletic. An Information Retrieval Approach to Concept Location in Source Code. In *Proc. WCRE*, pages 214–223, 2004.
43. R Mihalcea and P Tarau. TextRank: Bringing Order into Texts. In *Proc. EMNLP*, pages 404–411, 2004.
44. George A. Miller. Wordnet: A lexical database for english. *Commun. ACM*, 38 (11):39–41, 1995.
45. C. Mills, G. Bavota, S. Haiduc, R. Oliveto, A. Marcus, and A. D. Lucia. Predicting query quality for applications of text retrieval to software engineering tasks. *TOSEM*, 26(1):3:1–3:45, 2017.
46. C. Mills, J. Pantiuchina, E. Parra, G. Bavota, and S. Haiduc. Are bug reports enough for text retrieval-based bug localization? In *Proc. ICSME*, pages 381–392, 2018.
47. L. Moreno, J. J. Treadway, A. Marcus, and W. Shen. On the use of stack traces to improve text retrieval-based bug localization. In *Proc. ICSME*, pages 151–160, 2014.

48. L Moreno, G Bavota, S Haiduc, M Di Penta, R Oliveto, B Russo, and A Marcus. Query-based Configuration of Text Retrieval Solutions for Software Engineering Tasks. In *Proc. ESEC/FSE*, pages 567–578, 2015.
49. Ani Nenkova and Rebecca J. Passonneau. Evaluating content selection in summarization: The pyramid method. In *Proc. HLT-NAACL*, pages 145–152, 2004.
50. A Panichella, B Dit, R Oliveto, M D Penta, D Poshyvanyk, and A D Lucia. Parameterizing and Assembling IR-Based Solutions for SE Tasks Using Genetic Algorithms. In *Proc. SANER*, pages 314–325, 2016.
51. C Parnin and A Orso. Are Automated Debugging Techniques Actually Helping Programmers? In *Proc. ISSTA*, pages 199–209, 2011.
52. F. Perez, J. Font, L. Arcega, and C. Cetina. Automatic query reformulations for feature location in a model-based family of software products. *Data & Knowledge Engineering*, 116:159 – 176, 2018.
53. D Poshyvanyk, Y G Gueheneuc, A Marcus, G Antoniol, and V Rajlich. Feature Location Using Probabilistic Ranking of Methods Based on Execution Scenarios and Information Retrieval. *TSE*, 33(6):420–432, 2007.
54. M. M. Rahman and C. K. Roy. An insight into the unresolved questions at stack overflow. In *Proc. MSR*, pages 426–429, 2015.
55. M M Rahman and C K Roy. QUICKAR: Automatic Query Reformulation for Concept Location Using Crowdsourced Knowledge. In *Proc. ASE*, pages 220–225, 2016.
56. M M Rahman and C K Roy. Improved Query Reformulation for Concept Location using CodeRank and Document Structures. In *Proc. ASE*, pages 428–439, 2017.
57. M M Rahman and C K Roy. STRICT: Information Retrieval Based Search Term Identification for Concept Location. In *Proc. SANER*, pages 79–90, 2017.
58. M. M. Rahman and C. K. Roy. Improving ir-based bug localization with context-aware query reformulation. In *Proc. ESEC/FSE*, pages 621–632, 2018.
59. M. M. Rahman and C. K. Roy. Improving bug localization with report quality dynamics and query reformulation. In *Proc. ICSE-C*, pages 348–349, 2018.
60. M M Rahman, C K Roy, and D Lo. RACK: Automatic API Recommendation using Crowdsourced Knowledge. In *Proc. SANER*, pages 349–359, 2016.
61. M. M. Rahman, C. K. Roy, and R. G. Kula. Predicting usefulness of code review comments using textual features and developer experience. In *Proc. MSR*, pages 215–226, 2017.
62. S Rao and A Kak. Retrieval from Software Libraries for Bug Localization: A Comparative Study of Generic and Composite Text Models. In *Proc. MSR*, pages 43–52, 2011.
63. S. E. Robertson. On term selection for query expansion. *J. Doc.*, 46(4):359–364, 1991.
64. J J Rocchio. *The SMART Retrieval System—Experiments in Automatic Document Processing.* Prentice-Hall, Inc.
65. R K Saha, M Lease, S Khurshid, and D E Perry. Improving Bug Localization using Structured Information Retrieval. In *Proc. ASE*, pages 345–355, 2013.
66. C. E. Shannon. A mathematical theory of communication. *The Bell System Technical Journal*, 27(3):379–423, 1948.

67. D Shepherd, Z P Fry, E Hill, L Pollock, and K Vijay-Shanker. Using Natural Language Program Analysis to Locate and Understand Action-Oriented Concerns. In *Proc. ASOD*, pages 212–224, 2007.
68. Z Shi, J Keung, and Q Song. An Empirical Study of BM25 and BM25F Based Feature Location Techniques. In *Proc. InnoSWDev*, pages 106–114, 2014.
69. B Sisman and A C Kak. Incorporating Version Histories in Information Retrieval Based Bug Localization. In *Proc. MSR*, pages 50–59, 2012.
70. B Sisman and A C Kak. Assisting Code Search with Automatic Query Reformulation for Bug Localization. In *Proc. MSR*, pages 309–318, 2013.
71. K Toutanova, D Klein, C D Manning, and Y Singer. Feature-Rich Part-of-Speech Tagging with a Cyclic Dependency Network. In *Proc. HLT-NAACL*, pages 252–259, 2003.
72. Q Wang, C Parnin, and A Orso. Evaluating the Usefulness of IR-based Fault Localization Techniques. In *Proc. ISSTA*, pages 1–11, 2015.
73. S Wang and D Lo. Version History, Similar Report, and Structure: Putting Them Together for Improved Bug Localization. In *Proc. ICPC*, pages 53–63, 2014.
74. S. Wang and D. Lo. Amalgam+: Composing rich information sources for accurate bug localization. *JSEP*, 28(10):921–942, 2016.
75. M Wen, R Wu, and S C Cheung. Locus: Locating bugs from software changes. In *Proc. ASE*, pages 262–273, 2016.
76. C P Wong, Y Xiong, H Zhang, D Hao, L Zhang, and H Mei. Boosting Bug-Report-Oriented Fault Localization with Segmentation and Stack-Trace Analysis. In *Proc. ICSME*, pages 181–190, 2014.
77. E Wong, J Yang, and L Tan. AutoComment: Mining Question and Answer sites for Automatic Comment Generation. In *Proc. ASE*, pages 562–567, 2013.
78. W. E. Wong, R. Gao, Y. Li, R. Abreu, and F. Wotawa. A survey on software fault localization. *TSE*, 42(8):707–740, 2016.
79. R. Wu, H. Zhang, S. Kim, and S. Cheung. Relink: Recovering links between bugs and changes. In *Proc. ESEC/FSE*, pages 15–25, 2011.
80. J Yang and L Tan. Inferring Semantically Related Words from Software Context. In *Proc. MSR*, pages 161–170, 2012.
81. K C Youm, J Ahn, J Kim, and E Lee. Bug Localization Based on Code Change Histories and Bug Reports. In *Proc. APSEC*, pages 190–197, 2015.
82. T Yuan, D Lo, and J Lawall. Automated Construction of a Software-Specific Word Similarity Database. In *Proc. CSMR-WCRE*, pages 44–53, 2014.
83. S. Zamani, S. Peck Lee, R. Shokripour, and J. Anvik. A noun-based approach to feature location using time-aware term-weighting. *IST*, 56(8):991 – 1011, 2014.
84. Tao Zhang, He Jiang, Xiapu Luo, and Alvin T.S. Chan. A literature review of research in bug resolution: Tasks, challenges and future directions. *The Computer Journal*, 59(5):741–773, 2016.
85. J. Zhou, H. Zhang, and D. Lo. Where should the bugs be fixed? more accurate information retrieval-based bug localization based on bug reports. In *Proc. ICSE*, pages 14–24, 2012.
86. W. Zou, D. Lo, Z. Chen, X. Xia, Y. Feng, and B. Xu. How practitioners perceive automated bug report management techniques. *TSE*, page to appear, 2018.