Reformulating Queries for Duplicate Bug Report Detection

Oscar Chaparro, Juan Manuel Florez, Unnati Singh, Andrian Marcus
The University of Texas at Dallas, Richardson, TX, USA
{ojchaparroa, jflorez}@utdallas.edu, unnati2501@gmail.com, amarcus@utdallas.edu

***Abstract*—When bugs are reported, one important task is to check if they are new or if they were reported before. Many approaches have been proposed to partially automate duplicate bug report detection, and most of them rely on text retrieval techniques, using the bug reports as queries. Some of them include additional bug information and use complex retrieval- or learning-based methods. In the end, even the most sophisticated approaches fail to retrieve duplicate bug reports in many cases, leaving the bug triagers to their own devices. We argue that these duplicate bug retrieval tools should be used interactively, allowing the users to reformulate the queries to refine the retrieval. With that in mind, we are proposing three query reformulation strategies that require the users to simply select from the bug report the description of the software's observed behavior and/or the bug title, and combine them to issue a new query. The paper reports an empirical evaluation of the reformulation strategies, using a basic duplicate retrieval technique, on bug reports with duplicates from 20 open source projects. The duplicate detector failed to retrieve duplicates in top 5-30 for a significant number of the bug reports (between 34% and 50%). We reformulated the queries for a sample of these bug reports and compared the results against the initial query. We found that using the observed behavior description, together with the title, leads to the best retrieval performance. Using only the title or only the observed behavior for reformulation is also better than retrieval with the initial query. The reformulation strategies lead to 56.6%-78% average retrieval improvement, over using the initial query only.**

I. Introduction

Software systems with large user bases constantly receive many bug reports, especially after major releases, as new bugs are encountered by the users [25]. For example, Eclipse [4], a popular open-source development environment, received more than 45,000 issue reports during 2008 alone [80]. Before the reported bugs are confirmed and assigned to developers for fixing, someone usually checks if these bugs were reported before (*i.e.*, the reports duplicate previous reports) [25]. When the duplicates are found, the bug reports are marked accordingly, thus avoiding potentially unnecessary work or complementing the information of their duplicates for bug fixing [13]. When the number of bug reports is large, finding duplicates can be a time-consuming and error-prone activity. For example, among the bugs reported for Eclipse in 2008, more than 3,000 were marked as duplicates [80]. Hence, researchers proposed automated tool support that is meant to save time and improve the accuracy of duplicate bug report detection (see Sec. II). For simplicity, for the remainder of the paper, we will use *duplicate detection* (or just *DD*) for "duplicate bug report detection".

Duplicate detection techniques have two aspects in common. First, most of them rely on some form of text retrieval (TR) or text-based classifiers, where a query is formulated from the full textual description in the new bug report (*i.e.*, title + description) and old bug reports are retrieved (or classified) as duplicate candidates. Some techniques use simple TR models, while others use more complex ones. The common trend in improving such approaches is adding additional information, such as stack traces and other information recorded in the reports. Other approaches also consider information from other sources and/or employ techniques to learn from past duplicate sets. The common premise here is that there are language commonalities between bug reports corresponding to the same bugs [20]. However, even the most sophisticated duplicate detection approaches are far from perfect. Among the ones we review in section II, the highest reported recall rates@10 are around 80%-90% [11, 49, 60, 80, 85]. This means that in 10%-20% of the cases, these techniques fail to return existing duplicates in the top 10 of the retrieved list of bug reports.

Second, all these approaches assume a rather limited usage scenario. The tools use the new bug report as a query and then the user inspects the ranked list of retrieved bug reports to check if any are duplicates of the new bug report. At some point, if a duplicate is not found, the user chooses to stop and mark the bug as *new* or tries some other approach.

A. Motivation

We contend that duplicate bug report detection is different from other text retrieval applications in software engineering, and tool support should address the differences. For example, in applications for other tasks, such as feature/bug localization [30, 58], traceability link recovery [27], or impact analysis [50, 64], the users have a specific information need, such as finding a relevant part of the code. They would formulate a query (sometimes with tool support) and retrieve relevant documents from the code corpus. Then, the users assess the relevance of the retrieved code documents and may reformulate the query to retrieve other documents. They may also use some of the retrieved documents as a starting point for navigating the code. The users eventually *stop only when their information need is met* (*i.e.*, they found the code document of interest), and they can continue solving their main task.

In contrast, during duplicate bug detection, the user stops when she finds a duplicate report or *when she is confident enough that there are no duplicates*. In other words, users

may stop without finding any relevant result, as there is no specific information need that must be met, in form of a retrieved artifact, which is then used to solve the task at hand (*e.g.*, fixing the bug). Not retrieving any duplicates is a likely outcome. Our main goal is to increase the user confidence when she makes the decision to stop. We propose that tool support should return a number, say N, of bug reports, rather than rank the entire search space. More importantly, when a duplicate is not found among the N retrieved candidates, the user should be able to refine the query and retrieve N more, before making a final decision. The expectation is that the reformulated query is better at retrieving the duplicates (if any) in top N than the original query is at retrieving them in top 2N, which would yield higher confidence in the retrieval.

Furthermore, in the other text retrieval applications, when the relevant documents are not found, users rely on their specific information need, the retrieved results, and the software knowledge, for reformulating queries. Differently, during bug report duplicate detection, when no duplicates are returned, the lack of a specific information need and the knowledge about the code or the retrieved results are unlikely to help with query reformulation. The user is also unlikely to have much specific information about the (potentially) thousands of previous reports, which she could leverage for reformulation. Hence, the major challenge is to have a query reformulation strategy that is independent of such factors and would help retrieve duplicates in more cases.

B. Contributions

This paper rethinks tool-supported duplicate bug report detection as a two-step process, using the entire new bug report as a query in the first step (*i.e.*, the initial query), for retrieving N bug reports, and a reformulated query in the second step, for retrieving additional N bug reports, if needed. The paper proposes and evaluates three reformulation strategies for improving duplicate bug report detection. The reformulation strategies are independent of the underlying detection approach and they do not depend on the returned results or any information from other bug reports or external sources. In other words, they are easy to use by any potential user and should work with any existing duplicate bug report detection tool based on bug reports.

Our approach is based on the observation that most bug reports have an inherent structure, consisting of the bug title (BT), the observed behavior (OB), the expected behavior (EB), and the steps to reproduce the noted bug (S2R) [19, 22, 26, 87]. While many bug reports may lack EB and S2R, previous research found that OB is present in most bug reports, *i.e.*, in ~93% of them [22, 26]. Our previous work also found that OB contains salient information that helps locate the corresponding bugs in the code, more than other parts of the bug report [19]. We conjecture that *the information contained in the OB may result in better retrieval of duplicate bug reports*. Intuitively, we expect that OB encodes information unique to the reported bug, while other parts of the bug reports would have elements potentially common to many other bugs. With that in mind, our reformulation strategy is a query reduction approach, where the new query is obtained by retaining only the OB from a bug report (*i.e.*, from the title and description) and removing the rest of the textual bug description. We call this reformulation strategy *OB-DD*.

The bug title also exhibits some of the same properties that OB does. BT is present in all bug reports, hence it can be easily leveraged for query reformulation. Previous research noted that bug reporters create the bug title as a summary of the entire report, meaning that they often select the words that best describe the bug [46, 54]. In addition, nearly all existing duplicate detection techniques rely specifically on the bug title as a feature for retrieval or classification (see Sec. II). Hence, we consider an additional query reduction strategy, where the initial query (*i.e.*, the entire bug report) is reduced to its BT only. We call this reformulation strategy *BT-DD*.

While the bug title may contain words that describe the OB, we conjecture that OB and BT each have unique terms that are important for retrieval. Therefore, we also consider the hybrid reformulation strategy, which reduces the initial query to the OB *and* BT only. We call this combined strategy *OBT-DD*.

The three reformulation strategies can all be employed with any duplicate bug detection tool that relies on the new bug report (and potentially other information) as query, and the bug report contains a part describing the software's observed behavior. If no duplicates are retrieved within the top-N results, when using the entire bug report as an initial query, then the user should select the OB and/or BT and remove the rest of the textual bug description. Elements such as stack traces or code snippets may be retained if the tool uses them. Then, the user can run the new query and retrieve N more bug reports.

We evaluated the three reformulation strategies using a duplicate bug report retrieval approach based on Lucene [1] (see Sec. III-A). We selected 42,226 bug reports with duplicates from 20 open source software projects, used in previous research [20, 80] (see Sec. III-B). We used these reports as initial queries to the DD tool for retrieving the corresponding duplicates in top N, for N={5, 6, ..., 30} (see Secs. III-C and III-D). Lucene retrieved duplicates in top N for 49.4% (N=5) to 66.1% (N=30) of the bug reports used as queries. We sampled a subset of bug reports from those that did not retrieve duplicates in top N, and used OB-DD, BT-DD, and OBT-DD to reformulate them (see Sec. III-E). The reformulated queries retrieved *more duplicates* in top N for 23.6%-26.8% of the sample set (on average), which translates to a 56.6%-78% average improvement over the initial query retrieving duplicates in top 2N. Based on the results (see Sec. IV), we conclude that *all three reformulation strategies lead to the retrieval of more duplicates than inspecting 2N results with the original query*, *with OBT-DD being the best among them*.

II. Background and Related Work

Duplicate bug report detection (DD) has received significant attention in the past years and dozens of approaches have been proposed using a variety of text retrieval, natural language processing, and machine learning techniques. In addition, query

reformulation has been applied for many tasks in software engineering, except for DD.

A. Duplicate Bug Report Detection

As proposed by Lin [52] and reused by Kang [43], duplicate detection approaches can be classified into three categories:

- *Ranking* approaches [11, 15, 16, 39, 40, 49, 52, 53, 60, 65, 70, 71, 74, 80–83, 85, 86], which use a bug report as input (*i.e.*, query) and output a ranked list of duplicate candidates from a corpus of existing bug reports;
- *Binary* approaches [84], which take a bug report as input and label it as either duplicate or not duplicate; and
- *Decision-making* approaches [10, 39, 45, 47], which consider a pair of bug reports and determine whether they are duplicates of each other.

The proposed reformulation strategies are designed to work with most DD approaches, but the evaluation we present in this paper focuses on ranking approaches. Evaluating the strategies on the other two approaches is subject of future work.

Most issue trackers provide two fields that capture the natural language bug description reported by the user, *i.e.*, *title* and *description*. All the approaches presented here combine the textual information found in both fields [14], with a few exceptions. Two approaches use the report's title only [11, 65] and Lerch *et al.* [49] use only a part of the description, *i.e.*, stack traces. As OB-DD and OBT-DD are based on the description, they would not work with these approaches. The approaches from Amoui *et al.* [11] and Prifti *et al.* [65] are similar to BT-DD, which only use the bug title. However, they are DD techniques rather than query reformulation approaches.

Researchers have studied the value of adding extra information from fields other than the title and description to improve duplicate retrieval. The main sources of additional information are the report's creation date [16, 45, 47, 49, 53, 74] and categorical features [10, 15, 39, 45, 47, 53, 70, 71, 74, 80, 84] such as the affected system version or component. Boisselle *et al.* [15] group categorical fields that are unique to two issue trackers into an additional textual field. Domain information, extracted from textbooks, project documentation, or Wikipedia, has also been leveraged, mainly in machine learning approaches [10, 39, 53]. Wang *et al.* [85] use manually-extracted software execution information for detecting duplicates.

More related to our work, Amoui *et al.* [11] used the observed results (*a.k.a.* OB) and the steps to reproduce (S2R) from the bug report as a source of extra information for retrieving duplicates at BlackBerry. This information is found inside a field called *first_email*, which is unique to BlackBerry's issue tracker and includes the OB and S2R by convention. However, this is not the case in most bug reports submitted to open-source projects [19, 42]. Most issue trackers used by these projects do not enforce the existence of this information in the bug report's textual description [19]. Amoui *et al.*'s research does not study the effect of these sources of information on duplicate detection or focus on query reformulation.

Empirical evaluations of DD techniques have used various metrics, depending on the approach type. Ranking approaches are usually evaluated using standard Information Retrieval measures, such as precision, recall, and MAP [11, 15, 16, 39, 40, 47, 49, 53, 70, 71, 80]. These metrics are suitable when all duplicate bug reports in the corpus are considered useful for retrieval since it is possible for a new bug report to have more than one duplicate. However, many researchers reckon that it is sufficient to find a single duplicate, since the triagers' goal is to mark the new report as duplicate. In this case, MRR and Recall Rate@N are usually used [11, 16, 49, 60, 65, 70, 71, 74, 80–82, 85, 86]. We contend that Recall Rate@N is better suited for evaluating DD approaches, since the triager is likely to inspect the top-N results only (rather than the entire ranking), hence, it is the measure we mainly use in our evaluation (see Sec. III). For binary and decision-making approaches, accuracy and true positive/negative rates are typically used [10, 39, 47, 84].

Notable is the work by Hindle *et al.* [40], who propose a continuous querying approach. While not focusing on query reformulation as we do, in Hindle *et al.*'s approach, queries are generated as the user types the bug description. Each new word will trigger the creation of a new query, which consists of the new word and all the words typed before. It is possible that, as the user enters the bug description, a query would be formulated in such a way that roughly corresponds to the OB. For example, if she starts describing the OB in a contiguous sequence of words. In that case, the query would be similar to our reformulated query. However, differently to that approach, we consider already-submitted bug reports as queries and we do not assume that the OB description is sequential.

Finally, it is worth noting that our reformulation approach addresses the cases when duplicates are hard to retrieve. For example, for the new reports that *just-in-time* retrieval tools in issue trackers fail to identify as duplicates [71].

More comprehensive surveys on duplicate bug report detection are published by Kang [43] and Cavalcanti *et al.* [18].

B. Query Reformulation in Software Engineering

Duplicate detection techniques offer no guidance on what to do when a duplicate is not retrieved within the top-N candidates. We argue that *query reformulation* can be applied in such situations to increase the triager's confidence in the retrieval. Query reformulation has not been applied to the problem of detecting duplicates of bug reports. However, researchers have investigated the impact of query reformulation on other software engineering tasks based on text retrieval methods, especially on code search and concept location.

There are three commonly used strategies of query reformulation: *query expansion* [17], which adds alternative terms to a query; *query replacement* [33, 35], which changes part of a query with a set of new terms; and *query reduction* [44, 55, 68], which removes terms from the query.

Query expansion is the most common strategy used for tasks based on source code retrieval. The additional terms to the query can be selected in different ways, for example, by using ontologies (*e.g.*, WordNet) [78] or co-occurring terms from code or external documents [28, 57, 66]; and by applying relevance [31, 62] or pseudo-relevance feedback [36, 79].

In code search, other reformulation approaches include co-occurrence and frequency of query terms with previous results and code [38, 72], thesauri-based reformulation [32, 48, 51], pseudo-relevance feedback from Stack Overflow results [61], and textual similarity between the query and APIs [56].

Query replacement has been utilized mostly for traceability link recovery [33], where similar web and domain-specific documents to the query are leveraged to select a set of candidate terms to replace the initial query. Another strategy is learning frequent terms from existing requirement-regulation trace corpora and using them as the new query [35].

Query reduction is the category that our approach falls into. Our prior research found that removing a small amount of (noisy) terms from the query/bug report leads to substantial improvement in text-retrieval-based bug localization [23]. Similarly, Mills *et al.* [59] found that near-optimal (reduced) queries from bug reports lead to high improvement in code retrieval. Rahman *et al.* [68] employed term co-occurrences and syntactic dependencies to build a query out of the most important terms in a change request. Kevic *et al.* [44] found that specific terms in a change request have the highest predictive power to retrieve the relevant code documents. Other research focused on increasing the weights of query terms that correspond to source code file names [29] or occur in method names and calls [12]. In another work, Rahman *et al.* [67] leveraged term co-occurrences and syntactic dependencies to select the most important terms in a change request as a query. Recently, the same authors proposed weighting and selecting query terms based on how these relate to each other and whether they reference code entities and/or appear in particular parts of the bug reports, *e.g.*, in stack traces [69]. Other research used heuristics to remove irrelevant terms. Haiduc *et al.* [36] discarded non-noun terms or those appearing in more than 25% of the code documents, since their discriminatory power is likely to be low. Rahman *et al.* [66] employed a similar strategy, by leveraging crowd-sourced information.

Our OB-DD reformulation strategy is motivated by our prior work that proposed a query reformulation approach for *low-quality* queries in bug localization based on OB [19]. While the reformulation strategies are similar, their applications and evaluations are quite different. In bug localization, the buggy code element is guaranteed to exist in the code corpus, and the developer must find it eventually. However, in duplicate detection, a duplicate may or may not exist in the bug report corpus, and at one point the developer must choose to stop looking through the list of candidates. These are two distinct scenarios that must be evaluated differently.

III. Empirical Evaluation

We performed an empirical evaluation of the proposed reformulation strategies. The evaluation aims at answering the following research questions:

RQ1: *Do the three query reformulation strategies help duplicate detection approaches retrieve more duplicate bug reports than without query reformulation?*

RQ2: *Which of the three query reformulation strategies retrieve more duplicate bug reports?*

This section details the procedure we followed to answer our research questions, while section IV presents and discusses the evaluation results. We used a Lucene-based approach (Sec. III-A) to detect duplicates for a large set of bug reports (Secs. III-B and III-C). Then, for a subset of the bug reports for which the tool failed to retrieve duplicates in top N (Sec. III-D), we used the three strategies to reformulate the reports (Secs. III-E and III-F) and assessed how many more duplicates are retrieved among the next N candidates (Sec. III-F).

A. Duplicate Bug Report Detector

We selected a DD approach based on Lucene [37], which was used in previous research for retrieving duplicate bug reports [16, 20] and also for retrieving duplicate Stack Overflow posts [20]. Since the OB-based query reformulation strategies do not depend on the underlying duplicate detection approach, we expect the reformulations to be effective for other, more complex duplicate detection approaches as well. However, such an investigation is subject of future research.

Lucene [37] is a retrieval technique implemented in the open source library of the same name [1], which combines the standard information retrieval Boolean model and the Vector Space Model (based on TF-IDF [76]) to compute the similarity between a new bug report (*i.e.*, the query) and an existing report. Lucene is a technique that relies only on textual information to retrieve a ranked list of candidate duplicate reports. Typically, a Lucene query is created by concatenating the bug report's title and description, including any information embedded in these sources (*e.g.*, code snippets). In our evaluation, we used Apache Lucene v5.3.0 [1] with the default similarity measure and parameters (see [8] and [9] for details).

B. Bug Report Data Set

We use data that corresponds to 449,901 bug reports (*a.k.a.* issues) from 20 open source projects (see Table I). This data was constructed based on two data sets used in prior duplicate detection research: the one used by Sun *et al.* [80] (*a.k.a. SDS*) and the one provided by Chaparro *et al.* [20] (*a.k.a. CDS*).

We used the bug reports for three projects from SDS (*i.e.*, Eclipse, Mozilla Firefox, and OpenOffice). The original data in SDS is in the format used by Sun *et al.*'s tool (*i.e.*, REP), which is essentially a set of real-valued vectors corresponding to *n*-grams extracted from the bug reports. Since the reformulation strategies require the bug description, we downloaded the full text and extra meta-information (*e.g.*, bug type, version, *etc.*) of the bug reports used in SDS (from the issue trackers), rather than using the original data set made available by Sun *et al.*

As for CDS, we selected 17 (of 115) projects with the largest number of bug reports. From the projects' issue trackers, we downloaded the same bug reports information as for SDS. Our data set includes all bug reports submitted to the issue trackers of each project, at the time of creation of CDS (*i.e.*, May 2016) and SDS (*i.e.*, Dec. 2007), except for Firefox and OpenOffice. For these projects, our data set contains

TABLE I
STATISTICS OF OUR DUPLICATE DETECTION DATA SET.

Project	# of queries	Total # of bug reports	# of duplicate buckets
Accumulo	79	4,106	72
Ambari	117	14,763	100
ActiveMQ	120	5,936	104
Cassandra	334	11,011	273
Cordova	247	10,208	188
Continuum	89	2,722	68
Drill	186	4,305	155
Eclipse	28,518	209,056	16,833
Groovy	115	7,486	97
Hadoop	231	10,645	194
Hbase	97	15,114	93
Hive	315	12,854	268
Maven	275	4,758	177
M. Firefox	7,585	75,653	4,510
MyFaces	79	3,657	55
OpenOffice	3,019	31,138	1,708
PDFBox	155	3,207	106
Spark	430	12,676	359
Wicket	149	6,077	127
Struts	86	4,529	75
Total	**42,226**	**449,901**	**25,562**

only the subset of bug reports provided by Sun *et al.* [80]. Firefox' bug reports are from 2010 and OpenOffice's reports are from 2008 to 2010 only. The CDS projects use Jira [7] issue tracker, while the SDS projects use Bugzilla [3]. Our data set spans different software types that range from desktop applications (*e.g.*, Eclipse) to frameworks/libraries (*e.g.*, Struts and PDFBox), in a variety of domains, such as web browsing (Firefox), office productivity (OpenOffice), mobile/web development (*e.g.*, Cordova and MyFaces), distributed computing (*e.g.*, Hadoop and Spark), databases (*e.g.*, Cassandra or Hive), development tools (*e.g.*, Groovy and Maven), *etc.*

C. Queries and Ground Truth

We used the duplicate references between bug reports in the issue tracker to determine the set of queries and their respective ground truth, *i.e.*, the duplicate reports. To do so, we first identified buckets of duplicate reports in the data, using direct and transitive duplicate references in the issue tracker (similar to Sadat *et al.*'s approach [75]). For example, if A is a duplicate of B, which is a duplicate of C, with C being the oldest (*i.e.*, submitted first in the issue tracker) and A the youngest (*i.e.*, submitted last), then the respective bucket is {A, B, C}. If A is a duplicate of C, and B is a duplicate of C, then we form the same bucket {A, B, C}.

We sorted the bug reports in each bucket chronologically by creation date (in our example: {C, B, A}), and for each bucket of size n, we created $n - 1$ queries, each having as ground truth the ones that are older from the bucket. The oldest bug report in each bucket is called the *master* bug report [80] and it is the only one that does not reference any past duplicate, hence it is not considered as a query. In the above example, C is the *master*, while A and B are used as queries, with {B, C} and {C} as ground truth, respectively. We created the ground truth for each query, as described in the above example, which includes the list of corresponding past duplicate reports.

Overall, our data sets contain 25,562 duplicate buckets, from which 42,226 queries were created by using the title and description of the bug reports (see Table I). All bug reports in the corpus and all queries were normalized by using standard preprocessing operations. First, we identified the words in the bug report text, and performed code identifier splitting based on the camel case and underscore formats (*e.g.*, *CodeIdentifier* or *code_identifier* would split into *code* and *identifier*). Then, we removed words that are unlikely to contribute to retrieval, namely, special characters (*e.g.*, *#* or *$*), numbers, common English stop words (*e.g.*, *about*, *because*, *the*, *etc.*), Java keywords (*e.g.*, *for*, *while*, *at*, *with*, *like*, *etc.*), and words shorter than three characters. Finally, we applied Porter's algorithm to reduce the words to their root form [63].

It is important to note the differences between the number of queries for the SDS projects presented by Sun *et al.* [80] and the ones presented in Table I. Before discussing such differences, we must point out that each query in the SDS data set uses the *master* bug report as the only duplicate in the ground truth. We claim that the ground truth based on all past duplicates for a query is better suited for evaluating DD approaches, because, when used in practice, users expect to retrieve any of the duplicate bug reports and not necessarily the master bug report only. We use this approach in our evaluation.

We compared the list of queries in SDS and our data set and found 2,741 differences in the following three categories:

1) Queries that are not present in SDS but are present in our data set. We found 1,824 queries that reference duplicate reports transitively or directly, which are missing in SDS.
2) Queries that are not present in our data set but are present in SDS. We found 521 queries that do not reference any duplicate reports at all (by following the query generation approach described above). However, the SDS data set (incorrectly) includes these queries.
3) Queries that are present in both our data set and SDS, but the ground truth (*i.e.*, the *master* report) is different. We found 396 queries for which the issue tracker reports a different master bug report than the one in SDS.

We manually inspected a subset of the reports in each category and confirmed the differences. Our replication package contains the full list of queries with differences [21].

In summary, our data set includes: (1) a set of queries generated from the bug reports submitted on the projects' issue tracker; (2) the past duplicate bug reports for each query, which represent the ground truth; and (3) the entire set of existing bug reports which represents the document search space for DD. The full set of stop words, bug reports, queries, and ground truth, is available in our replication package [21].

D. Low-quality Queries

Our reformulation approach follows the scenario in which the triager issues the *initial query* (using the full text of a bug report) and inspects the top-N candidates returned by the DD technique at hand (*e.g.*, Lucene). If none of the candidates are deemed duplicated (by inspecting their title and/or description), the triager reformulates the query

(via the reformulation strategies) and inspects additional N candidates. In this scenario, the user would inspect a total of 2N candidates. Large N values (say 30 and beyond) would mean that our approach is impractical because, in the worst-case scenario, it would imply inspecting 60 results total, which could demand a significant effort from the user. It is likely that the triager would not assess more than 30 reports. Very small N values (say less than 5) would imply an unrealistic scenario. If the triager finds at least one duplicate report within the top-5 results, then she does not need reformulation. We contend that inspecting 5 to 10 documents (i.e., 10 to 20 documents total, following reformulation) is a realistic scenario for DD. In other words, if a query retrieves the buggy code in top-5/10, then it is likely that no reformulation is needed. Similar thresholds have been used in prior DD research [40, 70, 74, 80]. Since there is no specific research on user behavior during query reformulation for DD, we do not want to limit the evaluation only to the thresholds we consider most realistic. Hence, in this paper, we include results for the threshold set N={5, 6, 7, ..., 30}, which amounts to 26 thresholds total.

Our reformulation strategies focus on queries that fail to retrieve the duplicate reports within the top-N results (*i.e.*, *low-quality* queries). Therefore, in order to determine the set of *low-quality* queries, we executed Lucene with the initial queries (generated from the entire text of the bug report's title and description) and checked if none of the duplicates (among the set of previously-reported reports for each query [70]) were retrieved in the top-N results, for N={5, 6, 7, ..., 30}.

Lucene fails at retrieving duplicates for 50.6%, 43.9%, 40.2%, 37.5%, 35.5%, and 33.9% of the queries, within the top-5, -10, -15, -20, -25, and -30 results, respectively (see Table II). These numbers mean that a large number of queries require reformulation (between 14,3k and 21,3k queries, for N=30 and 5, respectively). More sophisticated duplicate detectors may achieve better retrieval [11, 49, 60, 80, 85], but there is still a large percentage of queries that fail to retrieve duplicates.

TABLE II
NUMBER AND PROPORTION OF QUERIES FOR WHICH LUCENE FAILS TO RETRIEVE THE DUPLICATE REPORTS WITHIN THE TOP-N RESULTS.

Project	Top-5	Top-10	Top-15	Top-20
Accumulo	31 (39.2%)	26 (32.9%)	22 (27.8%)	21 (26.6%)
Ambari	25 (21.4%)	21 (17.9%)	21 (17.9%)	19 (16.2%)
ActiveMQ	54 (45.0%)	44 (36.7%)	42 (35.0%)	37 (30.8%)
Cassandra	194 (58.1%)	176 (52.7%)	164 (49.1%)	154 (46.1%)
Cordova	106 (42.9%)	90 (36.4%)	82 (33.2%)	76 (30.8%)
Continuum	31 (34.8%)	28 (31.5%)	27 (30.3%)	24 (27.0%)
Drill	102 (54.8%)	90 (48.4%)	81 (43.5%)	72 (38.7%)
Eclipse	14,7k (51.5%)	12,9k (45.2%)	11,8k (41.5%)	11,1k (38.9%)
Groovy	48 (41.7%)	46 (40.0%)	41 (35.7%)	38 (33.0%)
Hadoop	79 (34.2%)	68 (29.4%)	63 (27.3%)	58 (25.1%)
Hbase	38 (39.2%)	33 (34.0%)	27 (27.8%)	26 (26.8%)
Hive	160 (50.8%)	131 (41.6%)	123 (39.0%)	113 (35.9%)
Maven	156 (56.7%)	131 (47.6%)	117 (42.5%)	106 (38.5%)
M. Firefox	3,7k (49.3%)	3,2k (42.2%)	2,9k (38.4%)	2,7k (35.7%)
MyFaces	38 (48.1%)	30 (38.0%)	27 (34.2%)	25 (31.6%)
OpenOffice	1,5k (48.3%)	1,2k (39.6%)	1,1k (35.4%)	968 (32.1%)
PDFBox	84 (54.2%)	74 (47.7%)	71 (45.8%)	63 (40.6%)
Spark	212 (49.3%)	170 (39.5%)	151 (35.1%)	138 (32.1%)
Wicket	69 (46.3%)	61 (40.9%)	57 (38.3%)	47 (31.5%)
Struts	35 (40.7%)	28 (32.6%)	24 (27.9%)	24 (27.9%)
Total	**21,3k (50.6%)**	**18,5k (43.9%)**	**17k (40.2%)**	**15,8k (37.5%)**

All Top-N results, for N={5, 6, ..., 30}, are available in our replication package [21].

E. Observed Behavior Identification

In order to answer our research questions, we need to identify the terms corresponding to the system's observed behavior (OB) for the bug reports that require reformulation (*i.e.*, for the *low-quality* queries), just as a potential user would do. In contrast, the bug title is found in a separate field within the bug report and its identification is trivial.

We randomly sampled 749 bug reports for which Lucene fails to retrieve duplicates within the top-5 results (see Table III). The sample includes reports for each project, and excludes reports referring to new features and enhancements (*i.e.*, non-bugs) – see our replication package for the full list of manually excluded reports [21]. The amount of sampled reports represents 3.5% of the 21,346 *low-quality* queries for N=5. The query set also contains a subsample of the queries that fail to retrieve the duplicates in top-N for N={6, ..., 30}. Having a relatively small percentage of reports in our overall sample comes from the large query set for Eclipse, Firefox, and OpenOffice. Excluding these projects, the sample would represent 30% of the *low-quality* queries for N=5.

Three of the authors of this paper (*a.k.a.* coders) conducted qualitative text *coding* [77] on all 749 bug reports. The reports were distributed among the coders in such a way that each report was coded by one coder. The coders had to select any part of the text (*e.g.*, words, clauses, or sentences) in the reports' title and description that corresponded to the OB. This task was performed using the text annotation tool BRAT [2].

TABLE III
NUMBER OF SAMPLED/CODED BUG REPORTS (BRS), THE ONES WITH OB, AND NUMBER OF REDUCED QUERIES.

Project	# of coded BRs[a]	# of BRs with OB[b]	# of reduced queries
Accumulo	27 (87.1%)	25 (92.6%)	25
Ambari	7 (28.0%)	7 (100%)	7
ActiveMQ	35 (64.8%)	35 (100%)	35
Cassandra	18 (9.3%)	18 (100%)	18
Cordova	29 (27.4%)	29 (100%)	29
Continuum	29 (93.5%)	29 (100%)	29
Drill	34 (33.3%)	34 (100%)	34
Eclipse	96 (0.7%)	93 (96.9%)	93
Groovy	32 (66.7%)	31 (96.9%)	31
Hadoop	26 (32.9%)	22 (84.6%)	22
Hbase	27 (71.1%)	27 (100%)	27
Hive	17 (10.6%)	17 (100%)	17
Maven	35 (22.4%)	32 (91.4%)	32
M. Firefox	102 (2.7%)	99 (97.1%)	99
MyFaces	31 (81.6%)	30 (96.8%)	30
OpenOffice	113 (7.8%)	111 (98.2%)	110
PDFBox	16 (19.0%)	15 (93.8%)	15
Spark	14 (6.6%)	13 (92.9%)	13
Wicket	32 (46.4%)	32 (100%)	32
Struts	29 (82.9%)	28 (96.6%)	28
Total	**749 (3.5%)**	**727 (97.1%)**	**726**

[a]Proportions with respect to the total # of *low-quality* queries for each project (for N=5). [b]Proportions with respect to the total # of coded BRs.

We summarize the main criteria used by the coders to tag the OB in the bug report title and description (the full list can

be found in our replication package [21]):

- The coding focused only on natural language content written by the users, ignoring code snippets, stack traces, or program logs. However, the natural language referencing this information may indicate OB and was allowed for coding. An example of this case is: "*When I click the File menu, I get the following error and stack trace: ...*".
- Internal system behavior, described by the reporters, was also allowed for coding, for example: "*The open() method in the class FileMenu reads the options from the file...*".
- Descriptions of graphical user interface issues can be considered as OB, for example: *"The menu's color is too light, it should be darker"*.
- Uninformative sentences, such as "*The File menu does not work*" are insufficient to be considered OB. There must be a clear description of the software's OB, for example: "*The File menu doesn't open when I click on it*".
- Explanations of code attached to the bug reports are not considered OB, for example: *"The attached code defines the openMenu() method, which iterates on the options..."*.

Overall, 727 (*i.e.*, 97.1%) of the tagged bug reports describe an OB (see Table III). The OB coding required significant manual effort for all 749 reports, however, in an actual usage scenario, a user only needs to select the OB terms from a single report, which takes seconds.

Figures 1 and 2 show examples of coded bug report descriptions, where their OB part is highlighted in yellow. In practical scenarios, the user would only select the highlighted text and use it for reformulation (*i.e.*, in the case of OB-DD). Note that the OB may be described in non-contiguous parts of the text, including in parts of the title.

We made the choice of having one coder for each bug report in order to maximize the number of queries used in the evaluation. However, our past research that uses multiple coders per bug report reveals high agreement between coders (*i.e.*, +80% *kappa* [19]), hence, we expect minimal differences between single- and multiple-coders-based coding. In any case, our future work will investigate and assess the robustness of OB-DD and OBT-DD with respect to these differences.

F. Query Reformulation Strategies and Measures

We reformulate each one of the *initial queries*, by using each reformulation strategy, as follows:

- For OB-DD, we concatenate the parts of the text corresponding to OB only, from the bug report title and description, and remove the rest of the textual description.
- For BT-DD, we consider the bug title only and remove the rest of the textual description.
- For OBT-DD, we retain the title as is, and the OB parts from the bug report, while removing the rest of the textual description.

We reformulated all 727 bug reports that describe an OB. We call these queries *reduced queries*. However, after preprocessing, one OpenOffice query (from bug report #113872) became empty because its OB contained stop words, numbers, and special characters only. Hence, in total, we obtained 726 reduced queries (see Table III). For these 726 queries, we also used BT-DD and OBT-DD to produce reduced queries. For each initial query, we generated *three* reduced queries.

Bug report title:
Heading lines do not show in MsWord export

Bug report description:
When exporting a document to MsWord, in my case in order to submit it to the LuLu website, the 'Heading Lines' ('Righe d'intestazione' in the Italian version I use), so painstakingly inserted, do not show in the Word export.

Fig. 1. Bug report #104924 for OpenOffice. The highlighted text corresponds to the observed behavior (OB).

Bug report title:
Security Wizard: Delete ATS call should be non-blocking

Bug report description:
Before Start All Services web-client triggers Delete ATS call. This call should not be marked with jquery ajax async flag as false. Doing so makes the call blocking and page becomes unresponsive untill call returns

Fig. 2. Bug report #4968 for Ambari. The highlighted text corresponds to the observed behavior (OB).

We executed each one of the 726 *initial* and *reduced queries* with the Lucene-based duplicate detector. We measured the performance of duplicate retrieval using *Recall Rate@N*, *i.e.*, the proportion of queries for which a DD approach returns at least one duplicate report within the top-N candidates. This is one of the most commonly used measures in past duplicate detection research (see Sec. II) and is ideal for assessing the performance of DD techniques as, in practice, users would likely inspect no more than top-N results before stopping the DD search (*i.e.*, when they find a duplicate or they are confident enough that there are no duplicates). For the sake of completeness, we also measured the performance of the strategies based on MRR and MAP, which are other common metrics used in DD research (see Sec. II). However, for space limitations, we omit the MRR/MAP results in this paper. Our replication package contains the full MRR/MAP results [21].

As mentioned before, our empirical evaluation mimics an actual usage scenario, where the user issues the initial query and inspects the N returned bug reports (step #1). If she does not find a duplicate, then she makes a choice whether to retrieve additional N bug reports with the same query or use any strategy to reformulate the query (*i.e.*, OB-DD, BT-DD, or OBT-DD) and then retrieve additional N bug reports (step #2). In both cases (reformulation and no reformulation in step #2), the N bug reports originally returned in top N (in step #1) are removed from the ranked list and then *Recall Rate@N* is computed for the initial query and the reformulated query as well. We repeat this process for all queries, for N={5, 6, ..., 30}. In the end, if the *Recall Rate@N* for the reformulated queries is higher than the one for the initial queries, we can conclude that the reformulation is the better

TABLE IV
NUMBER AND PROPORTION OF QUERIES (*i.e.*, RECALL RATE@N) FOR WHICH LUCENE RETRIEVES AT LEAST ONE DUPLICATE REPORT WITHIN THE TOP-N RESULTS USING EACH ONE OF THE REFORMULATION STRATEGIES (REFORMULATION) VS. NO REFORMULATION.

N	# of queries	No reformulation	Reformulation			Improvement		
			OB-DD	BT-DD	OB-DD	OB-DD	BT-DD	OB-DD
5	726	90 (12.4%)	139 (19.1%)	125 (17.2%)	150 (20.7%)	54.4%	38.9%	66.7%
10	636	92 (14.5%)	129 (20.3%)	130 (20.4%)	151 (23.7%)	40.2%	41.3%	64.1%
15	594	92 (15.5%)	137 (23.1%)	141 (23.7%)	154 (25.9%)	48.9%	53.3%	67.4%
20	544	78 (14.3%)	135 (24.8%)	137 (25.2%)	153 (28.1%)	73.1%	75.6%	96.2%
25	528	92 (17.4%)	135 (25.6%)	147 (27.8%)	155 (29.4%)	46.7%	59.8%	68.5%
30	502	87 (17.3%)	136 (27.1%)	141 (28.1%)	155 (30.9%)	56.3%	62.1%	78.2%
Avg.*	**583**	**87 (15.1%)**	**136 (23.6%)**	**139 (24.1%)**	**155 (26.8%)**	**56.6%**	**59.6%**	**78.0%**

* Average values across the 26 thresholds (*i.e.*, N={5, 6, ..., 30}).

strategy. Conversely, if the measures are the other way around, we can conclude that it is not worth reformulating the query, as there is no gain over just simply investigating N more results returned by the initial query.

We measured the magnitude of improvement for the *Recall Rate@N* measurements, by computing the change percentage of the metric before (M_b) and after reformulation (M_a), *i.e.*, $\text{Improvement}(M) = (M_a - M_b)/M_b$. We aim at maximizing the improvement, avoiding negative values, which would mean deterioration rather than improvement.

We assessed the statistical significance of our measures using the Mann-Whitney test [41], a paired non-parametric test that does not assume normal distributions (as in our case). This method was used to test if a measure M, when applying a reformulation strategy (M_a), is higher than when using no reformulation (M_b). We carried out the test on the paired *Recall Rate@N* values that we collected across the 26 thresholds. We define the null hypothesis as $H_0 : M_b \geq M_a$, and the alternative hypothesis as $H_1 : M_b < M_a$. We applied the test with a 95% confidence level, thus rejecting the null hypothesis, in favor of the alternative, if *p*-value < 5%.

We also estimated the magnitude of the difference between *Recall Rate@N* measures, by using Cliff's Delta (*d*), a non-parametric measure of the *effect size* for ordinal data that does not assume normal distributions [24, 34]. We applied this measure with a 95% confidence level. The *effect size* (*i.e.*, *Recall Rate@N* difference) is interpreted as *negligible* if $|d| < 0.147$, *small* if $0.147 \leq |d| < 0.33$, *medium* if $0.33 \leq |d| < 0.474$, and *large* if $|d| \geq 0.474$ [34, 73].

IV. EVALUATION RESULTS AND DISCUSSION

We present and discuss the evaluation results for the three query reformulation strategies. For space limitations, we present the results for N={5, 10, 15, 20, 15, 30} only and focus our analysis on Recall Rate@N. The results for the full threshold set, including the MRR/MAP measurements, are available in our online replication package [21].

A. Duplicate Detection Performance

Table IV shows the Recall Rate@N (RR@N) results for the initial and reduced queries when using OB-DD, BT-DD, and OBT-DD. Remember that we remove the original top-N results returned by the initial queries and compare the ability of both the initial and reduced queries on retrieving the duplicates within the next top-N results. This means that for each N, we have a different number of queries (see Table IV). Note that we compare the three strategies on the same set of bug reports used for creating the initial and reduced queries.

When using the initial queries (*i.e.*, no reformulation), Lucene retrieves the duplicates in top N for 15.1% of the queries, on average. When reformulating the queries via OB-DD, Lucene retrieves at least one duplicate report for 23.6% of the queries, on average. This means that 8.5% more queries (on average) return duplicates when using OB-DD, compared to no reformulation, which represents a 56.6% improvement. When reducing the queries based on BT (*i.e.*, BT-DD), Lucene retrieves at least one duplicate for 24.1% of the queries, on average, which represents 9% more queries with retrieved duplicates (*i.e.*, 59.6% improvement). The highest performance is achieved when combining both the OB and BT for reducing the queries, as this strategy leads to 26.8% of the queries (on average) to retrieve duplicates in top N, *i.e.*, 11.7% more queries or 78% improvement, on average, with respect to no reformulation. The three reformulation strategies achieve RR@N improvement (and 63.4%-317.9% MRR/MAP improvement, see our replication package [21]), compared to no reformulation, for each of the 26 thresholds N. The RR@N achieved by all three strategies is statistically significant higher than the one achieved by the initial queries, across the 26 thresholds N (Mann-Whitney, *p*-value < 5%). All strategies achieve a *large* RR@N improvement according to our *effect size* analysis based on Cliff's delta ($|d| \geq 0.474$) – see our replication package for the full statistical test results [21].

We answer our first research question (**RQ1**) positively, as the RR@N indicates that using all reformulation strategies (*i.e.*, OB-DD, BT-DD, and OBT-DD) lead to the retrieval of more duplicates bug reports than with no reformulation. As for our second research question (**RQ2**), we found that OBT-DD achieves a statistically significant higher RR@N compared to OB-DD and BT-DD (Mann-Whitney, *p*-value < 5%), and the RR@N improvement is *large* ($|d| = 0.536$) and *medium* ($|d| = 0.411$), respectively. We also found that BT-DD's RR@N is statistically significant higher than OB-DD's RR@N (Mann-Whitney, *p*-value < 5%). However, this RR@N improvement is *negligible* ($|d| = 0.129$). The results indicate that OB-DD and BT-DD achieve comparable performance. We conclude that OBT-DD leads to the best detection performance, in terms of number of queries with retrieved duplicates, compared to using OB-DD and BT-DD.

The results also reveal an interesting issue. We expected

that Lucene would retrieve more duplicates (with or without reformulation) as N increases. As we report in Table IV, Lucene retrieves almost the same number of duplicates across thresholds N. To better understand this phenomenon and the detection improvements, we analyze the results in more detail.

B. Trade-offs between Successful and Unsuccessful Queries

During query reformulation, there is always a trade-off: some queries become *successful* while others become *unsuccessful*. A good reformulation strategy would lead to more *successful* queries (*i.e.*, retrieve duplicates) than *unsuccessful* queries (*i.e.*, fail to retrieve duplicates), compared to the initial ones. All three reformulation strategies achieve that, but we aim to understand better the trade-offs.

We refer to all the queries that retrieve duplicates in top N as *successful queries*, and to those that do not retrieve duplicates as *unsuccessful queries*. The ideal reformulation strategy would preserve the *successful queries* (*i.e.*, an initial *successful* query, which reformulated remains *successful*, *a.k.a. successful* $\rightarrow$ *successful*), while converting all (or at least some) of the *unsuccessful* initial queries into *successful* ones (*i.e.*, *unsuccessful* $\rightarrow$ *successful*). In other words, we want to avoid the situation when *successful* queries turn *unsuccessful* (*i.e.*, *successful* $\rightarrow$ *unsuccessful*) via the reformulation.

Table V shows that OB-DD and BT-DD transform about the same number of *successful* queries into *unsuccessful* ones, on average (*i.e.*, approx. 45 and 48, out of 87 *good* queries, respectively). However, BT-DD converts more *unsuccessful* queries into *successful* ones (on average), compared to OB-DD (*i.e.*, 99 vs. 93 queries, respectively). OB-DD preserves more of the *successful* queries as *successful* than BT-DD does, on average (*i.e.*, 43 vs 39, respectively), while preserving more of the *unsuccessful* queries as *unsuccessful* (*i.e.*, 403 vs 397, respectively), which is undesirable. These results support BT-DD's higher performance against OB-DD. As for OBT-DD, Table V reveals that it outperforms the other two strategies in all the aspects. Specifically, 39 of 87 *successful* queries turn *unsuccessful*, while 107 *unsuccessful* queries become *successful*. OBT-DD preserves 48 of the *successful* queries as *successful*, and 390 of the *unsuccessful* queries as *unsuccessful*, on average.

The *successful* $\rightarrow$ *unsuccessful* cases need further analysis. Being query reduction strategies, the assumption is that OB-, BT-, and OBT-DD eliminate parts of the bug report (different than BT and OB) that should not be removed. Identifying these parts helps us improve the strategies in the future.

C. Analysis of Successful $\rightarrow$ Unsuccessful Queries

We analyzed the set of *successful* queries that turned *unsuccessful* for N=5 when using the reformulation strategies. This set corresponds to 55 unique queries/reports. From this set, 30 queries (*i.e.*, 54.5%) do not retrieve duplicates with any of the reformulation strategies, which means that the strategies removed important information from the bug report, leading to non-retrieved duplicates. Thirteen queries (*i.e.*, 23.6%) retrieve duplicates only with OB-DD and OBT-DD, which means that

TABLE V
NUMBER AND PROPORTION OF SUCCESSFUL (**S**) AND UNSUCCESSFUL (**U**) QUERIES BEFORE REFORMULATION THAT TURNED UNSUCCESSFUL (**U**) AND SUCCESSFUL (**S**) AFTER REFORMULATION.

N	U → S	S → U	S → S	U → U
5	89 (12.3%)	40 (5.5%)	50 (6.9%)	547 (75.3%)
10	88 (13.8%)	51 (8.0%)	41 (6.4%)	456 (71.7%)
15	92 (15.5%)	47 (7.9%)	45 (7.6%)	410 (69.0%)
20	96 (17.6%)	39 (7.2%)	39 (7.2%)	370 (68.0%)
25	93 (17.6%)	50 (9.5%)	42 (8.0%)	343 (65.0%)
30	97 (19.3%)	48 (9.6%)	39 (7.8%)	318 (63.3%)
Avg.*	**93 (16.3%)**	**45 (7.7%)**	**43 (7.4%)**	**403 (68.7%)**

(a) OB-DD

N	U → S	S → U	S → S	U → U
5	80 (11.0%)	45 (6.2%)	45 (6.2%)	556 (76.6%)
10	96 (15.1%)	58 (9.1%)	34 (5.3%)	448 (70.4%)
15	101 (17.0%)	52 (8.8%)	40 (6.7%)	401 (67.5%)
20	100 (18.4%)	41 (7.5%)	37 (6.8%)	366 (67.3%)
25	104 (19.7%)	49 (9.3%)	43 (8.1%)	332 (62.9%)
30	101 (20.1%)	47 (9.4%)	40 (8.0%)	314 (62.5%)
Avg.*	**99 (17.3%)**	**48 (8.3%)**	**39 (6.8%)**	**397 (67.6%)**

(b) BT-DD

N	U → S	S → U	S → S	U → U
5	100 (13.8%)	40 (5.5%)	50 (6.9%)	536 (73.8%)
10	103 (16.2%)	44 (6.9%)	48 (7.5%)	441 (69.3%)
15	100 (16.8%)	38 (6.4%)	54 (9.1%)	402 (67.7%)
20	109 (20.0%)	34 (6.3%)	44 (8.1%)	357 (65.6%)
25	108 (20.5%)	45 (8.5%)	47 (8.9%)	328 (62.1%)
30	111 (22.1%)	43 (8.6%)	44 (8.8%)	304 (60.6%)
Avg.*	**107 (18.5%)**	**39 (6.8%)**	**48 (8.3%)**	**390 (66.4%)**

* Average values across the 26 thresholds (*i.e.*, N={5, 6, ..., 30}).
Percentage values with respect to the # of queries for each N, from Table IV.

(c) OBT-DD

for these cases, the OB is required for better duplicate retrieval. Eight (*i.e.*, 14.5%) retrieve duplicates when using BT-DD only, two (*i.e.*, 4.6%) when using BT-DD and OBT-DD, and two more (*i.e.*, 4.6%) when using OB-DD only.

We manually analyzed the 30 queries that fail to retrieve duplicates with any of the strategies. Among these, we found that the main source of important terms for retrieval (besides OB and BT) is the steps to reproduce the bug (*i.e.*, S2R).

For 12 *successful* queries, key S2R terms from the bug report were removed by the reformulation strategies. These terms are also present in the duplicate reports, hence, removing them from the initial query leads to non-retrieved duplicates. For example, the Firefox bug report #619430 [6] duplicates the report #611991 [5]. In the former one (*i.e.*, the query), the OB is described by the sentence *"font size menu drop down is not displayed"*, which is repeated three times in the bug description. The terms *font*, *size*, and *menu* are relevant because they appear in the duplicate report. However, phrases such as *"Under preferences"* or *"under content"*, present only in S2R from report #619430 and shared with the duplicate report, were removed, which led to not retrieving the duplicate.

Overall, we expected S2R terms to be common among bugs. For example, *"open a file*" may be a necessary step for replicating many bugs. However, we observed that the S2R often contains terms that better indicate what the bug is about and includes key terms for retrieval. We also hypothesized that

the bugs that can be reproduced in more than one way, would benefit from removing the distinct S2R in the reports, while preserving the OB. However, we did not find any of these cases in the analyzed reports.

For several of the analyzed queries, we found that sources, such as code snippets (in eight bug reports), stack traces (in six reports), and the software's expected behavior (in three reports), contained relevant terms that should not be removed by the reformulation. We hypothesize that code snippets are good candidates to be preserved in the reformulated queries. Our future work will investigate ways of leveraging code snippets in combination with the BT and OB/S2R descriptions for reformulating queries.

Finally, in seven of the analyzed queries, we also observed that the frequency of some terms, present in the OB and/or BT and relevant for retrieval, decreases when reformulating the query with the strategies. Given that Lucene's algorithm is based on term frequencies, we conjecture that increasing the frequency of these terms may lead to better retrieval. In the future, we will increase the frequency of OB/BT terms based on their entire document frequency.

V. Threats to Validity

The main threat to the *construct validity* of the evaluation is the subjectivity introduced in the coded set of bug reports, as each bug report was coded by a single coder. We made this choice to maximize the number of coded bug reports. Since our past research revealed high agreement (*i.e.*, +80% *kappa* [19]) when each report is coded by multiple coders, we expect minimal differences in the results when using multiple coders.

In order to mitigate threats to the *conclusion validity*, we compared the performance of the initial and reduced queries using Recall Rate@N and MRR/MAP, metrics widely used in DD research [18, 43]. We also analyzed the trade-offs of the reformulation strategies, to further strengthen our conclusions. As in prior research [19, 36], we defined two query categories (*i.e.*, *successful* and *unsuccessful*) and analyzed the query transitions between categories before and after reformulation.

The *internal validity* of our evaluation is mainly affected by our selected data set. We built such a data set based on two existing ones, previously used in duplicate detection research [20, 80]. This data set contains bug reports with duplicates, used as queries, with their set of duplicate reports in the corresponding duplicate buckets. While these buckets were extracted from the projects' issue tracker, this does not guarantee that all reports in a bucket are duplicates, either because of incorrect references in the issue tracker or the way we created the buckets (see Sec. III-C). To mitigate this issue, we recreated the queries from the bug reports used in the SDS data set by Sun *et al.* [80]. We identified inconsistencies between the data set we constructed and SDS, based on the same bug reports. We manually checked a subset of the inconsistencies in favor of our data set. Given the small size of the inconsistent buckets, compared to the size of SDS, we believe that the impact on the results is minimal. As for the CDS data set by Chaparro *et al.* [19], we used it as is.

We addressed the *external validity* of our evaluation by using a large set of bug reports from the issue trackers of 20 open source projects that span different domains and software types. We used one ranking-based duplicate detector, based on Lucene. In the future, we will investigate how the reformulation strategies help other DD approaches on retrieving more duplicates. Those techniques that use BT as a separate feature may benefit less from BT-DD, but we believe that OB-DD and OBT-DD would still lead to better retrieval.

VI. Conclusions and Future Work

We argue that duplicate bug report detection approaches, based on text retrieval, should be viewed as a two-step process. The process should allow for query reformulation, which may lead to the retrieval of more duplicates or higher user confidence when none are retrieved. The challenge is that any reformulation strategy should be tool agnostic and independent of the user's knowledge.

We hypothesized that the description of the software's observed behavior (*i.e.*, OB) in bug reports and the bug report title (BT) contain relevant information that is bug specific, while other parts of the description include less specific information (*i.e.*, noise) with respect to duplicate retrieval. Based on this observation, we defined three query reformulation techniques that are based on the user selecting the OB part of the bug description and/or the BT. The reformulation strategies are simple, they do not depend on any information outside the bug report, and they can be applied in more than 97% of the cases. Better yet, they retrieve more duplicates (*i.e.*, for 23.6%-26.8% of the cases, on average, equivalent to a 56.6%-78% avg. improvement) than inspecting more candidate reports retrieved by the initial query. We conclude that triagers should use both the BT and OB from the bug report to reformulate the initial *low-quality* query and expect to find duplicates for more cases than without reformulation. The results bear evidence in support of our conjecture about the OB, BT, and the proposed two-step paradigm for duplicate detection, at least when using Lucene as a duplicate bug report detector.

On the other hand, the results also revealed that relevant information is present in the steps to reproduce (*i.e.*, S2R) and code snippets, which may improve duplicate retrieval. This observation guides our future research plans. Specifically, we will evaluate additional reformulation strategies that would combine information from the BT, OB, S2R, and code snippets. Any of such strategies would still meet the main challenge of being independent of the underlying duplicate detector and other information external to the bug report. With that in mind, we plan to also evaluate the proposed reformulation strategies with duplicate detectors more sophisticated than Lucene. Finally, our future work will focus on automatically reducing queries based on specific bug descriptions.

Acknowledgments

This research was supported by the grants CCF-1848608 and CCF-1526118 from the US National Science Foundation.

REFERENCES

[1] "Apache Lucene: https://lucene.apache.org/," accessed on Oct. 10, 2018.

[2] "BRAT: http://brat.nlplab.org/," accessed on Oct. 10, 2018.

[3] "Bugzilla: https://www.bugzilla.org/," accessed on Oct. 10, 2018.

[4] "Eclipse project: https://www.eclipse.org/," accessed on Oct. 10, 2018.

[5] "Firefox bug report #611991: https://bugzilla.mozilla.org/show_bug.cgi?id=611991," accessed on Oct. 10, 2018.

[6] "Firefox bug report #619430: https://bugzilla.mozilla.org/show_bug.cgi?id=619430," accessed on Oct. 10, 2018.

[7] "Jira: https://tinyurl.com/l5c9ctc," accessed on Oct. 10, 2018.

[8] "Lucene's DefaultSimilarity Javadoc - https://tinyurl.com/y84vawy5," accessed on Oct. 8, 2018.

[9] "Lucene's TFIDFSimilarity Javadoc - https://tinyurl.com/ybhqqrqm," accessed on Oct. 8, 2018.

[10] K. Aggarwal, F. Timbers, T. Rutgers, A. Hindle, E. Stroulia, and R. Greiner, "Detecting Duplicate Bug Reports with Software Engineering Domain Knowledge," *Journal of Software: Evolution and Process*, vol. 29, no. 3, 2017.

[11] M. Amoui, N. Kaushik, A. Al-Dabbagh, L. Tahvildari, S. Li, and W. Liu, "Search-Based Duplicate Defect Detection: An Industrial Experience," in *Proceedings of the Working Conference on Mining Software Repositories (MSR'13)*, 2013, pp. 173–182.

[12] B. Bassett and N. A. Kraft, "Structural Information Based Term Weighting in Text Retrieval for Feature Location," in *Proceedings of the International Conference on Program Comprehension (ICPC'13)*, 2013, pp. 133–141.

[13] N. Bettenburg, R. Premraj, T. Zimmermann, and S. Kim, "Duplicate bug reports considered harmful ... really?" in *Proceedings of the International Conference on Software Maintenance (ICSM'08)*, 2008, pp. 337–345.

[14] ——, "Extracting Structural Information from Bug Reports," in *Proceedings of the International Working Conference on Mining Software Repositories (MSR'08)*, 2008, pp. 27–30.

[15] V. Boisselle and B. Adams, "The Impact of Cross-Distribution Bug Duplicates, Empirical Study on Debian and Ubuntu," in *Proceedings of the 15th International Working Conference on Source Code Analysis and Manipulation (SCAM'15)*, 2015, pp. 131–140.

[16] M. Borg, P. Runeson, J. Johansson, and M. V. Mäntylä, "A Replicated Study on Duplicate Detection: Using Apache Lucene to Search Among Android Defects," in *Proceedings of the 8th International Symposium on Empirical Software Engineering and Measurement (ESEM'14)*, 2014, pp. 8:1–8:4.

[17] C. Carpineto and G. Romano, "A Survey of Automatic Query Expansion in Information Retrieval," *Computing Surveys*, vol. 44, no. 1, p. 1, 2012.

[18] Y. C. Cavalcanti, P. A. da Mota Silveira Neto, I. d. C. Machado, T. F. Vale, E. S. de Almeida, and S. R. d. L. Meira, "Challenges and Opportunities for Software Change Request Repositories: A Systematic Mapping Study," *Journal of Software: Evolution and Process*, vol. 26, no. 7, pp. 620–653, 2014.

[19] O. Chaparro, J. M. Florez, and A. Marcus, "Using Observed Behavior to Reformulate Queries during Text Retrieval-based Bug Localization," in *Proceedings of the International Conference on Software Maintenance and Evolution (ICSME'17)*, 2017, pp. 376–387.

[20] ——, "On the Vocabulary Agreement in Software Issue Descriptions," in *Proceedings of the International Conference on Software Maintenance and Evolution (ICSME'16)*, 2016, pp. 448–452.

[21] O. Chaparro, J. M. Florez, U. Singh, and A. Marcus, "Replication package," 2019. [Online]. Available: https://tinyurl.com/y8vylrpg

[22] O. Chaparro, J. Lu, F. Zampetti, L. Moreno, M. Di Penta, A. Marcus, G. Bavota, and V. Ng, "Detecting Missing Information in Bug Descriptions," in *Proceedings of the Joint Meeting on Foundations of Software Engineering (ESEC/FSE'17)*, 2017, 396-407.

[23] O. Chaparro and A. Marcus, "On the Reduction of Verbose Queries in Text Retrieval Based Software Maintenance," in *Proceedings of the International Conference on Software Engineering (ICSE'16)*, 2016, pp. 716–718.

[24] N. Cliff, *Ordinal methods for behavioral data analysis*. Psychology Press, 2014.

[25] J. L. Davidson, N. Mohan, and C. Jensen, "Coping with Duplicate Bug Reports in Free/Open Source Software Projects," in *Proceedings of the Symposium on Visual Languages and Human-Centric Computing (VL/HCC'11)*, 2011, pp. 101–108.

[26] S. Davies and M. Roper, "What's in a Bug Report?" in *Proceedings of the International Symposium on Empirical Software Engineering and Measurement (ESEM'14)*, 2014, pp. 26:1–26:10.

[27] A. De Lucia, A. Marcus, R. Oliveto, and D. Poshyvanyk, "Information Retrieval Methods for Automated Traceability Recovery," in *Software and Systems Traceability*, J. Cleland-Huang, O. Gotel, and A. Zisman, Eds. Springer, 2012, pp. 71–98.

[28] T. Dietrich, J. Cleland-Huang, and Y. Shin, "Learning Effective Query Transformations for Enhanced Requirements Trace Retrieval," in *Proceedings of the International Conference on Automated Software Engineering (ASE'13)*, 2013, pp. 586–591.

[29] T. Dilshener, M. Wermelinger, and Y. Yu, "Locating Bugs Without Looking Back," in *Proceedings of the International Conference on Mining Software Repositories (MSR'16)*, 2016, pp. 286–290.

[30] B. Dit, M. Revelle, M. Gethers, and D. Poshyvanyk, "Feature Location in Source code: A Taxonomy and Survey," *Journal of Software: Evolution and Process*, vol. 25, no. 1, pp. 53–95, 2012.

[31] G. Gay, S. Haiduc, A. Marcus, and T. Menzies, "On the Use of Relevance Feedback in IR-based Concept Location," in *Proceedings of the International Conference on Software Maintenance (ICSM'09)*, 2009, pp. 351–360.

[32] X. Ge, D. C. Shepherd, K. Damevski, and E. Murphy-Hill, "Design and Evaluation of a Multi-recommendation System for Local Code Search," *Journal of Visual Languages & Computing*, 2016.

[33] M. Gibiec, A. Czauderna, and J. Cleland-Huang, "Towards Mining Replacement Queries for Hard-to-retrieve Traces," in *Proceedings of the International Conference on Automated Software Engineering (ASE'10)*, 2010, pp. 245–254.

[34] R. J. Grissom and J. J. Kim, *Effect sizes for research: A broad practical approach.* Lawrence Erlbaum Associates Publishers, 2005.

[35] J. Guo, M. Gibiec, and J. Cleland-Huang, "Tackling the term-mismatch problem in automated trace retrieval," *Empirical Software Engineering*, vol. 22, no. 3, pp. 1103–1142, 2017.

[36] S. Haiduc, G. Bavota, A. Marcus, R. Oliveto, A. De Lucia, and T. Menzies, "Automatic Query Reformulations for Text Retrieval in Software Engineering," in *Proceedings of the International Conference on Software Engineering (ICSE'13)*, 2013, pp. 842–851.

[37] E. Hatcher and O. Gospodnetic, *Lucene in Action*. Manning Publications, 2004.

[38] E. Hill, M. Roldan-Vega, J. A. Fails, and G. Mallet, "NL-Based Query Refinement and Contextualized Code Search Results: A User Study," in *Proceedings of the Conference on Software Maintenance, Reengineering, and Reverse Engineering (CSMR-WCRE'14)*, 2014, pp. 34–43.

[39] A. Hindle, A. Alipour, and E. Stroulia, "A Contextual Approach towards More Accurate Duplicate Bug Report Detection and Ranking," *Empirical Software Engineering*, vol. 21, no. 2, pp. 368–410, 2016.

[40] A. Hindle and C. Onuczko, "Preventing duplicate bug reports by continuously querying bug reports," *Empirical Software Engineering*, pp. 1–35, 2018.

[41] M. Hollander, D. A. Wolfe, and E. Chicken, *Nonparametric statistical methods*. John Wiley & Sons, 2013, vol. 751.

[42] S. Just, R. Premraj, and T. Zimmermann, "Towards the next generation of bug tracking systems," in *Proceedings of the Symposium on Visual Languages and Human-Centric Computing (VL/HCC'18)*, 2008, pp. 82–85.

[43] L. Kang, "Automated Duplicate Bug Reports Detection - An Experiment at Axis Communication AB," Master's thesis, 2017.

[44] K. Kevic and T. Fritz, "Automatic Search Term Identification for Change Tasks," in *Proceedings of the International Conference on Software Engineering (ICSE'14)*, 2014, pp. 468–471.

[45] N. Klein, C. S. Corley, and N. A. Kraft, "New Features for Duplicate Bug Detection," in *Proceedings of the 11th Working Conference on Mining Software Repositories (MSR'14)*. ACM, 2014, pp. 324–327.

[46] A. J. Ko, B. A. Myers, and D. H. Chau, "A Linguistic Analysis of How People Describe Software Problems," in *Proceedings of the Symposium on Visual Languages and Human-Centric Computing (VL/HCC'06)*, 2006, pp. 127–134.

[47] A. Lazar, S. Ritchey, and B. Sharif, "Improving the Accuracy of Duplicate Bug Report Detection Using Textual Similarity Measures," in *Proceedings of the 11th Working Conference on Mining Software Repositories (MSR'14)*, 2014, pp. 308–311.

[48] O. A. L. Lemos, A. C. d. Paula, H. Sajnani, and C. V. Lopes, "Can the Use of Types and Query Expansion Help Improve Large-scale Code

Search?" in *Proceedings of the International Working Conference on Source Code Analysis and Manipulation (SCAM'15)*, 2015, pp. 41–50.

[49] J. Lerch and M. Mezini, "Finding Duplicates of Your Yet Unwritten Bug Report," in *Proceedings of the 17th European Conference on Software Maintenance and Reengineering (CSMR'13)*, 2013, pp. 69–78.

[50] B. Li, X. Sun, H. Leung, and S. Zhang, "A survey of code-based change impact analysis techniques," *Software Testing, Verification and Reliability*, vol. 23, no. 8, pp. 613–646, 2013.

[51] Z. Li, T. Wang, Y. Zhang, Y. Zhan, and G. Yin, "Query Reformulation by Leveraging Crowd Wisdom for Scenario-based Software Search," in *Proceedings of the Asia-Pacific Symposium on Internetware (Internetware'16)*, 2016, pp. 36–44.

[52] M.-J. Lin, C.-Z. Yang, C.-Y. Lee, and C.-C. Chen, "Enhancements for Duplication Detection in Bug Reports with Manifold Correlation Features," *Journal of Systems and Software*, vol. 121, pp. 223–233, 2016.

[53] K. Liu, H. B. K. Tan, and H. Zhang, "Has This Bug Been Reported?" in *Proceedings of the 20th Working Conference on Reverse Engineering (WCRE'13)*, 2013, pp. 82–91.

[54] R. Lotufo, Z. Malik, and K. Czarnecki, "Modelling the "Hurried" bug report reading process to summarize bug reports," in *Proceedings of the International Conference on Software Maintenance (ICSM'12)*, 2012, pp. 430–439.

[55] X. A. Lu and R. B. Keefer, "Query Expansion/Reduction and its Impact on Retrieval Effectiveness," *NIST Special Publication*, pp. 231–231, 1995.

[56] F. Lv, H. Zhang, J. g. Lou, S. Wang, D. Zhang, and J. Zhao, "CodeHow: Effective Code Search Based on API Understanding and Extended Boolean Model (E)," in *Proceedings of the International Conference on Automated Software Engineering (ASE'15)*, 2015, pp. 260–270.

[57] A. Marcus, A. Sergeyev, V. Rajlich, and J. I. Maletic, "An Information Retrieval Approach to Concept Location in Source Code," in *Proceedings of the Working Conference on Reverse Engineering (WCRE'04)*, 2004, pp. 214–223.

[58] A. Marcus and S. Haiduc, "Text Retrieval Approaches for Concept Location in Source Code," in *Software Engineering: International Summer Schools, ISSSE 2009-2011, Salerno, Italy. Revised Tutorial Lectures*, ser. Lecture Notes in Computer Science. Springer, 2013, vol. 7171, pp. 126–158.

[59] C. Mills, J. Pantiuchina, E. Parra, G. Bavota, and S. Haiduc, "Are Bug Reports Enough for Text Retrieval-based Bug Localization?" in *Proceedings of the 34th International Conference on Software Maintenance and Evolution (ICSME'18)*, 2018, pp. 410–421.

[60] A. T. Nguyen, T. T. Nguyen, T. N. Nguyen, D. Lo, and C. Sun, "Duplicate Bug Report Detection with a Combination of Information Retrieval and Topic Modeling," in *Proceedings of the 27th International Conference on Automated Software Engineering (ASE'12)*, 2012, pp. 70–79.

[61] L. Nie, H. Jiang, Z. Ren, Z. Sun, and X. Li, "Query Expansion Based on Crowd Knowledge for Code Search," *IEEE Transactions on Services Computing*, vol. 9, no. 5, pp. 771–783, 2016.

[62] A. Panichella, A. D. Lucia, and A. Zaidman, "Adaptive User Feedback for IR-Based Traceability Recovery," in *Proceedings of the 8th International Symposium on Software and Systems Traceability (SST'15)*, 2015, pp. 15–21.

[63] M. F. Porter, "An algorithm for suffix stripping," *Program*, vol. 14, no. 3, pp. 130–137, 1980.

[64] D. Poshyvanyk, A. Marcus, R. Ferenc, and T. Gyimóthy, "Using information retrieval based coupling measures for impact analysis," *Empirical software engineering*, vol. 14, no. 1, pp. 5–32, 2009.

[65] T. Prifti, S. Banerjee, and B. Cukic, "Detecting Bug Duplicate Reports Through Local References," in *Proceedings of the 7th International Conference on Predictive Models in Software Engineering (PROMISE'11)*, 2011, pp. 8:1–8:9.

[66] M. M. Rahman and C. K. Roy, "QUICKAR: Automatic Query Reformulation for Concept Location using Crowdsourced Knowledge," in *Proceedings of the International Conference on Automated Software Engineering (ASE'16)*, 2016, pp. 220–225.

[67] ——, "Improved query reformulation for concept location using CodeRank and document structures," in *Proceedings of the International Conference on Automated Software Engineering (ASE'17)*, 2017, pp. 428–439.

[68] ——, "STRICT: Information Retrieval Based Search Term Identification for Concept Location," in *Proceeding of the Conference on Software Analysis, Evolution, and Reengineering (SANER'17)*, 2017, pp. 79–90.

[69] ——, "Improving IR-Based Bug Localization with Context-Aware Query Reformulation," in *Proceedings of the 26th Joint Meeting on the Foundations of Software Engineering (ESEC/FSE'18)*, 2018, pp. 621–632.

[70] M. S. Rakha, C.-P. Bezemer, and A. E. Hassan, "Revisiting the Performance Evaluation of Automated Approaches for the Retrieval of Duplicate Issue Reports," *IEEE Transactions on Software Engineering*, vol. 44, no. 12, pp. 1245–1268, 2018.

[71] ——, "Revisiting the performance of automated approaches for the retrieval of duplicate reports in issue tracking systems that perform just-in-time duplicate retrieval," *Empirical Software Engineering*, vol. 23, no. 5, pp. 2597–2621, 2018.

[72] M. Roldan-Vega, G. Mallet, E. Hill, and J. A. Fails, "CONQUER: A Tool for NL-Based Query Refinement and Contextualizing Code Search Results." in *Proceedings of the International Conference on Software Maintenance (ICSM'13)*, 2013, pp. 512–515.

[73] J. Romano, J. D. Kromrey, J. Coraggio, and J. Skowronek, "Appropriate statistics for ordinal level data: Should we really be using t-test and Cohen's d for evaluating group differences on the NSSE and other surveys," in *Annual meeting of the Florida Association of Institutional Research*, 2006, pp. 1–33.

[74] P. Runeson, M. Alexandersson, and O. Nyholm, "Detection of Duplicate Defect Reports Using Natural Language Processing," in *Proceedings of the 29th International Conference on Software Engineering (ICSE'07)*, 2007, pp. 499–510.

[75] M. Sadat, A. B. Bener, and A. V. Miranskyy, "Rediscovery Datasets: Connecting Duplicate Reports," in *Proceedings of the 14th International Conference on Mining Software Repositories (MSR'17)*, 2017, pp. 527–530.

[76] G. Salton, A. Wong, and C. S. Yang, "A vector space model for automatic indexing," *Communications of the ACM*, vol. 18, no. 11, pp. 613–620, 1975.

[77] C. B. Seaman, "Qualitative Methods in Empirical Studies of Software Engineering," *IEEE Transactions on Software Engineering*, vol. 25, no. 4, pp. 557–572, 1999.

[78] 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 *Proceedings of the International Conference on Aspect-oriented Software Development (AOSD'07)*, 2007, pp. 212–224.

[79] B. Sisman and A. C. Kak, "Assisting Code Search with Automatic Query Reformulation for Bug Localization," in *Proceedings of the Working Conference on Mining Software Repositories (MSR'13)*, 2013, pp. 309–318.

[80] C. Sun, D. Lo, S. C. Khoo, and J. Jiang, "Towards More Accurate Retrieval of Duplicate Bug Reports," in *In Proceedings of the 26th International Conference on Automated Software Engineering (ASE'11)*, 2011, pp. 253–262.

[81] C. Sun, D. Lo, X. Wang, J. Jiang, and S.-C. Khoo, "A Discriminative Model Approach for Accurate Duplicate Bug Report Retrieval," in *Proceedings of the 32nd International Conference on Software Engineering (ICSE'10)*, 2010, pp. 45–54.

[82] A. Sureka and P. Jalote, "Detecting Duplicate Bug Report Using Character N-Gram-Based Features," in *Proceedings of the Asia Pacific Software Engineering Conference (ASPEC'10)*, 2010, pp. 366–374.

[83] F. Thung, P. S. Kochhar, and D. Lo, "DupFinder: Integrated Tool Support for Duplicate Bug Report Detection," in *Proceedings of the 29th International Conference on Automated Software Engineering (ASE'14)*, 2014, pp. 871–874.

[84] Y. Tian, C. Sun, and D. Lo, "Improved Duplicate Bug Report Identification," in *Proceedings of the 16th European Conference on Software Maintenance and Reengineering (CSMR'12)*, 2012, pp. 385–390.

[85] X. Wang, L. Zhang, T. Xie, J. Anvik, and J. Sun, "An Approach to Detecting Duplicate Bug Reports Using Natural Language and Execution Information," in *Proceedings of the 30th International Conference on Software Engineering (ICSE'08)*, 2008, pp. 461–470.

[86] J. Zhou and H. Zhang, "Learning to Rank Duplicate Bug Reports," in *Proceedings of the 21st International Conference on Information and Knowledge Management (CIKM'18)*, 2012, pp. 852–861.

[87] T. Zimmermann, R. Premraj, N. Bettenburg, S. Just, A. Schroter, and C. Weiss, "What Makes a Good Bug Report?" *IEEE Transactions on Software Engineering*, vol. 36, no. 5, pp. 618–643, 2010.