Improving IR-Based Bug Localization with Context-Aware Query Reformulation

Anonymous Author(s)

ABSTRACT

Recent findings suggest that Information Retrieval (IR)-based bug localization techniques do not perform well if the bug report lacks rich structured information (e.g., relevant program entity names). Conversely, excessive structured information (e.g., stack traces) in the bug report might not always help the automated localization either. In this paper, we propose a novel technique–BLIZZARD– that automatically localizes buggy entities from project source using appropriate query reformulation and effective information retrieval. In particular, our technique determines whether there are excessive program entities or not in a bug report (query), and then applies appropriate reformulations to the query for bug localization. Experiments using 5,139 bug reports show that our technique can localize the buggy source entities with 8%–56% higher Hit@10, 8%–62% higher MAP@10 and 8%–62% higher MRR@10 than the baseline technique. Comparison with the state-of-the-art techniques and their variants report that our technique can improve 19% in MAP@10 and 20% in MRR@10 over the state-of-the-art, and can improve 59% of the noisy queries and 39% of the poor queries.

CCS CONCEPTS

•Software and its engineering → Software verification and validation; Software testing and debugging; *Software defect analysis;* Software maintenance tools;

ACM Reference format:
Anonymous Author(s). 2018. Improving IR-Based Bug Localization with Context-Aware Query Reformulation. In *Proceedings of The ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering, Florida, USA, November 2018 (ESEC/FSE'18),* 11 pages.
DOI: 10.475/123_4

1 INTRODUCTION

Despite numerous attempts for automation [5, 15, 19, 35, 67], software debugging is still largely a manual process which costs a significant amount of development time and efforts [4, 37, 60]. One of the three steps of debugging is the identification of the location of a bug in the source code, i.e., bug localization [37, 55]. Recent bug localization techniques can be classified into two broad families–*spectra based* and *information retrieval (IR) based* [29]. While spectra-based techniques rely on execution traces of a software system, IR-based techniques analyse shared vocabulary between a bug report (i.e., query) and the project source for bug localization [34, 65]. Performances of IR-based techniques are reported to be as good as that of spectra-based techniques, and such performances are achieved using a low cost text analysis [44, 55]. Unfortunately, recent qualitative and empirical studies [43, 55] have reported two major limitations. First, IR-based techniques cannot perform well without the presence of rich structured information (e.g., program entity names pointing to defects) in the bug reports. Second, they also might not perform well with a bug report that contains excessive structured information (e.g., stack traces, Table 1) [55]. One possible explanation of these limitations could be that most of the contemporary IR-based techniques [29, 36, 44, 49, 50, 56, 65] use almost verbatim texts from a bug report as a *query* for bug localization. That is, they do not perform any meaningful modification to the query except a limited natural language pre-processing (e.g., stop word removal, token splitting, stemming). As a result, their query could be either *noisy* due to excessive structured information (e.g., stack traces) or *poor* due to the lack of relevant structured information (e.g., Table 2). One way to overcome the above challenges is to (a) refine the noisy query (e.g., Table 1) using appropriate filters and (b) complement the poor query (e.g., Table 2) with relevant search terms. Existing studies [28, 56, 57, 62] that attempt to complement basic IR-based localization with costly data mining or machine learning alternatives can also equally benefit from such query reformulations.

In this paper, we propose a novel technique –BLIZZARD– that locates software bugs from source code by employing *context-aware query reformulation* and information retrieval. Our technique (1) first determines the quality (i.e., prevalence of structured entities or lack thereof) of a bug report (i.e., query) and classifies it as either *noisy*, *rich* or *poor*, (2) then applies appropriate reformulation to the query, and (3) finally uses the improved query for the bug localization with information retrieval. Unlike earlier approaches [48, 49, 56, 65], it either refines a noisy query or complements a poor query for effective information retrieval. Thus, BLIZZARD has a high potential for improving IR-based bug localization.

Our best performing baseline technique that uses all terms except punctuation marks, stop words and digits from a bug report, returns its first correct result for the noisy query containing stack traces in Table 1 at the 53rd position. On the contrary, our technique refines the same noisy query, and returns the first correct result at the first position of the ranked list which is a significant improvement over the baseline. Similarly, when we use a poor query containing no structured entities such as in Table 2, the baseline technique returns the correct result at the 30th position. On the other hand, our technique improves the same poor query, and returns the result again at the first position. BugLocator [65], one of the high performing and well cited IR-based techniques, returns such results at the 19th and 26th positions respectively for the noisy and poor queries which are far from ideal. Thus, our proposed technique clearly has a high potential for improving the IR-based bug localization.

We evaluate our technique in several different dimensions using four widely used performance metrics and 5,139 bug reports (i.e., queries) from six Java-based subject systems. First, we evaluate

Table 1: A Noisy Bug Report (#31637, eclipse.jdt.debug)

Field	Content
Title	should be able to cast "**null**"

Description	When trying to debug an application the variables tab is empty. Also when I try to inspect or display a variable, I get following error logged in the eclipse log file:
	`java.lang.`**`NullPointerException`**
	`at org.eclipse.jdt.internal.debug.core.`
	`model.`**`JDIValue.toString`**`(JDIValue.java:362)`
	`at org.eclipse.jdt.internal.debug.eval.ast.`
	instructions.**Cast.execute**(Cast.java:88)
	`at org.eclipse.jdt.internal.debug.eval.ast.engine.`
	Interpreter.execute(Interpreter.java:44)
	`at org.eclipse.jdt.internal.debug.eval.ast.engine.`
	.... (8 more)......................................

in terms of the performance metrics, contrast with the baseline, and BLIZZARD localizes bugs with 8%–56% higher accuracy (i.e., Hit@10), 8%–62% higher precision (i.e., MAP@10) and and 8%–62% higher result ranks (i.e., MRR@10) than the baseline (Section 4.3). Second, we compare our technique with three bug localization techniques [49, 57, 65], and our technique can improve 19% in MAP@10 and 20% in MRR@10 over the state-of-the-art [57] (Section 4.4). Third, we also compare our approach with four state-of-the-art query reformulations techniques, and BLIZZARD improves the result ranks of 59% of the noisy queries and 39% of the poor queries which are 22% and 28% higher respectively than that of the state-of-the-art [42] (Section 4.4). By incorporating *report quality aspect* and *query reformulation* into IR-based bug localization, we resolve an important issue which was either not addressed properly or otherwise overlooked by earlier studies, which makes our work *novel.* Thus, the paper makes the following contributions:

- A novel query reformulation technique that filters noise from and adds complementary information to the bug report, and suggests improved queries for bug localization.
- A novel bug localization technique that locates bugs from the project source by employing quality paradigm of bug reports, query reformulation, and information retrieval.
- Comprehensive evaluation of the technique using 5,139 bug reports from six open source systems and validation against seven techniques including the state-of-the-art.
- A working prototype with detailed experimental data for replication and third party reuses.

2 GRAPH-BASED TERM WEIGHTING

Term weighting is a process of determining relative importance of a term within a body of texts (e.g., document). Jones [25] first introduced TF-IDF (i.e., term frequency × inverse document frequency) as a proxy to term importance which had been widely used by information retrieval community for the last couple of decades. Unfortunately, TF-IDF does not consider semantic dependencies among the terms in their importance estimation. Mihalcea and Tarau [31] later proposed TextRank as a proxy of term importance which was adapted from Google's PageRank [11] and was reported to perform better than TF-IDF. In TextRank, a textual document is encoded into a text graph where unique words from the document are denoted as nodes, and meaningful relations among the words are denoted as connecting edges [31]. Such relationships could be

Table 2: A Poor Bug Report (#187316, eclipse.jdt.ui)

Field	Content
Title	[preferences] Mark Occurences Pref Page
Description	There should be a link to the pref page on which you can change the color. Namely: General/Editors/Text Editors/Annotations. It's a pain in the a** to find the pref if you do not know Eclipse's preference structure well.

statistical (e.g., co-occurrence), syntactic (e.g., grammatical modification) or semantic (i.e., conceptual relevance) in nature [10]. In this research, we identify important terms using graph-based term weighting from a bug report that might contain structured elements (e.g., stack traces) and unstructured regular texts.

3 BLIZZARD: PROPOSED TECHNIQUE

Fig. 1 shows the schematic diagram of our proposed technique–BLIZZARD. Furthermore, Algorithm 1 shows the pseudo-code for BLIZZARD. We make use of bug report quality, query reformulation, and information retrieval for localizing bugs in source code from bug reports of any quality as shown in the following sections:

3.1 Bug Report Classification

Since our primary objective with this work is to overcome the challenges posed by the different kinds of information bug reports may contain, we categorize the reports prior to bug localization. In addition to having natural language texts, a bug report typically may contain different structured elements: (1) stack traces (reported active stack frames during the occurrence of a bug, e.g., Table 1), and (2) program elements such as method invocations, package names, and source file names. Having consulted with the relevant literature [8, 9, 55], we classify the bug reports into three board categories (Steps 1, 2a, 2b and 2c, Fig. 1) as follows:

BR$_{\text{ST}}$: *ST* stands for stack traces. If a bug report contains one or more stack traces besides the regular texts or program elements, it is classified into BR$_{ST}$. Since trace entries contain too much structured information, query generated from such a report is generally considered *noisy*. We apply the following regular expression [34] to locate the trace entries from the report content.

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

BR$_{\text{PE}}$: *PE* stands for program elements. If a bug report contains one or more program elements (e.g., method invocations, package names, source file name) but no stack traces in the texts, it is classified into BR$_{PE}$. Queries generated from such report are considered *rich.* We use the appropriate regular expressions [46] to identify the program elements from the texts. For example, we use the following one to identify method invocations.

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

BR$_{\text{NL}}$: *NL* stands for natural language. If a bug report contains neither any program elements nor any stack traces, it is classified into BR$_{NL}$. That is, it contains only unstructured natural language description of the bug. Queries generated from such reports are generally considered *poor* in this work.

We adopt a semi-automated approach in classifying the bug reports (i.e., the queries). Once a bug report is provided, we employ

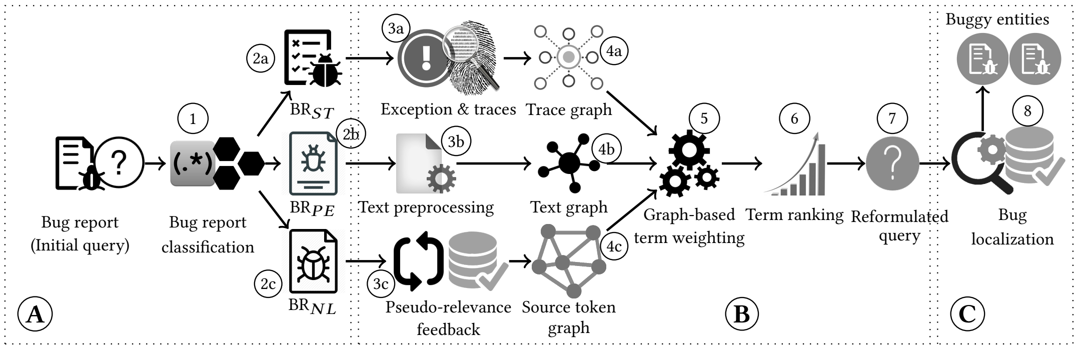

Figure 1: Schematic diagram of the proposed technique: (A) Bug report classification, (B) Query reformulation, and (C) Bug localization

each of our regular expressions to determine its class. If the automated step fails due to ill-defined structures of the report, the class is determined based on manual analysis. Given the explicit nature of the structured entities, human developers can identify the class easily. The contents of each bug report are considered as the *initial queries* which are reformulated in the next few steps.

3.2 Query Reformulation

Once bug reports (i.e., queries) are classified into three classes above based on their structured elements or lack thereof, we apply appropriate reformulations to them. In particular, we analyse either bug report contents or the results retrieved by them, employ graph-based term weighting, and then identify important keywords from them for query reformulation as follows:

Trace Graph Development from BR$_{\mathrm{ST}}$: According to existing findings [43, 55], bug reports containing stack traces are potentially noisy, and performances of the bug localization using such reports (i.e., the queries) are below the average. Hence, important *search keywords* should be *extracted* from the noisy queries for effective bug localization. In this work, we transform the stack traces into a trace graph (e.g., Fig. 2, Steps 3a, 4a, Fig. 1, Lines 8–10, Algorithm 1), and identify the important keywords using a graph-based term weighting algorithm namely PageRank [10, 31].

To the best of our knowledge, to date, graph-based term weighting has been employed only on unstructured regular texts [42] and semi-structured source code [41]. On the contrary, we deal with stack traces which are structured and should be analysed carefully. Stack traces generally comprise of an error message containing the encountered exception(s), and an ordered list of method invocation entries. Each invocation entry can be considered as a tuple $t\{P, C, M\}$ that contains a package name P, a class name C, and a method name M. While these entities are statically connected within a tuple, they are often hierarchically connected (e.g., caller-callee relationships) to other tuples from the traces as well. Hill et al. [22] consider method signatures and field signatures as salient entities from the source code, and suggest code search keywords from them. Similarly, we consider class name and method name from each of the N tuples as the salient items, and represent them as the nodes and their dependencies as the connecting edges in the graph. In stack traces, the topmost entry (i.e., $i = 1$) has the highest degree of interest [16] which gradually decreases for the entries at the lower positions in the list. That is, if $t_i\{P_i, C_i, M_i\}$ is a tuple under analysis, and $t_j\{P_j, C_j, M_j\}$ is a neighbouring tuple with greater degree of interest, then the nodes V_i and edges E_i are added to the trace graph G_{ST} as follows:

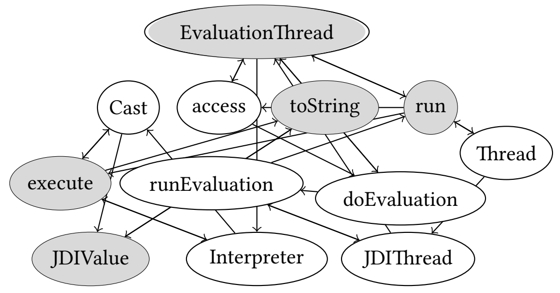

Figure 2: Trace graph of stack traces in Table 1

$$V_i = \{C_i, M_i\},\ E_i = \{C_i \leftrightarrow M_i\} \cup \{C_i \rightarrow C_j, M_i \rightarrow M_j\} \mid j = i - 1$$

$$V = \bigcup_{i=1}^{N}\{V_i\},\ E = \bigcup_{i=1}^{N}\{E_i\},\ G_{ST} = (V, E)$$

For the example stack traces in Table 1, the following connecting edges: `JDIValue`↔`toString`, `Cast`↔`execute`, `Cast`→`JDIValue`, `execute`→`toString`, `Interpreter`↔`execute`, and `Interpreter` →`Cast` are added to the example trace graph in Fig. 2.

Text Graph Development from BR$_{\mathrm{PE}}$: Bug reports containing relevant program entities (e.g., method names) are found effective as queries for IR-based bug localization [43, 49, 55]. However, we believe that *appropriate keyword selection* from such reports can further *boost up* the localization performance. Existing studies employ TextRank and POSRank on natural language texts, and identify search keywords for concept location [42] and information retrieval [10, 31]. Although bug reports (i.e., from BR$_{PE}$) might contain certain structures such as program entity names (e.g., class name, method name) and code snippets besides natural language texts, the existing techniques could still be applied to them given that these structures are treated appropriately. We thus remove stop words [1] and programming keywords [2] from a bug report, *split* the structured tokens using *Samurai* (i.e., a state-of-the-art token splitting tool [17]), and then transform the preprocessed report (R_{pp}) into a set of sentences ($S \in R_{pp}$). We adopt Rahman and Roy [42] that exploits *co-occurrences* and *syntactic dependencies* among the terms for identifying important terms from a textual entity. We thus develop two text graphs (Steps 3b, 4b, Fig. 1, Lines 10–11, Algorithm 1) using co-occurrences and syntactic dependencies among the words from each report as follows:

(1) Text Graph using Word Co-occurrences: In natural language texts, the semantics (i.e., senses) of a given word are often determined by its contexts (i.e., surrounding words) [30, 32, 62]. That is, co-occurring words complement the semantics of each other. We thus consider a sliding window of size K (e.g., $K = 2$) [31], capture co-occurring words, and then encode the word co-occurrences within each window into connecting edges E of a text graph [42]. The individual words ($\forall w_i \in V$) are denoted as nodes in the graph. Thus, for a target word w_i, the following node V_i and two edges E_i will be added to the text graph G_{PE} as follows:

$$V_i = \{w_i\},\ E_i = \{w_i \leftrightarrow w_{i-1}, w_i \leftrightarrow w_{i+1}\} \mid S = [w_1..w_i..w_N]$$

$$V = \bigcup_{\forall S \in R_{pp}} \bigcup_{w_i \in S} \{V_i\},\ E = \bigcup_{\forall S \in R_{pp}} \bigcup_{w_i \in S} \{E_i\},\ G_{PE} = (V, E)$$

Thus, the example phrase– *"source code directory"*–yields two edges, *"source"*↔*"code"* and *"code"*↔*"directory"* while extending the text graph with three distinct nodes– *"source"*, *"code"* and *"directory"*.

(2) Text Graph using POS Dependencies: According to *Jespersen's Rank* theory [10, 24, 42], parts of speech (POS) from a sentence can be divided into three ranks– *primary* (i.e., noun), *secondary* (i.e., verb, adjective) and *tertiary* (i.e., adverb)– where words from a higher rank generally define (i.e., modify) the words from the same or lower ranks. That is, a noun modifies only another noun whereas a verb modifies another noun, verb or an adjective. We determine POS tags using Stanford POS tagger [53], and encode such syntactic dependencies among words into connecting edges and individual words as nodes in a text graph. For example, the sentence annotated using Penn Treebank tags [53]–*"Open$_{VB}$ the$_{DT}$ source$_{NN}$ code$_{NN}$ directory$_{NN}$"*–has the following syntactic dependencies: *"source"*↔*"code"*, *"code"*↔*"directory"*, *"source"*↔*"directory"*, *"open"*←*"source"*, *"open"*←*"code"* and *"open"*←*"directory"*, and thus adds six connecting edges to the text graph.

Source Term Graph Development for BR$_{\text{NL}}$: Bug reports containing only natural language texts and no structured entities are found not effective for IR-based bug localization [43, 55]. We believe that such bug reports possibly miss the right keywords for bug localization. Hence, they need to be *complemented* with *appropriate keywords* before using. A recent study [41] provides improved reformulations to a poor natural language query for concept location by first collecting *pseudo-relevance feedback* and then employing graph-based term weighting. In pseudo-relevance feedback, Top-K result documents, returned by a given query, are naively considered as relevant and hence, are selected for query reformulation [12, 20]. Since bug reports from BR$_{NL}$ class contain only natural language texts, the above study might directly be applicable to them. We thus adopt their approach for our query reformulation, collect Top-K (e.g., $K = 10$) source code documents retrieved by a BR$_{NL}$-based query, and develop a source term graph (Steps 3c, 4c, Fig. 1, Lines 13–15, Algorithm 1).

Hill et al. [22] consider method signatures and fields signatures from source code as the salient items, and suggest keywords for code search from them. In the same vein, we also collect these signatures from each of the K feedback documents for query reformulation. In particular, we extract structured tokens from each signature, split them using *Samurai*, and then generate a natural language

Algorithm 1 IR-Based Bug Localization with Query Reformulation

```
 1: procedure BLIZZARD($R$)                          ▷ $R$: a given bug report
 2:     $Q' \leftarrow \{\}$                          ▷ reformulated query terms
 3:     ▷ Classifying and preprocessing the bug report $R$
 4:     $C_R \leftarrow$ getBugReportClass($R$)
 5:     $R_{pp} \leftarrow$ preprocess ($R$)
 6:     ▷ Representing the bug report as a graph
 7:     switch $C_R$ do
 8:         case BR$_{ST}$
 9:             $ST \leftarrow$ getStackTraces ($R$)
10:             $G_{ST} \leftarrow$ getTraceGraph ($ST$)
11:         case BR$_{PE}$
12:             $G_{PE} \leftarrow$ getTextGraphs ($R_{pp}$)
13:         case BR$_{NL}$
14:             $R_F \leftarrow$ getPseudoRelevanceFeedback ($R_{pp}$)
15:             $G_{NL} \leftarrow$ getSourceTermGraph ($R_F$)
16:     ▷ Getting term weights and search keywords
17:     if ClassKey $CK \in \{ST, PE, NL\}$ then
18:         $PR_{CK} \leftarrow$ getPageRank ($G_{CK}$)
19:         $Q[C_R] \leftarrow$ getTopKTerm(sortByWeight($PR_{CK}$))
20:     end if
21:     ▷ Constructing the reformulated query $Q'$
22:     switch $C_R$ do
23:         case BR$_{ST}$
24:             $N_E \leftarrow$ getExceptionName($R$)
25:             $M_E \leftarrow$ getErrorMessage($R$)
26:             $Q' \leftarrow \{N_E \cup M_E \cup Q[C_R]\}$
27:         case BR$_{PE}$
28:             $Q' \leftarrow Q[C_R]$
29:         case BR$_{NL}$
30:             $Q' \leftarrow \{R_{pp} \cup Q[C_R]\}$
31:     ▷ Bug localization with $Q'$ from codebase $corpus$
32:     return Lucene($corpus$, $Q'$)
33: end procedure
```

phrase from each token [22]. For example, the method signature–`getContextClassLoader()`–can be represented as a verbal phrase–*"get Context Class Loader"*. We then analyze such phrases across all the feedback documents, capture co-occurrences of terms within a fixed window (i.e., $K = 2$) from each phrase, and develop a source term graph. Thus, the above phrase adds four distinct nodes and three connecting edges – *"get"*↔*"context"*, *"context"*↔*"class"* and *"class"*↔*"loader"* – to the source term graph.

Term Weighting using PageRank: Once each body of texts (e.g., stack traces, regular texts, source document) is transformed into a graph, we apply PageRank [11, 31, 41, 42] to the graph for identifying important keywords. PageRank was originally designed for web link analysis, and it determines the reputation of a web page based on the votes or recommendations (i.e., hyperlinks) from other reputed pages on the web [11]. Similarly, in the context of our developed graphs, the algorithm determines importance of a node (i.e., term) based on incoming links from other important nodes of the graph. In particular, it analyses the connectivity (i.e., connected neighbours and their weights) of each term V_i in the graph recursively, and then calculates the node's weight $TW(V_i)$, i.e., term's importance as follows:

$$TW(V_i) = (1 - \phi) + \phi \sum_{j \epsilon In(V_i)} \frac{TW(V_j)}{|Out(V_j)|}\ (0 \leq \phi \leq 1)$$

Here, $In(V_i)$ refers to nodes providing incoming links to V_i, $Out(V_j)$ refers to nodes that V_j is connected to through outgoing links, and ϕ is the damping factor. Brin and Page [11] consider ϕ as the probability of staying on the web page and $1 - \phi$ as the probability of jumping off the page by a random surfer. They use $\phi = 0.85$ which was adopted by later studies [10, 31, 42], and we also do the same. We initialize each node in the graph with a value of 0.25 [31], and recursively calculate their weights unless they converge below a certain threshold (i.e., 0.0001) or the iteration count reaches the maximum (i.e., 100) [31]. Once the calculation is over, we end up with an accumulated weight for each node (Step 5, Fig. 1, Lines 16–20, Algorithm 1). Such weight of a node is considered as an estimation of relative importance of corresponding term among all the terms (i.e., nodes) from the document (i.e., graph).

Reformulation of the Initial Query: Once term weights are calculated, we rank the terms based on their weights, and select the Top-K ($8 \leq K \leq 30$, Fig. 4) terms for query reformulations. Since bug reports (i.e., initial queries) from three classes have different degrees of structured information (or lack thereof), we carefully apply our reformulations to them (Steps 6, 7, Fig. 1, Lines 21–30, Algorithm 1). In case of BR$_{ST}$ (i.e., noisy query), we replace trace entries with the reformulation terms, extract the error message(s) containing exception name(s), and combine them as the reformulated query. For BR$_{NL}$ (i.e., poor query), we combine preprocessed report texts with the highly weighted source terms as the reformulated query. In the case of BR$_{PE}$, only Top-K weighted terms from the bug report are used as a reformulated query for bug localization.

3.3 Bug Localization

Code Search: Once a reformulated query is constructed, we submit the query to *Lucene* [20, 33]. Lucene is a widely adopted search engine for document search that combines Boolean search and VSM-based search methodologies (e.g., TF-IDF [25]). In particular, we employ the Okapi BM25 similarity from the engine, use the reformulated query for the code search, and then collect the results (Step 8, Fig. 1, Lines 31–32, Algorithm 1). These resultant and potentially buggy source code documents are then presented as a ranked list to the developer for manual analysis.

Working Examples: Table 3 shows our reformulated queries for the showcase bug reports in Table 1 (i.e., BR$_{ST}$) and Table 2 (i.e., BR$_{NL}$), and another example report from BR$_{PE}$ class. Baseline queries from these reports return their first correct results at the 53rd (for BR$_{ST}$), 27th (for BR$_{PE}$) and 30th (for BR$_{NL}$) positions of their corresponding ranked lists. On the contrary, BLIZZARD refines the noisy query from BR$_{ST}$ report, selects important keywords from BR$_{PE}$ report, and enriches the poor query from BR$_{NL}$ report by adding complementary terms from relevant source code. As a result, all three reformulated queries return their first correct results (i.e., buggy source files) at the topmost (i.e., first) positions, which demonstrate the potential of our technique for bug localization.

Table 3: Working Examples

Technique	Group	Query Terms	QE
Baseline	BR$_{ST}$	127 terms from Table 1 after preprocessing, **Bug ID# 31637, eclipse.jdt.debug**	53
BLIZZARD		`NullPointerException` + *"Bug should be able to cast* `null`*"* + {`JDIValue toString execute EvaluationThread run`}	**01**
Baseline	BR$_{PE}$	195 terms (after preprocessing) from **Bug ID# 15036, eclipse.jdt.core**	27
BLIZZARD		{astvisitor post postvisit previsit pre file post pre astnode visitor}	**01**
Baseline	BR$_{NL}$	32 terms from Table 2 after preprocessing, **Bug ID# 475855, eclipse.jdt.ui**	30
BLIZZARD		Preprocessed report texts + {`compliance create preference add configuration field dialog annotation`}	**01**

QE = Query Effectiveness, rank of the first returned correct result

4 EXPERIMENT

We evaluate our proposed technique in several different dimensions using four widely used performance metrics and more than 5K bug reports (the queries) from six different subject systems. First, we evaluate in terms of the performance metrics and contrast with the baseline for different classes of bug reports/queries (Section 4.3). Second, we compare our approach with three state-of-the-art bug localization techniques (Section 4.4). Third, and possibly the most importantly, we also compare our approach with four state-of-the-art query reformulations techniques (Section 4.4). In particular, we answer four research questions using our experiments as follows:

- **RQ$_1$:** (a) How does BLIZZARD perform in bug localization, and (b) how do various parameters affect its performance?
- **RQ$_2$:** Do our reformulated queries perform better than the baseline search queries from the bug reports?
- **RQ$_3$:** Can BLIZZARD outperform the existing bug localization techniques including the state-of-the-art?
- **RQ$_4$:** Can BLIZZARD outperform the existing query reformulation techniques targeting concept/feature location and bug localization?

4.1 Experimental Dataset

Data Collection: We collect a total of 5,139 bug reports from six open source subject systems for our experiments. The dataset was taken from an earlier empirical study [43]. Table 4 shows our dataset. First, all the resolved (i.e., marked as RESOLVED) bugs of each subject system from BugZilla and JIRA repositories were collected given that they were submitted within a specific time interval. Then the version control history of each system at GitHub was consulted to identify the bug-fixing commits [6]. Such approach was regularly adopted by the relevant literature [7, 34, 65], and we also follow the same. In order to ensure a fair evaluation, we also discard such reports from our dataset for which no source code files (e.g., Java classes) were changed or no relevant source files exist in the collected system snapshot.

Goldset Development: We collect *changeset* (i.e., list of changed files) from each of our selected bug-fixing commits, and develop a *goldset*. Multiple changesets for the same bug were merged together.

Replication Package: The working prototype of our tool, experimental dataset and other associated materials are made publicly available [3] for replication and third-party reuse.

4.2 Performance Metrics

We use four performance metrics for the evaluation and comparison of our technique. Since these metrics were frequently used by

Table 4: Experimental Dataset

System	Time Period	BR_{ST}	BR_{PE}	BR_{NL}	BR_{All}	System	Time Period	BR_{ST}	BR_{PE}	BR_{NL}	BR_{All}	Total
ecf	Oct, 2001–Jan, 2017	71	319	163	553	eclipse.jdt.ui	Oct, 2001–Jun, 2016	130	578	407	1,115	BR_{ST} = **826** (16.06%)
eclipse.jdt.core	Oct, 2001–Sep, 2016	159	698	132	989	eclipse.pde.ui	Oct, 2001–Jun, 2016	123	239	510	872	BR_{PE} = **2,767** (53.81%)
eclipse.jdt.debug	Oct, 2001–Jan, 2017	126	202	229	557	tomcat70	Sep, 2001–Aug, 2016	217	731	105	1,053	BR_{NL}= **1,546** (30.08%)
Total: 5,139												

BR_{ST}=Bug reports with stack traces, **BR_{PE}**=Bug reports with only program entities, **BR_{NL}**=Bug reports with only natural language texts

the relevant literature [34, 42, 49, 56, 62, 65], they are also highly appropriate for our experiments in this work.

Hit@K: It is defined as the percentage of queries for which at least one buggy file (i.e., from the goldset) is correctly returned within the Top-K results. It is also called Recall@Top-K [49] and Top-K Accuracy [42] in the literature.

Mean Average Precision@K (MAP@K): Unlike regular precision, this metric considers the ranks of correct results within a ranked list. Precision@K calculates precision at the occurrence of each buggy file in the list. Average Precision@K (AP@K) is defined as the average of Precision@K for all the buggy files in a ranked list for a given query. Thus, Mean Average Precision@K is defined as the mean of Average Precision@K (AP@K) of all queries as follows:

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

Here, function $buggy(k)$ determines whether k^{th} file (or result) is buggy (i.e., returns 1) or not (i.e., returns 0), P_k provides the precision at k^{th} result, and D refers to the number of total results. S is the gold set for a query, and Q is the set of all queries. The bigger the MAP@K value is, the better a technique is.

Mean Reciprocal Rank@K (MRR@K): Reciprocal Rank@K is defined as the multiplicative inverse of the rank of first correctly returned buggy file (i.e., from gold set) within the Top-K results. Thus, Mean Reciprocal Rank@K (MRR@K) averages such measures for all queries in the dataset as follows:

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

Here, $firstRank(q)$ provides the rank of first buggy file within a ranked list. MRR@K can take a maximum value of 1 and a minimum value of 0. The bigger the MRR@K value is, the better a bug localization technique is.

Effectiveness (E): It approximates a developer's effort in locating the first buggy file in the result list [34]. That is, the measure returns the rank of first buggy file in the result list. The lower the effectiveness value is, the better a given query is, i.e., the developer needs to check less amount of results from the top before reaching the actual buggy file in the list.

4.3 Experimental Results

We first show the performance of our technique in terms of appropriate metrics (RQ_1-(a)), then discuss the impacts of different adopted parameters upon the performance (RQ_1-(b)), and finally show our comparison with the baseline queries (RQ_2) as follows:

Selection of Baseline Queries, and Establishment of Baseline Technique and Baseline Performance: Existing studies suggest that text retrieval performances could be affected by query quality [20], underlying retrieval engine [33] or even text preprocessing steps [23, 26]. Hence, we choose the baseline queries and baseline technique pragmatically for our experiments. We conduct a detailed study where three independent variables– bug report field (e.g., title, whole texts), retrieval engine (e.g., Lucene [20], Indri [49]) and text preprocessing step (i.e., stemming, no stemming)–are alternated, and then we choose the best performing configuration as the baseline approach. In particular, we chose the preprocessed version (i.e., performed stop word and punctuation removal, split complex tokens but avoided stemming) of the whole texts (i.e., *title + description*) from a bug report as a baseline query. Lucene was selected as the baseline technique since it outperformed Indri on our dataset. The performance of Lucene with the baseline queries was selected as the baseline performance (i.e., Table 5) for IR-based bug localization in this study. In short, our baseline is: (preprocessed whole texts + splitting of complex tokens + Lucerne search engine).

Table 5: Performance of BLIZZARD in Bug Localization

Dataset	Technique	Hit@1	Hit@5	Hit@10	MAP@10	MRR@10
BR_{ST}	Baseline	21.67%	40.03%	48.25%	28.09%	0.29
	BLIZZARD	*__34.42%__	*__66.28%__	*__75.21%__	*__45.50%__	*__0.47__
BR_{PE}	Baseline	39.85%	64.29%	72.09%	47.28%	0.50
	BLIZZARD	**44.31%**	*__69.48%__	*__77.84%__	*__52.08%__	*__0.55__
BR_{NL}	Baseline	28.24%	50.96%	61.23%	35.48%	0.38
	BLIZZARD	**29.89%**	**54.57%**	**66.18%**	*__38.37%__	*__0.41__
All	Baseline	34.32%	57.83%	66.47%	41.66%	0.44
	BLIZZARD	*__38.59%__	*__65.08%__	*__74.52%__	*__47.14%__	*__0.50__

*=Significantly higher than baseline, **Emboldened**= Comparatively higher

Answering RQ_1(a) – Performance of BLIZZARD: As shown in Table 5, on average, our technique–BLIZZARD–localizes 74.52% of the bugs from a dataset of 5,139 bug reports with 47% mean average precision@10 and a mean reciprocal rank@10 of 0.50 which are 12%, 13% and 14% higher respectively than the baseline performance measures. That is, on average, our technique can return the first buggy file at the second position of the ranked list, almost half of returned files are buggy (i.e., true positive) and it succeeds three out of four times in localizing the bugs. Furthermore, while the baseline technique is badly affected by the noisy (i.e., BR_{ST}) and poor queries (i.e., BR_{NL}), our technique overcomes such challenges with appropriate query reformulations, and provides significantly higher performances. For example, the baseline technique can localize 48% of the bugs from BR_{ST} dataset (i.e., noisy queries) with only 28% precision when Top-10 results are considered. On the contrary, our technique localizes 75% of the bugs with 46% precision in the same context which are 56% and 62% higher respectively than the corresponding baseline measures. Such improvements are about 8% for BR_{NL}, i.e., poor queries. In the cases where bug reports contain program entities, i.e., BR_{PE}, and the baseline performance measures are already pretty high, our technique further refines the query and provides even higher performances. For example, BLIZZARD improves both baseline MRR@10 and baseline MAP@10 for BR_{PE} dataset by 10% which is promising.

Fig. 3 further demonstrates the comparative analyses between BLIZZARD and the baseline technique for various Top-K results in terms of (a) precision and (b) reciprocal rank in the bug localization.

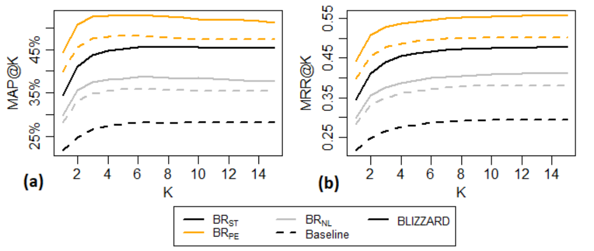

Figure 3: Comparison of BLIZZARD with baseline technique in terms of (a) MAP@K and (b) MRR@K

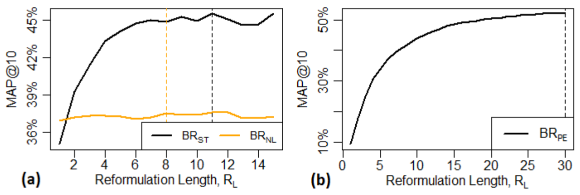

Figure 4: Impact of query reformulation length on performance

Table 6: Query Improvement by BLIZZARD over Baseline Queries

Dataset	Query Pair	Improved/MRD	Worsened/MRD	Preserved
BR$_{ST}$	**BLIZZARD** vs. BL$_T$	484 (**58.60**%)/-82	206 (24.94%)/+34	136 (16.46%)
	BLIZZARD vs. BL	485 (**58.72**%)/-122	174 (21.07%)/+72	167 (20.22%)
BR$_{PE}$	**BLIZZARD** vs. BL$_T$	**1,397** (**50.49**%)/-60	600 (21.68%)/+38	770 (27.83%)
	BLIZZARD vs. BL	865 (31.26%)/-34	616 (22.26%)/+24	1,286 (46.48%)
BR$_{NL}$	**BLIZZARD** vs. BL$_T$	869 (**56.21**%)/-27	355 (22.96%)/+29	322 (20.83%)
	BLIZZARD vs. BL	597 (38.62%)/-16	455 (29.43%)/+31	494 (31.95%)
All	**BLIZZARD** vs. BL$_T$	**2,750** (**53.51**%) /-55	1,161 (22.59%)/+32	1,228 (23.90%)
	BLIZZARD vs. BL	1,947 (37.89%)/-50	1,245 (24.22%)/+30	1,947 (37.89)%

* = Significant difference, **Preserved**=Query quality unchanged, **MRD** = Mean Rank Difference between BLIZZARD and baseline queries, **BL**$_T$ = title, **BL** = title + description

From Fig. 3-(a), we see that precision reaches to the maximum pretty quickly (i.e., at $K \approx 4$) for both techniques. While the baseline technique suffers from noisy (i.e., from BR$_{ST}$) and poor (i.e., from BR$_{NL}$) queries, BLIZZARD achieves significantly higher precision than the baseline. Our non-parametric statistical tests–*Mann-Whitney Wilcoxon* and *Cliff's Delta*–reported *p-values*< 0.05 with an effect size of *large* (i.e., $0.85 \leq \Delta \leq 1.00$). Although the baseline precision for BR$_{PE}$ is higher, BLIZZARD offers even higher precision. From Fig. 3-(b), we see that mean reciprocal ranks of BLIZZARD have a logarithmic shape and whereas the baseline counterparts look comparatively flat. That is, as more Top-K results are considered, more true positives are identified by our technique than the baseline technique does. Statistical tests also reported strong significance (i.e., *p-values*<0.001) and large effect size (i.e., $0.68 \leq \Delta \leq 1.00$) of our measures over the baseline counterparts. That is, BLIZZARD performs a good job in reformulating the noisy and poor queries, and such reformulations contribute to a significant improvement in the bug localization performances.

Answering RQ$_1$(b) –Impact of Method Parameters: We investigate the impacts of different adopted parameters -*query reformulation length*, *word stemming*, and *retrieval engine* - upon our technique, and justify our choices. BLIZZARD reformulates a given query (i.e., bug report) for bug localization, and hence, size of the reformulated query is an important parameter. Fig. 4 demonstrates how various reformulation lengths can affect the MAP@10 of our technique. We see that precision reaches the maximum for three report classes at different query reformulation lengths (i.e., R$_L$). For BR$_{ST}$, we achieve the maximum precision at R$_L$=11, and for BR$_{NL}$, such maximum is detected with R$_L$ ranging between 8 and 12. On the contrary, precision increases in a logarithmic manner for BR$_{PE}$ bug reports. We investigated up to 30 reformulation terms and found the maximum precision. Given the above empirical findings, we chose R$_L$=11 for BR$_{ST}$, R$_L$=30 for BR$_{PE}$ and R$_L$=8 for R$_{NL}$ as the adopted query reformulation lengths and our choices are justified.

We also investigate the impact of stemming and text retrieval engine on our technique. We found that stemming did not improve the performance of BLIZZARD, i.e., reduced localization accuracy. Similar finding was reported by earlier studies as well [23, 26]. We also found that Lucene performs better than Indri on our dataset. Besides, Lucene has been widely used by relevant literature [20, 33, 34, 42]. Given the above findings and earlier suggestions, our choices on stemming and retrieval engine are also justified.

Our technique improves the baseline accuracy, precision and reciprocal rank by 8%–56%, 8%–62% and 8%–62% respectively on average, and our adopted parameters are also justified.

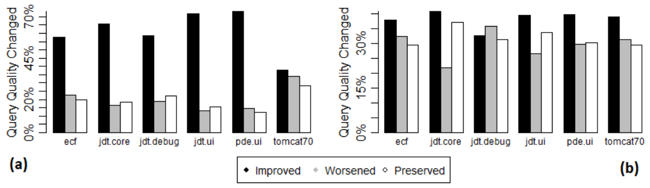

Figure 5: Quality improvement of (a) noisy and (b) poor baseline queries by our technique–BLIZZARD

Answering RQ$_2$-Comparison with Baseline Queries: While Table 5 contrasts BLIZZARD with the baseline approach for only Top-10 results, we further investigate how BLIZZARD performs compared to the baseline when all results returned by each query are considered. We compare our queries with two baseline queries –*title* (i.e., BL$_T$), *title+description* (i.e., BL) – from each of the bug reports. When our query returns the first correct result at a higher position in the result list than that of corresponding baseline query, we call it *query improvement* and vice versa *query worsening*. When result ranks of the reformulated query and the baseline query are the same, then we call it *query preserving*. From Table 6, we see that our applied reformulations improve the noisy (i.e., BR$_{ST}$) and poor (i.e., BR$_{NL}$) queries by 59% and 39%–56% respectively with ≈ 25% worsening ratios. That is, the improvements are more than two times the worsening ratios. Fig. 5 further demonstrates the potential of our reformulations where improvement, worsening and preserving ratios are plotted for each of the six subject systems. We see that noisy queries get benefited greatly from our reformulations, and on average, their query effectiveness improve up to 122 positions (i.e., MRD of BR$_{ST}$, Table 6) in the result list. Such improvement of ranks can definitely help the developers in locating the buggy files in the result list more easily. The poor queries also improve due to our reformulations significantly (i.e., *p-value*=0.004<0.05, *Cliff's* Δ=0.94 (*large*)), and the correct results can be found 16 positions earlier (than the baseline) in the result list starting from the top. Quantile analysis in Table 9 also confirms that noisy and poor queries are significantly improved by our provided reformulations. Besides, the benefits of query reformulations are also demonstrated by our findings in Table 5 and Fig. 3.

Table 7: Comparison with Existing Bug Localization Techniques

RG	Technique	Hit@1	Hit@5	Hit@10	MAP@10	MRR@10
BR_{ST}	BugLocator	28.79%	55.08%	67.00%	38.49%	0.40
	BLUiR	23.38%	44.34%	54.06%	30.96%	0.32
	AmaLgam+$_{BRO}$	45.33%	66.97%	73.29%	52.88%	0.55
	BLIZZARD	34.42%	66.28%	**75.21%**	45.50%	0.47
	BLIZZARD$_{BRO}$	**47.42%**	**73.74%**	**78.77%**	**56.22%**	**0.59**
	AmaLgam+	50.51%	66.47%	71.66%	55.97%	0.58
	BLIZZARD+	**53.39%**	***76.12%**	***80.03%**	**60.65%**	**0.63**
BR_{PE}	BugLocator	36.25%	61.37%	70.96%	44.24%	0.47
	BLUiR	35.54%	62.93%	72.17%	43.67%	0.47
	AmaLgam$_{BRO}$	33.90%	60.48%	69.09%	42.00%	0.45
	BLIZZARD	***44.31%**	***69.48%**	**77.84%**	***52.08%**	***0.55**
	BLIZZARD$_{BRO}$	**47.16%**	**71.26%**	**78.25%**	**53.69%**	**0.57**
	Amalgam+	52.00%	68.54%	72.93%	55.80%	0.59
	BLIZZARD+	**56.84%**	**74.70%**	**80.09%**	**60.78%**	**0.65**
BR_{NL}	BugLocator	25.11%	48.52%	59.04%	32.19%	0.35
	BLUiR	29.87%	56.63%	66.10%	38.07%	0.41
	AmaLgam+$_{BRO}$	29.40%	56.07%	65.01%	37.74%	0.40
	BLIZZARD	**29.89%**	54.57%	**66.18%**	**38.37%**	**0.41**
	BLIZZARD$_{BRO}$	**35.45%**	**58.75%**	**69.17%**	**42.26%**	**0.46**
	AmaLgam+	49.72%	65.42%	71.49%	52.74%	0.57
	BLIZZARD+	47.97%	**66.24%**	**74.49%**	52.12%	0.56
All	BugLocator	31.85%	57.37%	67.87%	40.17%	0.43
	BLUiR	32.45%	59.18%	68.65%	40.82%	0.44
	Amalgam+$_{BRO}$	35.03%	61.32%	69.89%	43.36%	0.46
	BLIZZARD	**38.59%**	**65.08%**	**74.52%**	**47.14%**	***0.50**
	BLIZZARD$_{BRO}$	**44.26%**	**69.15%**	**76.61%**	**51.41%**	***0.55**
	AmaLgam+	52.29%	68.53%	73.58%	56.03%	0.59
	BLIZZARD+	**54.78%**	**73.76%**	**79.66%**	**59.32%**	**0.63**

RG=Report Group, **BRO**=Bug Report Only, *=Significantly higher

Table 8: Components behind Existing IR-Based Bug Localization

Technique	Bug Report Only				External Resources			MRR
	BRT	BRS	ST	QR	BRH	VCH	AH	
Baseline	●							0.44
BugLocator	●				●			0.43
BLUiR	●	●						0.44
AmaLgam+$_{BRO}$	●	●	●					0.46
BLIZZARD	●			●				***0.50**
BLIZZARD$_{BRO}$	●	●	●	●				***0.55**
AmaLgam+	●	●	●		●	●	●	0.59
BLIZZARD+	●	●	●	●	●	●	●	**0.63**

BRT=Bug Report Texts, **BRS**=Bug Report Structures, **ST**=Stack Traces, **QR**=Query Reformulation, **BRH**=Bug Report History, **VCH**=Version Control History, **AH**=Authoring History, **BRO**=Bug Report Only, ●=Feature used

Our applied reformulations to the bug localization queries improve 59% of the noisy queries and 39%–56% of the poor queries, and return the buggy files closer to the top of result list. Such improvements can reduce a developer's effort in locating bugs.

4.4 Comparison with Existing Techniques

Answering RQ3 –Comparison with Existing IR-Based Bug Localization Techniques: Our evaluation of BLIZZARD with four widely used performance metrics shows promising results. The comparison with the best performing baseline shows that our approach outperforms the baselines. However, in order to further gain confidence and to place our work in the literature, we also compared our approach with three IR-based bug localization techniques [49, 57, 65] including the state-of-the-art [57]. Zhou et al. [65] first employ improved Vector Space Model (i.e., rVSM) and bug report similarity for locating buggy source files for a new bug report. Saha et al. [49] employ structured information retrieval where (1) a bug report is divided into two fields–*title*, *description* and a source document is divided into four fields–*class*, *method*, *variable* and *comments*, and then (2) eight similarity measures between these two groups are accumulated to rank the source document. We collect authors' implementations of both techniques for our experiments.

While the above studies use bug report contents only, the later approaches combine them [48] and add more internal [59] or external information sources such as version control history [56] and author information [57]. In the same vein, Wang and Lo [57] recently combine five internal and external information sources - similar bug report, structured IR, stack traces, version control history and document authoring history – for ranking a source document, and outperform five earlier approaches which makes it the state-of-the-art in IR-based bug localization. Given that authors' implementation is not publicly available, we implement this technique ourselves by consulting with the original authors. Since BLIZZARD does not incorporate any external information sources, to ensure a fair comparison, we also implement a variant of the state-of-the-art namely AmaLgam+$_{BRO}$. It combines bug report texts, structured IR and stack traces (i.e., Table 8) for source document ranking.

From Table 7, we see that AmaLgam+ performs the best among the existing techniques. However, its performance comes at a high cost of mining six information contents (i.e., Table 8). Besides, for optimal performance, AmaLgam+ needs past bug reports, version control history and author history which might always not be available. Thus, to ensure a fair comparison, we develop two variants of our technique–BLIZZARD$_{BRO}$ and BLIZZARD+. BLIZZARD$_{BRO}$ combines query reformulation with bug report only features whereas BLIZZARD+ combines query reformulation with all ranking components of AmaLgam+ (i.e., details in Table 8). We then compare both BLIZZARD and BLIZZARD$_{BRO}$ with AmaLgam+$_{BRO}$, and BLIZZARD+ with AmaLgam+.

As shown in Table 7, BLIZZARD outperforms AmaLgam+$_{BRO}$ in terms of all three metrics especially for BR_{PE} reports while performing moderately high with other report groups. For example, BLIZZARD provides 22% higher MRR@10 and 24% higher MAP@10 than AmaLgam+$_{BRO}$ for BR_{PE}. When all report only features are complemented with appropriate query reformulations, our technique, BLIZZARD$_{BRO}$ outperforms AmaLgam+$_{BRO}$ in terms of all three metrics–Hit@K, MAP@10 and MRR@10– with each report groups. Such findings suggest that BLIZZARD$_{BRO}$ can better exploit the available resources (i.e., bug report contents) than the state-of-the-art variant, and returns the buggy files at relatively higher positions in the ranked list. Furthermore, BLIZZARD+ outperforms the state-of-the-art, AmaLgam+, by introducing query reformulation paradigm. For example, BLIZZARD+ improves Hit@5 and Hit@10 over AmaLgam+ for each of the three query types, e.g., 15% and 12% respectively for noisy queries (BR_{ST}). It also should be noted that none of the existing techniques is robust to all three report groups simultaneously. We overcome such issue with appropriate query reformulations, and deliver ≈75%–80% Hit@10 irrespective of the bug report quality. From Table 8, we see that BLIZZARD$_{PRO}$ provides 20% higher MRR@10 than AmaLgam+$_{BRO}$ by consuming equal amount of resources, i.e., bug report only. All these findings above suggest two important points. First, earlier studies might have failed to exploit the report contents and structures properly for bug localization. Second, query reformulation has a high potential for improving the IR-based bug localization.

Table 9: Comparison of Query Effectiveness with Existing Techniques

Technique	RG	Improvement							Worsening							Preserving
		#Improved	Mean	Q1	Q2	Q3	Min.	Max.	#Worsened	Mean	Q1	Q2	Q3	Min.	Max.	#Preserved
Rocchio [47]	BR_{ST} (826)	337 (40.80%)	68	4	12	60	1	1,245	264 (31.96%)	118	6	21	97	2	2,824	225 (27.24%)
RSV [20]		218 (26.39%)	163	10	43	158	1	2,103	236 (28.57%)	198	17	71	245	2	2,487	372 (45.04%)
Sisman and Kak [51]		339 (41.04%)	66	4	12	53	1	1,245	265 (32.08%)	121	7	23	100	2	2,846	222 (26.88%)
STRICT [42]		399 (48.30%)	35	1	4	17	1	1,538	318 (38.50%)	139	6	25	110	2	3,066	109 (13.20%)
Baseline			153	7	35	149	2	2,221		70	1	5	30	1	2,469	
BLIZZARD		**485 (58.72%)**	**22**	**1**	**3**	**9**	**1**	**932**	**174 (21.07%)**	**112**	**4**	**15**	**60**	**2**	3,258	**167 (20.22%)**
Rocchio [47]	BR_{NL} (1,546)	32 (2.07%)	33	4	8	19	1	365	24 (1.55%)	140	4	12	146	2	850	1,490 (96.38%)
RSV [20]		345 (22.27%)	112	3	9	38	1	6,564	751 (48.57%)	105	7	23	81	2	2,140	450 (29.11%)
Sisman and Kak [51]		499 (32.28%)	59	2	6	26	1	2,019	575 (37.19%)	98	5	15	64	2	2,204	472 (30.47%)
STRICT [42]		467 (30.21%)	57	2	6	30	1	1,213	654 (42.30%)	112	5	18	63	2	4,933	425 (27.44%)
Baseline			91	5	15	57	2	2,434		61	2	8	30	1	1,894	
BLIZZARD		**597 (38.62%)**	**75**	**2**	**8**	**32**	**1**	3,063	**455 (29.43%)**	**92**	**5**	**15**	**54**	**2**	2,024	**494 (31.95%)**

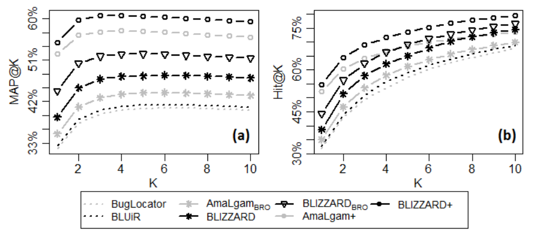

Figure 6: Comparison of (a) MAP@K and (b) Hit@K with the state-of-the-art bug localization techniques

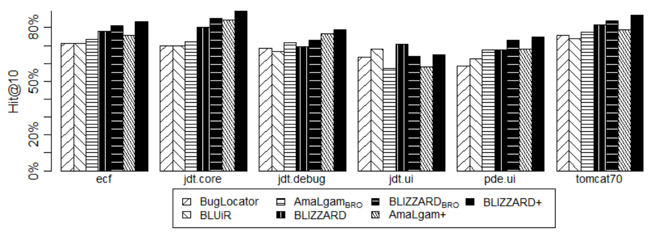

Figure 7: Comparison of Hit@10 across all subject systems

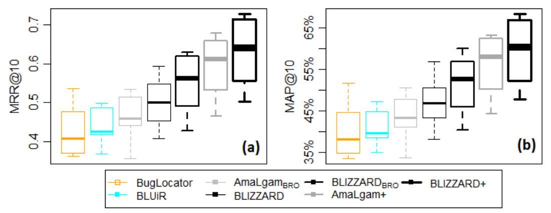

Figure 8: Comparison of (a) MRR@10 and (b) MAP@10 with existing techniques across subject systems

Fig. 6 demonstrates a comparison of BLIZZARD with the existing techniques in terms of (a) MAP@K and (b) Hit@K for various Top-K results. Our statistical tests report that BLIZZARD, $BLIZZARD_{BRO}$ and BLIZZARD+ outperform $AmaLgam+_{BRO}$ and AmaLgam+ respectively in MAP@K by a significant margin (i.e., *p-values*≤0.001) and large effect size (i.e., 0.82≤Δ≤1.00). Similar findings were achieved for Hit@K measures as well.

Fig. 7 and Fig. 8 focus on subject system specific performances. From Fig. 7, we see that BLIZZARD outperforms $AmaLgam+_{BRO}$ with four systems in Hit@10, and falls short with two systems. However, $BLIZZARD_{BRO}$ and BLIZZARD+ outperform $AmaLgam+_{BRO}$ and AmaLgam+ respectively for all six systems. As shown in the box plots of Fig. 8, BLIZZARD has a higher median in MRR@10 and MAP@10 than $AmaLgam+_{BRO}$ across all subject systems. AmaLgam+ improves both measures especially MAP@10. However, BLIZZARD+ provides even higher MRR@10 and MAP@10 than any of the existing techniques including the state-of-the-art.

Our technique outperforms the state-of-the-art from IR-based bug localization in various dimensions. It offers 20% higher precision and reciprocal rank than that of state-of-the-art variant (i.e., $AmaLgam+_{BRO}$) by using only query reformulation rather than costly alternatives, e.g., mining of version control history

Answering RQ$_4$ –Comparison with Existing Query Reformulation Techniques: While we have already showed that our approach outperforms the baselines and the state-of-the-art IR-based bug localization approaches, we also wanted to further evaluate our approach in the context of query reformulation. We thus compared BLIZZARD with four query reformulation techniques [20, 42, 47, 51] including the state-of-the-art [42] that were mostly used for concept/feature location. We use authors' implementation of the state-of-the-art, STRICT, and re-implement the remaining three techniques. We collect Query Effectiveness (i.e., rank of the first correct result) of each of the reformulated queries provided by each technique, and compare with ours using quantile analysis. From Table 9, we see that 48% of the noisy (i.e., BR_{ST}) queries are improved by STRICT, and 32% of the poor (i.e., BR_{NL}) queries are improved by Sisman and Kak [51]. Neither of these techniques considers bug report quality (i.e., prevalence of structured information or lack thereof) and each technique applies the same reformulation strategy to all reports. On the contrary, BLIZZARD chooses appropriate reformulation based on the class of a bug report, and improves 59% of the noisy queries and 39% of the poor queries which are 22% and 20% higher respectively. When compared using quantile analysis, we see that our quantiles are highly promising compared to the baseline. Our reformulations clearly improve the noisy queries, and 75% of the improved queries return their first correct results within Top-9 (i.e., Q_3=9) positions whereas STRICT needs Top-17 positions for the same. In the case of poor queries, quantiles of BLIZZARD are comparable to that of Sisman and Kak. However, BLIZZARD worsens less and preserves higher amount of the baseline queries which demonstrate its high potential.

BLIZZARD outperforms the state-of-the-art in query reformulation using context-aware (i.e., responsive to report quality) query reformulation. Whatever improvements are offered to noisy and poor queries by the state-of-the-art, our technique improves 22% more of noisy queries and 20% more of the poor queries.

5 THREATS TO VALIDITY

Threats to internal validity relate to experimental errors and biases [64]. Replication of existing studies and misclassification of the bug reports are possible sources of such threats. We use authors' implementation of three techniques [42, 49, 65] and re-implement the remaining four. While we cannot rule out the possibility of any implementation errors, we re-implemented them by consulting with the original authors [57] and their reported settings and parameters [20, 47, 51]. While our technique employs appropriate regular expressions for bug report classification, they are limited in certain contexts (e.g., ill-structured stack traces) which require limited manual analysis currently. More sophisticated classification approaches [38, 52, 66] could be applied in the future work.

Threats to external validity relate to generalizability of a technique [64]. We conduct experiments using Java systems. However, since we deal with mostly structured items (e.g., stack traces, program entities) from a bug report, our technique can be adapted to other OOP-based systems that have structured items.

6 RELATED WORK

Bug Localization: Automated bug localization has been an active research area for over two decades [49]. Existing studies from the literature can be roughly categorized into two broad families–*spectra* based and *information retrieval (IR)* based [29, 55]. We deal with IR-based bug localization in this work. Given that spectra based techniques are costly and lack scalability [34, 55], several studies adopt IR-based methods such as Latent Semantic Indexing (LSI) [39], Latent Dirichlet Allocation (LDA) [36, 44] and Vector Space Model (VSM) [27, 34, 49, 50, 59, 65] for bug localization. They leverage the shared vocabulary between bug reports and source code entities for bug localization. Unfortunately, as existing evidences [43, 55] suggest, they are inherently subject to the quality of bug reports. A number of recent studies complement traditional IR-based localization with spectra based analysis [29], machine learning [28, 61] and mining of various repositories– bug report history [48], version control history [50, 56], code change history [58, 63] and document authoring history [57]. Recently, Wang and Lo [57] combine bug report contents and three external repositories, and outperform six earlier IR-based bug localization techniques [48–50, 56, 59, 65] which makes it the state-of-the-art. In short, the contemporary studies advocate for combining (1) multiple localization approaches (e.g., dynamic trace analysis [29], Deep learning [28], learning to rank [61, 62]) and (2) multiple external information sources with classic IR-based localization, and thus, improve the localization performances. However, such solutions could be costly (i.e., multiple repository mining) and less scalable (i.e., dependency on external information sources), and hence, could be infeasible to use in practice. In this work, we approach the problem differently, and focus on better leveraging the potential of the resources at hand (i.e., bug report and source code) which might have been *underestimated* by the earlier studies. In particular, we refine the noisy queries (i.e., containing stack traces) and complement the poor queries (i.e., lacks structured items), and offer an effective information retrieval unlike the earlier studies. Thus, issues raised by low quality bug reports have been significantly addressed by our technique, and our experimental findings support such conjecture. We compare with three existing studies including the state-of-the-art [57], and the detailed comparison can be found in Section 4.4 (i.e., RQ$_3$).

A few studies [34, 59] analyse stack traces from a bug report for bug localization. However, they apply the trace entries to boost up source document ranking, and superfluous trace entries were not discarded from their stack traces. Learning-to-rank [61, 62] and Deep learning [28] based approaches might also suffer from noisy and poor queries since they adopt classic IR without query reformulation in their document ranking. Recent studies [54, 62] employ distributional semantics of words to address limitations of VSM. Since noisy terms in the report could be an issue, our approach can complement these approaches through query reformulation.

Query Reformulation: There exist several studies [14, 18, 20–22, 26, 40, 42, 45, 62] that support concept/feature/concern location tasks using query reformulation. However, these approaches mostly deal with unstructured natural language texts. Thus, they might not perform well with bug reports containing excessive structured information (e.g., stack traces), and our experimental findings also support this conjecture (Table 9). Sisman and Kak [51] first introduce query reformulation in the context of IR-based bug localization. However, their approach cannot remove noise from a query. Recently, Chaparro et al. [13] identify observed behaviour (OB), expected behaviour (EB) and steps to reproduce (S2R) from a bug report, and then use OB texts as a reformulated query for bug localization. However, they only analyze unstructured texts whereas we deal with both structured and unstructured contents. Since we apply query reformulation, we compare with four recent query reformulation techniques employed for concept location–Rocchio [47], RSV [20], STRICT [42] [41] and bug localization– SCP [51]. The detailed comparison can be found in Section 4.4 (i.e., RQ$_4$).

In short, existing IR-based techniques suffer from *quality issues* of bug reports whereas traditional query reformulation techniques are *not well-adapted* for the bug reports containing excessive structured information (e.g., stack traces). Our work fills this *gap* of the literature by incorporating context-aware (i.e., report quality aware) query reformulation into the IR-based bug localization. Our technique better exploits resources at hand and delivers equal or higher performance than the state-of-the-art at a relatively lower cost. To the best of our knowledge, such comprehensive solution was not provided by any of the existing studies.

7 CONCLUSION AND FUTURE WORK

Traditional IR-based bug localization is inherently subject to the quality of submitted bug reports. In this paper, we propose a novel technique that leverages the quality aspect of bug reports, incorporates context-aware query reformulation into the bug localization, and thus, overcomes the above limitation. Experiments using 5,139 bug reports from six open source systems report that BLIZZARD can offer up to 62% and 20% higher precision than the best baseline technique and the state-of-the-art respectively. Our technique also improves 22% more of noisy queries and 20% more of the poor queries than that of state-of-the-art. In future, we plan to apply our learned insights and our technique to further complex activities during debugging such as automatic bug fixing.

REFERENCES

[1] 2011. Stop words. (2011). https://code.google.com/p/stop-words Accessed: June 2017.
[2] 2015. Java keywords. (2015). https://docs.oracle.com/javase/tutorial/java/nutsandbolts/_keywords.html Accessed: June 2017.
[3] 2018. BLIZZARD: Replication package. (2018). https://goo.gl/2zb3e9
[4] J. Anvik, L. Hiew, and G. C. Murphy. 2006. Who Should Fix This Bug?. In *Proc. ICSE*. 361–370.
[5] B. Ashok, J. Joy, H. Liang, S. K. Rajamani, G. Srinivasa, and V. Vangala. 2009. DebugAdvisor: A Recommender System for Debugging. In *Proc. ESEC/FSE*. 373–382.
[6] A. Bachmann and A. Bernstein. 2009. Software Process Data Quality and Characteristics: A Historical View on Open and Closed Source Projects. In *Proc. IWPSE*. 119–128.
[7] B. Bassett and N. A. Kraft. 2013. Structural information based term weighting in text retrieval for feature location. In *Proc. ICPC*. 133–141.
[8] N. Bettenburg, S. Just, A. Schröter, C. Weiss, R. Premraj, and T. Zimmermann. 2008. What Makes a Good Bug Report?. In *Proc. FSE*. 308–318.
[9] N. Bettenburg, R. Premraj, T. Zimmermann, and S. Kim. 2008. Extracting Structural Information from Bug Reports. In *Proc. MSR*. 27–30.
[10] R. Blanco and C. Lioma. 2012. Graph-based Term Weighting for Information Retrieval. *Inf. Retr.* 15, 1 (2012), 54–92.
[11] S. Brin and L. Page. 1998. The Anatomy of a Large-Scale Hypertextual Web Search Engine. *Comput. Netw. ISDN Syst.* 30, 1-7 (1998), 107–117.
[12] C. Carpineto and G. Romano. 2012. A Survey of Automatic Query Expansion in Information Retrieval. *ACM Comput. Surv.* 44, 1 (2012), 1:1–1:50.
[13] O. Chaparro, J. M. Florez, and A Marcus. 2017. Using Observed Behavior to Reformulate Queries during Text Retrieval-based Bug Localization. In *Proc. ICSME*. to appear.
[14] O. Chaparro and A. Marcus. 2016. On the Reduction of Verbose Queries in Text Retrieval Based Software Maintenance. In *Proc. ICSE-C*. 716–718.
[15] F. Chen and S. Kim. 2015. Crowd Debugging. In *Proc. ESEC/FSE*. 320–332.
[16] J. Cordeiro, B. Antunes, and P. Gomes. 2012. Context-based Recommendation to Support Problem Solving in Software Development. In *Proc. RSSE*. 85 –89.
[17] E. Enslen, E. Hill, L. Pollock, and K. Vijay-Shanker. 2009. Mining source code to automatically split identifiers for software analysis. In *Proc. MSR*. 71–80.
[18] G. Gay, S. Haiduc, A. Marcus, and T. Menzies. 2009. On the Use of Relevance Feedback in IR-based Concept Location. In *Proc. ICSM*. 351–360.
[19] Z. Gu, E.T. Barr, D. Schleck, and Z. Su. 2012. Reusing Debugging Knowledge via Trace-based Bug Search. In *Proc. OOPSLA*. 927–942.
[20] S. Haiduc, G. Bavota, A. Marcus, R. Oliveto, A. De Lucia, and T. Menzies. 2013. Automatic Query Reformulations for Text Retrieval in Software Engineering. In *Proc. ICSE*. 842–851.
[21] S. Haiduc, G. Bavota, R. Oliveto, A. De Lucia, and A. Marcus. 2012. Automatic Query Performance Assessment During the Retrieval of Software Artifacts. In *Proc. ASE*. 90–99.
[22] E. Hill, L. Pollock, and K. Vijay-Shanker. 2009. Automatically Capturing Source Code Context of NL-queries for Software Maintenance and Reuse. In *Proc. ICSE*. 232–242.
[23] E Hill, S Rao, and A Kak. 2012. On the Use of Stemming for Concern Location and Bug Localization in Java. In *Proc. SCAM*. 184–193.
[24] Otto Jespersen. 1929. The Philosophy of Grammar. (1929).
[25] K S Jones. 1972. A Statistical Interpretation Of Term Specificity And Its Application In Retrieval. *Journal of Documentation* 28, 1 (1972), 11–21.
[26] K. Kevic and T. Fritz. 2014. Automatic Search Term Identification for Change Tasks. In *Proc. ICSE*. 468–471.
[27] D. Kim, Y. Tao, S. Kim, and A. Zeller. 2013. Where Should We Fix This Bug? A Two-Phase Recommendation Model. *TSE* 39, 11 (2013), 1597–1610.
[28] A. N. Lam, A. T. Nguyen, H. A. Nguyen, and T. N. Nguyen. 2017. Bug Localization with Combination of Deep Learning and Information Retrieval. In *Proc. ICPC*. 218–229.
[29] Tien-Duy B. Le, R. J. Oentaryo, and D. Lo. 2015. Information Retrieval and Spectrum Based Bug Localization: Better Together. In *Proc. ESEC/FSE*. 579–590.
[30] R. Mihalcea. 2005. Unsupervised Large-vocabulary Word Sense Disambiguation with Graph-based Algorithms for Sequence Data Labeling. In *Proc. HLT*. 411–418.
[31] R. Mihalcea and P. Tarau. 2004. TextRank: Bringing Order into Texts. In *Proc. EMNLP*. 404–411.
[32] T. Mikolov, I. Sutskever, K. Chen, G. S. Corrado, and J. Dean. 2013. Distributed Representations of Words and Phrases and their Compositionality. In *Proc. NIPS*. 3111–3119.
[33] L. Moreno, G. Bavota, S. Haiduc, M. Di Penta, R. Oliveto, B. Russo, and A. Marcus. 2015. Query-based Configuration of Text Retrieval Solutions for Software Engineering Tasks. In *Proc. ESEC/FSE*. 567–578.
[34] L. Moreno, J. J. Treadway, A. Marcus, and W. Shen. 2014. On the Use of Stack Traces to Improve Text Retrieval-Based Bug Localization. In *Proc. ICSME*. 151–160.
[35] D. Mujumdar, M. Kallenbach, B. Liu, and B. Hartmann. 2011. Crowdsourcing Suggestions to Programming Problems for Dynamic Web Development Languages. In *Proc. CHI*. 1525–1530.
[36] A. T. Nguyen, T. T. Nguyen, J. Al-Kofahi, H. V. Nguyen, and T. N. Nguyen. 2011. A Topic-based Approach for Narrowing the Search Space of Buggy Files from a Bug Report. In *Proc. ASE*. 263–272.
[37] C. Parnin and A. Orso. 2011. Are Automated Debugging Techniques Actually Helping Programmers?. In *Proc. ISSTA*. 199–209.
[38] N. Pingclasai, H. Hata, and K. i. Matsumoto. 2013. Classifying Bug Reports to Bugs and Other Requests Using Topic Modeling. In *Proc. APSEC*, Vol. 2. 13–18.
[39] D. Poshyvanyk, Y. G. Gueheneuc, A. Marcus, G. Antoniol, and V. Rajlich. 2007. Feature Location Using Probabilistic Ranking of Methods Based on Execution Scenarios and Information Retrieval. *TSE* 33, 6 (2007), 420–432.
[40] M. M. Rahman and C. K. Roy. 2016. QUICKAR: Automatic Query Reformulation for Concept Location Using Crowdsourced Knowledge. In *Proc. ASE*. 220–225.
[41] M. M. Rahman and C. K. Roy. 2017. Improved Query Reformulation for Concept Location using CodeRank and Document Structures. In *Proc. PeerJ Preprints*, Vol. 5. https://doi.org/10.7287/peerj.preprints.3186v1
[42] M. M. Rahman and C. K. Roy. 2017. STRICT: Information Retrieval Based Search Term Identification for Concept Location. In *Proc. SANER*. 79–90.
[43] M. M. Rahman and C. K. Roy. 2018. Improving Bug Localization with Report Quality Dynamics and Query Reformulation. In *Proc. ICSE-C*. to appear.
[44] S. Rao and A. Kak. 2011. Retrieval from Software Libraries for Bug Localization: A Comparative Study of Generic and Composite Text Models. In *Proc. MSR*. 43–52.
[45] S. Rastkar, G. C. Murphy, and G. Murray. 2010. Summarizing Software Artifacts: A Case Study of Bug Reports. In *Proc. ICSE*. 505–514.
[46] P. C. Rigby and M.P. Robillard. 2013. Discovering Essential Code Elements in Informal Documentation. In *Proc. ICSE*. 832–841.
[47] J.J. Rocchio. *The SMART Retrieval System—Experiments in Automatic Document Processing*. Prentice-Hall, Inc. 313–323 pages.
[48] R. K. Saha, J. Lawall, S. Khurshid, and D. E. Perry. 2014. On the Effectiveness of Information Retrieval Based Bug Localization for C Programs. In *Proc. ICSME*. 161–170.
[49] R. K. Saha, M. Lease, S. Khurshid, and D. E. Perry. 2013. Improving bug localization using structured information retrieval. In *Proc. ASE*. 345–355.
[50] B. Sisman and A. C. Kak. 2012. Incorporating Version Histories in Information Retrieval Based Bug Localization. In *Proc. MSR*. 50–59.
[51] B. Sisman and A. C. Kak. 2013. Assisting code search with automatic Query Reformulation for bug localization. In *Proc. MSR*. 309–318.
[52] F. Thung, D. Lo, and L. Jiang. 2012. Automatic Defect Categorization. In *Proc. WCRE*. 205–214.
[53] K. Toutanova, D. Klein, C.D. Manning, and Y. Singer. 2003. Feature-Rich Part-of-Speech Tagging with a Cyclic Dependency Network. In *Proc. HLT-NAACL*. 252–259.
[54] Y. Uneno, O. Mizuno, and E. H. Choi. 2016. Using a Distributed Representation of Words in Localizing Relevant Files for Bug Reports. In *Proc. QRS*. 183–190.
[55] Q. Wang, C. Parnin, and A. Orso. 2015. Evaluating the Usefulness of IR-based Fault Localization Techniques. In *Proc. ISSTA*. 1–11.
[56] S. Wang and D. Lo. 2014. Version History, Similar Report, and Structure: Putting Them Together for Improved Bug Localization. In *Proc. ICPC*. 53–63.
[57] S. Wang and D. Lo. 2016. AmaLgam+: Composing Rich Information Sources for Accurate Bug Localization. *JSEP* 28, 10 (2016), 921–942.
[58] M. Wen, R. Wu, and S. C. Cheung. 2016. Locus: Locating bugs from software changes. In *Proc. ASE*. 262–273.
[59] C. P. Wong, Y. Xiong, H. Zhang, D. Hao, L. Zhang, and H. Mei. 2014. Boosting Bug-Report-Oriented Fault Localization with Segmentation and Stack-Trace Analysis. In *Proc. ICSME*. 181–190.
[60] X. Xia, L. Bao, D. Lo, and S. Li. 2016. "Automated Debugging Considered Harmful" Considered Harmful: A User Study Revisiting the Usefulness of Spectra-Based Fault Localization Techniques with Professionals Using Real Bugs from Large Systems. In *Proc. ICSME*. 267–278.
[61] Xin Ye, Razvan Bunescu, and Chang Liu. 2014. Learning to Rank Relevant Files for Bug Reports Using Domain Knowledge. In *Proc. FSE*. 689–699.
[62] X. Ye, H. Shen, X. Ma, R. Bunescu, and C. Liu. 2016. From Word Embeddings to Document Similarities for Improved Information Retrieval in Software Engineering. In *Proc. ICSE*. 404–415.
[63] K. C. Youm, J. Ahn, J. Kim, and E. Lee. 2015. Bug Localization Based on Code Change Histories and Bug Reports. In *Proc. APSEC*. 190–197.
[64] T. Yuan, D. Lo, and J. Lawall. 2014. Automated Construction of a Software-Specific Word Similarity Database. In *Proc. CSMR-WCRE*. 44–53.
[65] J. Zhou, H. Zhang, and D. Lo. 2012. Where should the bugs be fixed? More accurate information retrieval-based bug localization based on bug reports. In *Proc. ICSE*. 14–24.
[66] Y. Zhou, Y. Tong, R. Gu, and H. Gall. 2014. Combining Text Mining and Data Mining for Bug Report Classification. In *Proc. ICSME*. 311–320.
[67] T. Zimmermann, N. Nagappan, and A. Zeller. 2008. Predicting Bugs from History. In *Software Evolution*. Springer, 69–88.