Improved Query Reformulation for Concept Location using CodeRank and Document Structures

Mohammad Masudur Rahman Chanchal K. Roy
Department of Computer Science, University of Saskatchewan, Canada
{masud.rahman, chanchal.roy}@usask.ca

***Abstract*—During software maintenance, developers usually deal with a significant number of software change requests. As a part of this, they often formulate an initial query from the request texts, and then attempt to map the concepts discussed in the request to relevant source code locations in the software system (a.k.a., concept location). Unfortunately, studies suggest that they often perform poorly in choosing the right search terms for a change task. In this paper, we propose a novel technique –ACER– that takes an initial query, identifies appropriate search terms from the source code using a novel term weight –CodeRank, and then suggests effective reformulation to the initial query by exploiting the source document structures, query quality analysis and machine learning. Experiments with 1,675 baseline queries from eight subject systems report that our technique can improve 71% of the baseline queries which is highly promising. Comparison with five closely related existing techniques in query reformulation not only validates our empirical findings but also demonstrates the superiority of our technique.**

***Index Terms*—Query reformulation, CodeRank, term weighting, query quality analysis, concept location, data resampling**

I. Introduction

Studies show that about 80% of the total efforts is spent in software maintenance [36] where developers deal with a significant number of software issues [35, 45, 52]. Software issue reports (a.k.a., change requests) discuss both unexpected (or erroneous features such as bugs) and expected but non-existent features (e.g., new functionality). For both bug resolution and new feature implementation, a developer is required to map the concepts discussed in the issue report to appropriate source code within the project which is widely known as concept location [29, 31, 40]. Developers generally choose one or more important keywords from the report texts, and then use a search method (e.g., regular expression) to locate the source code entities (e.g., classes, methods) that need to be changed. Unfortunately, as the existing studies [28, 30] report, developers regardless of their experience perform poorly in choosing appropriate search terms for software change tasks. According to Kevic and Fritz [28], only 12.20% of the search terms chosen by the developers were able to locate relevant source code entities for the change tasks. Furnas et al. [15] also suggest that there is a little chance (i.e., 10%–15%) that developers guess the exact words used in the source code. One way to assist the developers in this regard is to automatically suggest helpful reformulations (e.g., complementary keywords) to their initially chosen queries.

Existing studies apply relevance feedback from developers [16], pseudo-relevance feedback from information retrieval methods [21], and machine learning [21, 34] for such query reformulation tasks. They also make use of context of query terms from source code [23, 25, 40, 49, 53], text retrieval configuration [21, 34], and quality of queries [19, 20] in suggesting the reformulated queries. Gay et al. [16] capture explicit feedback on document relevance from the developers, and then suggest reformulated queries using Rocchio's expansion [43]. Haiduc et al. and colleagues [18, 19, 20, 21, 22] take quality of a given query (i.e., query difficulty) into consideration, and suggest the best reformulation strategy for the query using machine learning. While all these above techniques are reported to be novel or effective, most of them also share several limitations. First, source documents contain both structured items (e.g., method signatures, formal parameters) and unstructured items (e.g., code comments). Unfortunately, many of the above reformulation approaches [16, 21, 49] treat the source documents as simple plain text documents, and ignore most of their structural aspects except structured tokens. Such inappropriate treatment might lead to suboptimal or poor queries. In fact, Hill et al. [23] first consider document structures, and suggest natural language phrases from method signatures and field signatures for local code search. However, since they apply only simple textual matching between initial queries and the signatures, the suggested phrases are subject to the quality of not only the given queries and but also of the identifier names from those signatures. Second, many of these approaches often directly apply traditional metrics of term importance (e.g., avgIDF [20], TF-IDF [43]) to source code which were originally targeted for unstructured regular texts (e.g., news article) [26]. Thus, they might also fail to identify the appropriate terms from the structured source documents for query reformulation.

In this paper, we propose a novel technique–ACER–for automatic query reformulation for concept location in the context of software change tasks. We (1) first introduce a novel graph-based term weight *–CodeRank–* for identifying important terms from the source code, and then (2) apply that term weight and source document structures (e.g., method signatures) to our technique for automatic query reformulation. *CodeRank* identifies important terms not only by analyzing salient structured entities (e.g., camel case tokens), but also by exploiting the co-occurrences among the terms across various entities. Our technique–ACER–accepts a natural language query as input, develops multiple candidate queries from two different important contexts, (1) method signatures and (2) field signatures of the source documents independently using CodeRank, and then suggests the best reformulation (based

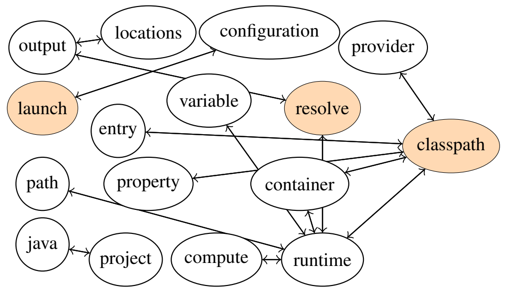

Fig. 1. An example term graph generated by CodeRank for source code of Listing 1

on query quality analysis and machine learning [19, 21]) to the poorly performing initial query.

Table I shows an example change request [2] submitted for `eclipse.jdt.debug` system, and it refers to "debugger source lookup" issue of Eclipse IDE. Let us assume that the developer chooses important keywords from the request title, and formulates a generic initial query–"debugger source lookup." Unfortunately, the query does not perform well, and returns the first correct result at the 79th position of the result list. Further extension–"debugger source lookup work variables"–also does not help, and returns the result at the 77th position. The existing technique – RSV [13]– extends the query as follows–"debugger source lookup work variables *launch configuration jdt java debug"*–where the new terms are collected from the project source using TF-IDF based term weight. This query returns the correct result at the 30th position which is also far from ideal unfortunately. The query of Sisman and Kak [49]–"debugger source lookup work variables *test exception suite core code*"–also returns the correct result at the 51st position. On the other hand, our suggested query–"debugger source lookup work variables *launch debug problem resolve required classpath"*–returns the correct result at the 2nd position which is highly promising. We first collect structured tokens (e.g., `resolveRuntimeClasspathEntry`) from method signatures and field signatures of the source code (e.g., Listing 1), and split them into simpler terms (e.g., `resolve`, `Runtime`, `Classpath` and `Entry`). The underlying idea is that such signatures often encode high level intents and important domain terms while the rest of the code focuses on more granular level implementation details, and thus possibly contains more noise [23, 48]. We develop individual term graph (e.g., Fig. 1) based on term co-occurrences from each signature type, apply CodeRank term weighting, and extract multiple candidate reformulations with the highly weighted terms (e.g., orange coloured, Fig. 1). Then we analyze the quality of the candidates using their quality measures [19], apply machine learning, and suggest the best reformulation to the initial query. Thus, our technique (1) first captures salient terms from the source documents by analyzing their structural aspects (i.e., unlike *bag of words* approaches [46]) and an appropriate term weight–CodeRank, and (2) then suggests the best query reformulation using document structures (i.e., multiple candidates derived from various signatures), query quality analysis and machine learning [19].

Experiments using 1,675 baseline queries from eight open

TABLE I
AN EXAMPLE CHANGE REQUEST (ISSUE #:31110, ECLIPSE.JDT.DEBUG)

Field	**Content**
Title	Debbugger Source Lookup does not work with variables
Description	In the Debugger Source Lookup dialog I can also select variables for source lookup. (Advanced... > Add Variables). I selected the variable which points to the archive containing the source file for the type, but the debugger still claims that he cannot find the source.
Initial Search Query	debugger source lookup work variables

```
public static IRuntimeClasspathEntry[] resolveRuntime
     ClasspathEntry(IRuntimeClasspathEntry entry,
     IJavaProject project) throws CoreException {
    switch (entry.getType()) {
      case IRuntimeClasspathEntry.PROJECT:
        // if the project has multiple output locations,
             they must be returned
        IResource resource = entry.getResource();
        if (resource instanceof IProject) {
          IJavaProject jp = JavaCore.create((IProject)
               resource);
          if (jp.exists() && jp.getProject().isOpen()) {
            IRuntimeClasspathEntry[] entries =
                 resolveOutputLocations(jp);
          }
        }
      break;
    ----------------------------------------------------
}}
```

Listing 1. Source code used for automatic query reformulation (abridged from [3])

source subject systems show that our technique can improve 71% (and preserve 26%) of the baseline queries which are highly promising according to relevant literature [13, 21, 34]. Our suggested queries return correct results for 64% of the queries in the Top-100 results. Our findings report that *CodeRank* is a more effective term weighting method than the traditional methods (e.g., TF, TF-IDF) for search query reformulation in the context of source code. Our findings also suggest that structure of a source code document is an important paradigm for both term weighting and query reformulation. Comparison with five closely related existing approaches [13, 21, 23, 43, 49] not only validates our empirical findings but also demonstrates the superiority of our technique. Thus, the paper makes the following contributions:

- A novel term weighting method –CodeRank– for source code that identifies the most important terms from a given code entity (e.g., class, method).
- A novel query reformulation technique that reformulates a given initial query using CodeRank, source document structures, query quality analysis and machine learning.
- Comprehensive evaluation using 1,675 baseline queries from eight open source subject systems.
- Comparison with five closely related existing approaches from the literature.

II. ACER: AUTOMATIC QUERY REFORMULATION USING CODERANK AND DOCUMENT STRUCTURES

Fig. 2 shows the schematic diagram of our proposed technique–ACER–for automatic query reformulation. We use a novel graph-based metric of term importance–*CodeRank*–for source code, and apply source document structures, query

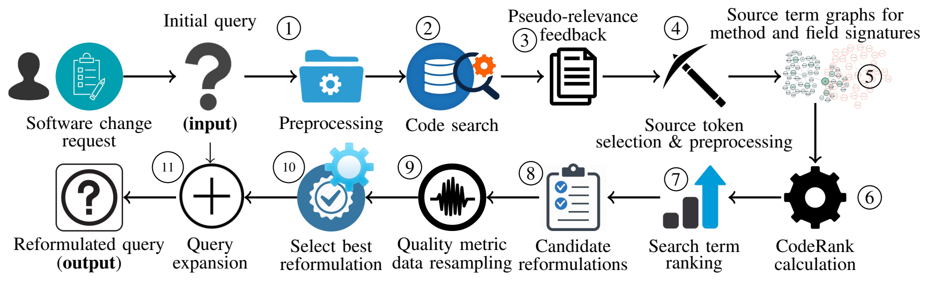
Initial query
1
2
Pseudo-relevance feedback
3
4
Source term graphs for method and field signatures
5
Software change request
(input)
Preprocessing
Code search
Source token selection & preprocessing
11
10
9
8
7
6
Reformulated query (output)
Query expansion
Select best reformulation
Quality metric data resampling
Candidate reformulations
Search term ranking
CodeRank calculation

Fig. 2. Schematic diagram of the proposed query reformulation technique–ACER

quality analysis and machine learning for query reformulation for concept location. We define CodeRank and discuss different steps of ACER in the following sections.

A. Pseudo-relevance Feedback

In order to suggest meaningful reformulations to an initial query, feedback on the query is required. Gay et al. [16] first reformulate queries based on explicit feedback from the developers. Although such feedback could be useful, gathering them is often time-consuming and sometimes infeasible. Hence, a number of recent studies [13, 21, 40, 41] apply pseudo-relevance feedback as a feasible alternative. The top ranked results returned by the code search tool for an initial query are considered as the *pseudo-relevance feedback* for the query. We first refine an initial query by removing the punctuation marks, numbers, special symbols and stop words (Step 1, Fig. 2). Then we collect the Top-K (i.e., $K = 10$, best performing heuristic according to our experiments) search results returned by the query, and use them as the source for our candidate terms for query reformulation (Steps 2, 3, Fig. 2).

B. Source Token Selection for Query Reformulation

Global Query Contexts: Pseudo-relevance feedback on an initial query provides a list of relevant source documents where one or more terms from the query generally occur. Sisman and Kak [49] choose such terms for query reformulation that frequently co-occur with the initial query terms within a fixed window size in the feedback documents. Hill et al. [23] consider presence of the query terms in method signatures or field signatures as an indicator of their relevance, and suggest natural language phrases from them as reformulated queries. Both reformulation approaches are highly subject to the quality of the initial query due to their imposed constraints– co-occurrences with query terms [49] and textual similarity with query terms [23]. Rocchio [43] determines importance (i.e., TF-IDF) of a candidate term across all the feedback documents, and suggests the top-ranked terms for query reformulation. Carmel et al. [12] suggest that a single natural language query might focus on multiple topics, and different parts of the returned results might cover different topics. That is, the same candidate term is not supposed to be important across all the feedback documents. In other words, accumulating term weight across all the documents might not always return the most appropriate terms for query reformulation. Such sort of calculation might add unnecessary noise to the term weight from the unrelated topics. Hence, we consider all the feedback documents as a *single body of structured texts* which acts as a *"global context"* for the query terms. Thus, with the help of an appropriate term weighting method, the terms representing the most dominant topic across the feedback documents (i.e., also in the initial query) could simply stand out, and could be chosen for reformulation.

Candidate Token Mining: Developers often express their intent behind the code and encode domain related concepts in the identifier names and comments [17]. However, code comments are often inadequate or outdated [51]. All identifier types also do not have the same level of importance. For example, while the signature of a method encodes the high level intent for the method, its body focuses on granular level implementation details and thus possibly contains more noisy terms [23]. In fact, Hill et al. [23] first analyze method signatures and field signatures to suggest natural language phrases as queries for code search. In the same vein, we thus also consider method signatures ($msig$) and field signatures ($fsig$) as the source for our candidate reformulation terms. We extract structured identifier names from these signatures using appropriate regular expressions [42] (Step 4, Fig. 2). Since different contexts of a source document might convey different types or levels of semantics (i.e., developers' intent), we develop a separate candidate token set (CT_{sig}) for each of the two signature types ($sig \in \{msig, fsig\}$) from the feedback documents ($\forall d \in D_{RF}$) as follows:

$$CT_{sig} = \bigcup_{\forall d \in D_{RF}} \{\exists t \in T_{sig}\} \mid structured(t) \wedge T_{sig} = sig(d)$$

Here $sig(d)$ extracts all tokens from method signatures or field signatures, and $structured(t)$ determines whether the token $t \in T_{sig}$ is structured or not. Although we deal with Java source code in this research where the developers generally use camel case tokens (e.g., `MessageType`) or occasionally might use same case tokens (e.g., `DECIMALTYPE`), our approach can be easily replicated for snake case tokens (e.g., `reverse_traversal`) as well.

C. Source Code Preprocessing

Token Splitting: Structured tokens often consist of multiple terms where the terms co-occur (i.e., are concatenated) due to their semantic or temporal relationships [48]. We first split each of the complex tokens based on punctuation marks (e.g., dot, braces) which returns the individual tokens (Step 4, Fig. 2). Then each of these tokens is splitted using a state-of-the-art token splitting tool–*Samurai* [14]–given that regular expression based splitting might not be always sufficient enough. *Samurai* mines software repositories to identify the most frequent terms, and then suggests the splits for a given token. We implement *Samurai* in our working environment where our subject systems (Section III-A) are used for mining the frequent terms, and the author's provided prefix and suffix lists [4] are applied to the splitting task.

Stop word and Keyword Removal: Since our structured tokens comprise of natural language terms, we discard stop words from them as a common practice (Step 4, Fig. 2). We use a standard list [6] hosted by Google for stop word removal. Programming keywords can often be considered as the equivalence of stop words in the source code which are also discarded from our analysis. Since we deal with Java source code, the keywords of Java are considered for this step. As suggested by earlier study [21], we also discard insignificant source terms (i.e., having word length$<$ 3) from our analysis.

Stemming: It extracts the root (e.g., "send") out of a word (e.g., "sending"). Although existing studies suggest contradictory [28, 45] or conflicting [24] evidences for stemming with the source code, we investigate the impact of stemming with RQ$_4$ where Snowball stemmer [24, 37] is used for stemming.

D. Source Term Graph Development

Once candidate tokens are extracted from method signatures and field signatures, and are splitted into candidate terms, we develop source term graphs (e.g., Fig. 1) from them (Step 5, Fig. 2). Developers often encode their intent behind the code and domain vocabulary into the carefully crafted identifier names where multiple terms are concatenated. For example, the method name–`getChatRoomBots`–looks like a natural language phrase–*"get chat room bots"*–when splitted properly. Please note that each of these three terms–*"chat", "room"* and *"bots"*– co-occur with each other to convey an important concept– a robotic technology, and thus, they are semantically connected. On the other hand, the remaining term–*"get"*– co-occurs with them due to a temporal relationship (i.e., develops a verbal phrase). Similar phrasal representations (refined with lexical matching) were directly returned by Hill et al. for query reformulation. However, their approach could be limited due to the added constraint (e.g., warrants query terms in signatures). We thus perform further analysis on such phrases, and exploit the co-occurrences among the terms for our graph based term weighting. In particular, we encode the term co-occurrences into connecting edges (E) in the term graph ($G(V,E)$) where the individual terms (V_i) are denoted as vertices (V).

$$V = \bigcup_{\forall t \in CT_{sig}} \{V_i \in splitted(t) \mid validterm(V_i)\}$$

$$E = \bigcup_{\exists V_i, V_j \in V} \{(V_i, V_j) \mid V_i, V_j \in t \wedge \mid i - j \mid = 1\}$$

Here $splitted(t)$ returns individual terms from the token $t \in CT_{sig}$, and $validterm(V_i)$ determines whether the term is valid (i.e., not an insignificant or a stop word) or not. We consider a *window size* of *two* within each phrase for capturing co-occurrences among the terms. Such window size for co-occurrence was reported to perform well by the earlier studies [9, 32, 41]. Thus, the above method name can be represented as the following edges– *get*$\longleftrightarrow$*chat*, *chat*$\longleftrightarrow$*room*, and *room*$\longleftrightarrow$*bots* – in the term graph. That is, if a set of terms are frequently shared across multiple tokens from two signature types, such occurrences are represented as the high connectivity in the term graph (e.g., *"Classpath"* in Fig. 1).

E. CodeRank Calculation

CodeRank: PageRank [10] is one of the most popular algorithms for web link analysis which was later adapted by Mihalcea and Tarau [32] for text documents as TextRank. In this research, we adapt our term weighting method from TextRank [9, 32, 41] for source code, and we call it *CodeRank*. To date, only traditional term weights (e.g., TF, TF-IDF [21, 43, 49]) are applied to source code which were originally proposed for regular texts [26] and are mostly based on isolated frequencies. On the contrary, CodeRank not only analyzes the connectivity (i.e., incoming links and outgoing links) of each source term, but also the relative weight of the connected terms from the graph recursively, and calculates the term weight, $S(V_i)$, as follows (Step 6, Fig. 2):

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

Here, $In(V_i)$, $Out(V_j)$, and ψ denote the vertices to which V_i is connected through incoming links, the vertices to which V_j is connected through outgoing links, and the damping factor respectively. As shown earlier using the example– `getChatRoomBots`, co-occurred terms complement each other with their semantics which are represented as bi-directional edges in the term graph. Thus, each (V_i) of the vertices from the graph has equal number of incoming links and outgoing links, i.e., *in-degree(V_i)=out-degree(V_i).*

Parameters and Configurations: Brin and Page [10] consider damping factor, ψ, as the probability of randomly choosing a web page in the context of web surfing by a random surfer. That is, $1-\psi$ is the probability of jumping off that page by the surfer. They use a well-tested value of 0.85 for ψ which was later adopted by Mihalcea and Tarau [32] for text documents. Similarly, we also use the same value of ψ for *CodeRank* calculation. Each of the vertices is assigned to a default value (i.e., base term weight) of 0.25 (as suggested by earlier studies [10, 32]) with which CodeRank is calculated. It should be noted that the base weight of a vertex does not determine its final weight when PageRank based algorithms are applied [32]. CodeRank adopts the underlying mechanism of recommendation or votes [32, 41] for term weighting. That is, each vertex feeds off from the scores of surrounding connected vertices from the graph in terms of recommendation (i.e., incoming edges). PageRank generally has two modes of computation–*iterative* version and *random walk* version. We use the iterative version for CodeRank, and the computation iterates until the weights of the terms converge below a certain threshold or they reach the maximum iteration limit (i.e., 100 as suggested by Blanco and Lioma [9]). As applied earlier [32], we apply a heuristic threshold of 0.0001 for the convergence checking. The algorithm captures importance of a source term not only by estimating its local impact but also by considering its global influence over other terms. For example, the term, *"Classpath"*, Fig. 1, occurs in multiple structured tokens (Listing 1), complements the semantics of five other terms, and thus is highly important within the term graph (i.e., Fig. 1). Once the iterative computation is over, each of the terms from the graph is found with a numeric score. We consider these scores as the relative weight or importance of the corresponding terms from the source code.

F. Suggestion of the Best Query Reformulation

Candidate Reformulation Selection: Algorithms 1 and 2 show the pseudo-code of our query reformulation technique–ACER–for concept location. We first collect pseudo-relevance feedback for the initially provided query (Q) where Top-K source documents are returned (Lines 3–5, Algorithm 1). Then we collect method signatures and field signatures from each of the documents ($\forall d \in D_{RF}$), and extract structured tokens from them. We prepare three token sets–CT_{msig}, CT_{fsig} and CT_{comb} from these signatures (Lines 6–12, Algorithm 1, Step 4, Fig. 2) where CT_{comb} combines tokens from both signatures. Then we perform limited natural language preprocessing on each token set where *Samurai* algorithm [14] is used for token splitting. We develop separate term graph for each of these token sets where individual terms are represented as vertices, and term co-occurrences are encoded as connecting edges (Lines 3–7, Algorithm 2, Step 5, Fig. 2). We apply CodeRank term weighting to each of the graphs which provides a ranked list of terms based on their relative importance. Then we select Top-K (e.g., $K = 10$) important terms from each of the three graphs, and prepare three reformulation candidates (Lines 8–12, Algorithm 2, Steps 6, 7, 8, Fig. 2).

Algorithm 1 ACER: Proposed Query Reformulation

```
1:  procedure ACER($Q$)                    ▷ $Q$: initial search query
2:      $L \leftarrow \{\}$                 ▷ list of best reformulation query terms
3:      ▷ collecting pseudo-relevance feedback for $Q$
4:      $Q_{pp} \leftarrow$ preprocess($Q$)
5:      $D_{RF} \leftarrow$ getRelevanceFeedback($Q_{pp}$)
6:      ▷ collecting candidate source tokens from signatures
7:      for SourceDocument $d \in D_{RF}$ do
8:          $CT_{msig} \leftarrow CT_{msig} \cup$ getMethodSigTokens($d$)
9:          $CT_{fsig} \leftarrow CT_{fsig} \cup$ getFieldSigTokens($d$)
10:     end for
11:     $CT_{comb} \leftarrow CT_{msig} \cup CT_{fsig}$
12:     $CT_{all} \leftarrow \{CT_{msig}, CT_{fsig}, CT_{comb}\}$
13:     for TokenList $CT_{sig} \in CT_{all}$ do
14:         $QR[sig] \leftarrow$ getQRCandidate($CT_{sig}$)
15:     end for
16:     ▷ suggesting the best reformulated query for $Q$
17:     $QD \leftarrow$ resample(getQueryQualityMetrics($QR$))
18:     $QR_{best} \leftarrow$ getBestCandidateUsingML($QR, Q_{pp}, QD$)
19:     $L \leftarrow$ combine($Q_{pp}, QR_{best}$)
20:     return $L$
21: end procedure
```

Selection of the Best Reformulation: Haiduc et al. [21] argue that the same type of reformulation (i.e., addition, deletion or replacement of query terms) might not be appropriate for all given queries. In the same vein, we argue that query reformulations from different contexts of the source document (e.g., method signature, field signature) might have different level of effectiveness given that they embody different level of semantics and noise. That means, one or more of the reformulation candidates could improve the initial query, but the best one should be chosen carefully for useful recommendation.

Haiduc et al. [19] suggest that quality of a query with respect to the corpus could be determined using four of its statistical properties– *specificity, coherency, similarity* and *term relatedness*–that comprise of 21 metrics [11]. They apply machine learning on these properties, and separate high quality queries from low quality ones. We thus also similarly apply machine learning on our reformulation candidates (and their metrics), and develop classifier model(s) where *Classification And Regression Tree* (CART) is used as the learning algorithm [19]. Since only the best of the four reformulation candidates (i.e., including baseline) is of our interest, the training data was inherently skewed. We thus perform *bootstrapping* (i.e., random resampling) [27, 50] on the data multiple times (e.g., 50) with 100% sample size and replacement (Step 9, Fig. 2), train multiple models using the sampled data, and then record their output predictions. Then, we average all the predictions for each test instance from all models, and determine their average probability of being the best candidate reformulation. Thus, we identify the best of the four candidates using our models, and suggest the best reformulation to the initial query (Lines 16–20, Algorithm 1, Steps 10, 11, Fig. 2). Bassett and Kraft [8] suggest that repetition of certain query terms might improve retrieval performance of the query. If none of the candidates is likely to improve the initial query according to the quality model (i.e., baseline itself is the best), we repeat all the terms from the initial query as the reformulation.

Algorithm 2 getQRCandidate: Get a candidate reformulation

```
1:  procedure GETQRCANDIDATE($CT_{sig}$)      ▷ $CT_{sig}$: extracted
    candidate tokens from the signatures $sig$
2:      $QR_{sig} \leftarrow \{\}$              ▷ candidate query reformulation
3:      ▷ extracting terms and their co-occurrences
4:      $ST_{sig} \leftarrow$ preprocess(Samurai($CT_{sig}$))
5:      $CO_{sig} \leftarrow$ getTermCo-occurrences($ST_{sig}$, $CT_{sig}$)
6:      ▷ developing term graph from token set
7:      $G_{sig} \leftarrow$ developTermGraph($ST_{sig}, CO_{sig}$)
8:      ▷ calculating CodeRank using the graph
9:      $CR_{sig} \leftarrow$ normalize(calculateCodeRank($G_{sig}$))
10:     ▷ getting candidate reformulated query
11:     $QR_{sig} \leftarrow$ getTopKTerms(sortByValue($CR_{sig}$))
12:     return $QR_{sig}$
13: end procedure
```

Working Example: Let us consider the query–{debugger source lookup work variables}–from our running example in Table II. Our term weighting method–*CodeRank*–extracts three candidate reformulations from method signatures and field signatures. We see that different candidates have different level of effectiveness (i.e., rank 02 to rank 16), and in this case, the candidate from the method signatures (QR_{msig}) is the most effective. Our technique–ACER– not only prepares such candidate queries from various contexts (using a novel

TABLE II
A WORKING EXAMPLE (BUG #31110, ECLIPSE.JDT.DEBUG)

Source	Query Terms	QE
Bug Title	Debbugger Source Lookup does not work with variables	72
Initial Query (Q)	{debugger source lookup work variables}	77
Q'_{msig}	Q_{pp} ∪ (QR_{msig}={launch debug resolve required classpath})	**02**
Q'_{fsig}	Q_{pp} ∪ (QR_{fsig}={label classpath system resolution launch})	**06**
Q'_{comb}	Q_{pp} ∪ (QR_{comb}={java type launch classpath label})	16
QR_{best} = getBestCandidateUsingML(QR_{msig}, QR_{fsig}, QR_{comb}, Q_{pp}, QD)		
Q'_{ACER}	$Q_{pp} \cup QR_{best}$	**02**

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

TABLE III
EXPERIMENTAL DATASET

System	#Classes	#CR	System	#Classes	#CR
`eclipse.jdt.core`–4.7.0	5,908	198	`ecf`–279.279	2,827	154
`eclipse.jdt.debug`–4.6.0	1,519	154	`log4j`–1.2.18	309	28
`eclipse.jdt.ui`–4.7.0	10,927	309	`sling`–9.0	4,328	76
`eclipse.pde.ui`–4.6.0	5,303	302	`tomcat70`–7.0.73	1,841	454

CR= Change requests

term weighting method) but also suggests the best candidate (QR_{best}) for query reformulation. The reformulated query–{debugger source lookup work variables *launch debug resolve required classpath*} – returns the first correct result at the top position (i.e., rank 02) of the result list which is highly promising. Such effective reformulations are likely to reduce a developer's effort during software change implementation.

III. EXPERIMENT

Although pre-retrieval methods (e.g., coherency, specificity [19]) are lightweight and reported to be effective for query quality analysis, post-retrieval methods are more accurate and more reliable [21]. Existing studies [21, 34, 41, 45] also adopt these methods widely for evaluation and validation. We evaluate our term weighting method and query reformulation technique using 1,675 baseline queries and three performance metrics. We also compare our technique with five closely related existing techniques [13, 21, 23, 43, 49]. We thus answer five research questions using our experiments as follows:

- **RQ$_1$:** Does query reformulation of ACER improve the baseline queries significantly in terms of query effectiveness and retrieval performance?
- **RQ$_2$:** Does CodeRank perform better than traditional term weighting methods (e.g., TF, TF-IDF) in identifying effective search terms from the source code?
- **RQ$_3$:** Does employment of document structure improve ACER's suggestion on good quality search terms from the source code?
- **RQ$_4$:** How stemming, query length, and relevance feedback size affect the performance of our technique?
- **RQ$_5$:** Can ACER outperform the existing query reformulation techniques from the literature in terms of effectiveness and retrieval performance of the queries?

A. Experimental Dataset

Data Collection: We collect a total of 1,675 bug reports from eight open source subject systems (i.e., five *Eclipse* systems and three *Apache* systems) for our experiments. Table III shows the experimental dataset. We first extract resolved bug reports (i.e., marked as RESOLVED) from BugZilla and JIRA repositories, and then collect corresponding bug-fixing commits from GitHub version control histories of these eight systems. Such approach was regularly adopted by the relevant literature [8, 21, 41, 49], and we also follow the same. In order to ensure a fair evaluation or validation, we discard the bug reports from our dataset for which no source code files (e.g., Java classes) were changed or no relevant source files exist in the system snapshot collected for our study. We also discard such bug reports that contain stack traces using appropriate regular expressions [33]. They do not represent a typical change request (i.e., mostly containing natural language texts) from the regular software users.

Baseline Query Selection: We select the *title* of a bug report as the baseline query for our experiments, as was also selected by earlier studies [21, 28, 49]. However, we discard such queries that (i.e., in verbatim titles) already return their first correct results within the Top-10 positions, i.e., they possibly do not need query reformulation [21]. Finally, we ended up with a collection of 1,675 baseline queries. We perform the same preprocessing steps as were done on the source documents (Section II-C), on the queries before using them for code search in our experiments.

Goldset Development: Developers often mention a Bug ID in the title of a commit when they fix the corresponding reported bug [7]. We collect the *changeset* (i.e., list of changed files) from each of our selected bug-fixing commits, and develop individual solution set (i.e., *goldset*) for each of the corresponding bug reports. Such solution sets are then used for the evaluation and validation of our suggested queries.

Replication: All experimental data and relevant materials are hosted online [1] for replication or third party reuse.

B. Corpus Indexing & Source Code Search

Since we locate concept within project source, each of the source files is considered as an individual document of the corpus [45]. We apply the same preprocessing steps on the corpus documents as were done for query reformulation (i.e., details in Section II-C). We remove punctuation marks and stop words from each document. Then, we split the structured tokens, and keep both the original and the splitted tokens in the preprocessed documents. We then apply *Apache Lucene*, a *Vector Space Model (VSM)* based popular search engine, to index all the documents and to search for relevant documents from the corpus for any given query. Such approaches and tools were widely adopted by earlier studies [21, 28, 41, 47].

C. Performance Metrics

Query Effectiveness (QE): It approximates the effort required to find out the first correct result for a query. In other words, query effectiveness is defined as the rank of the first correct result returned by the query [33]. The lower the effectiveness score, the better the query is.

Mean Reciprocal Rank (MRR): Reciprocal rank is defined as the multiplicative inverse of query effectiveness measure. Mean Reciprocal Rank averages such measures for all the queries. The higher the MRR value, the better the query is.

TABLE IV
EFFECTIVENESS OF ACER QUERY AGAINST BASELINE QUERY

System	#Queries	Improvement							Worsening							Preserving
		#Improved	Mean	Q1	Q2	Q3	Min.	Max.	#Worsened	Mean	Q1	Q2	Q3	Min.	Max.	#Preserved
ecf	154	100 (**64.94%**)	71	**8**	20	58	1	654	5 (3.25%)	125	48	88	220	43	329	49 (31.82%)
jdt.core	198	125 (**63.13%**)	89	**8**	20	51	1	1,485	7 (3.54%)	72	16	38	132	13	195	66 (33.33%)
jdt.debug	154	110 (**71.43%**)	72	**10**	23	73	1	1,234	3 (1.95%)	138	48	102	265	48	265	41 (26.62%)
jdt.ui	309	216 (**69.90%**)	169	**10**	27	92	1	3,162	13 (4.21%)	254	39	91	368	19	1,369	80 (25.89%)
pde.ui	302	191 (**63.25%**)	143	**8**	33	102	1	2,304	7 (2.32%)	507	70	477	1,060	40	1,172	104 (34.44%)
log4j	28	23 (**82.14%**)	35	**12**	17	58	3	136	0 (0.00%)	-	-	-	-	-	-	5 (17.86%)
sling	76	59 (**77.63%**)	165	**9**	18	120	2	1,940	0 (0.00%)	-	-	-	-	-	-	17 (22.37%)
tomcat70	454	345 (**75.99%**)	236	21	92	291	1	1,675	22 (4.84%)	292	97	261	429	34	938	87 (19.16%)
	Total = 1,675	**Avg = 71.05%**							Avg = 2.51%							**Avg = 26.44%**

jdt.core = `eclipse.jdt.core`, **jdt.debug** = `eclipse.jdt.debug`, **jdt.ui** = `eclipse.jdt.ui`, **pde.ui** = `eclipse.pde.ui`, **Mean** = Mean rank of first correct results returned by the queries, $\mathbf{Q}_i = i^{th}$ quartile of all ranks considered

Top-K Accuracy: It refers to the percentage of queries by which at least one correct result is returned within the Top-K results. The higher the metric value, the better the queries are.

D. Evaluation of ACER and CodeRank

We evaluate our technique using 1,675 baseline queries from eight subject systems and three performance metrics discussed above. We determine effectiveness and retrieval performance of our suggested reformulated queries, and then compare them with their baseline counterparts. We also contrast our term weight with traditional term weights, and calibrate our technique using various configurations.

Answering $\mathbf{RQ_1}$–Effectiveness of ACER Queries: Table IV and V show the effectiveness of ACER queries. If our query returns the first correct result closer to the top position than the baseline query, then we consider that as *query improvement*, and the vice versa as *query worsening*. If both queries return their first correct results at the same position, we cosider that as *query preserving*. From Table IV, we see that ACER can improve or preserve 97% of the baseline queries (i.e., about 71% improvement and about 26% preserving) while worsening the quality of only about 3% of the queries. All these statistics are highly promising according to the relevant literature [21, 34, 41], i.e., maximum 52% improvement reported [21], and they demonstrate the potential of our technique. When individual systems are considered, our technique provides 63%–82% improvement across eight systems. According to the quantile analysis in Table IV, 25% of our queries return their first correct results within the Top-10 positions for all the systems except two (i.e., Top-12 position for `log4j` and Top-21 position for `tomcat70`). Please note that only 6% of the baseline queries return their correct results within the Top-10 positions (Table VI). On the contrary, 25% of our queries do so for six out of eight systems, which demonstrates the potential of our technique. While query improvement ratios are significantly higher than the worsening ratios (i.e., 28 times higher), it should be noted that our technique does not worsen any of the queries for two of the systems–`log4j` and `sling`.

Table V reports further effectiveness and the extent of actual rank improvements by our suggested queries. We see that reformulations from the method signatures improve the baseline queries significantly. For example, they improve 59% of the baseline queries while worsening 38% of them. Reformulations from the field signatures are found relatively less effective. However, ACER reduces the worsening ratio to as low as 2.51%, and increases the improvement ratio up to 71%, which are highly promising. More importantly, the mean rank differences (MRD) suggest that ACER *elevates* first correct results in the ranked list by **81** positions on average for at least 71% of the queries while *dropping* them for only 3% of the queries by 104 positions. Such rank improvements are likely to reduce human efforts significantly during concept location.

TABLE V
EFFECTIVENESS OF ACER VARIANTS AGAINST BASELINE QUERIES

Query Pairs	**Improved** (MRD)	**Worsened** (MRD)	p-value	Preserved
ACER_{msig} vs. Baseline	**58.93%** (-61)	37.99% (+131)	***0.007**	**3.08%**
ACER_{fsig} vs. Baseline	52.51% (-51)	44.57% (+151)	0.063	**2.91%**
ACER_{comb} vs. Baseline	**58.62%** (-51)	38.19% (+136)	***0.018**	**3.20%**
ACER vs. Baseline	**71.05% (-81)**	2.51% (+104)	***<0.001**	**26.44%**

* = Statistically significant difference between improvement and worsening, **MRD** = Mean Rank Difference between ACER and baseline queries

Retrieval Performance of ACER Queries: Table VI reports the comparison of retrieval performance between our queries and baseline queries. Given that most of our selected queries are difficult (i.e., no correct results within the Top-10 positions [21]), the baseline queries retrieve at least one correct result within the Top-100 positions for 56% of the cases. However, our reformulations improve this ratio to about 64%, and the improvement is statistically significant (i.e., *paired t-test, p-value=0.010<0.05*, *Cohen's D=0.68 (moderate)*). Similar scenarios are observed with mean reciprocal rank as well.

Thus, to answer $\mathbf{RQ_1}$, the reformulation of ACER improves the baseline queries significantly both in terms of query effectiveness and retrieval performance. ACER improves 71% of the baseline queries with 64% Top-100 retrieval accuracy.

Answering $\mathbf{RQ_2}$–CodeRank vs. Traditional Term Weighting Methods: Table VII shows the comparative analysis between CodeRank and two traditional term weights–TF and TF-IDF– which are widely used in the text retrieval contexts [13, 28, 43]. While TF estimates the importance of a term based on its occurrences within a document, TF-IDF additionally captures the global occurrences of the term across all the documents of the corpus [26]. On the contrary, CodeRank employs a graph-based scoring mechanism that determines the importance of a term based on its co-occurrences with other important terms within a certain context. From Table VII, we see that CodeRank performs significantly better than both TF (i.e., *paired t-test, p-value=0.005<0.05*) and TF-IDF (i.e., *p-value<0.001*) in identifying important search terms from source code, especially from the method signatures. Considering the whole source code rather than signatures improves the performance of both TF (i.e., 56% query improvement) and

TABLE VI
COMPARISON OF ACER'S RETRIEVAL PERFORMANCE WITH BASELINE QUERIES

Query	Metric	Top-10	Top-20	Top-50	Top-100
Baseline	Top-K Accuracy	5.78%	18.91%	41.09%	56.30%
	MRR@K	0.01	0.02	0.03	0.03
ACER$_{msig}$	Top-K Accuracy	10.45%	21.48%	38.12%	51.31%
	MRR@K	0.02	0.03	0.04	0.04
ACER$_{fsig}$	Top-K Accuracy	7.77%	17.40%	36.25%	47.23%
	MRR@K	0.02	0.03	0.03	0.03
ACER$_{comb}$	Top-K Accuracy	8.68%	20.78%	36.87%	51.75%
	MRR@K	0.02	0.03	0.03	0.04
ACER	Top-K Accuracy	***14.72%**	***31.22%**	***49.89%**	***63.89%**
	MRR@K	0.04	0.05	**0.06**	**0.06**

* = Statistically significant difference between ACER and baseline

TABLE VII
COMPARISON BETWEEN CODERANK AND TRADITIONAL TERM WEIGHTS

Query Pairs	Improved	Worsened	Preserved
ACER$_{msig}$ vs. TF$_{msig}$	***58.93%** / 53.40%	***37.99%** / 44.60%	3.08% / 2.00%
ACER$_{fsig}$ vs. TF$_{fsig}$	52.51% / 51.57%	44.57% / 46.85%	2.91% / 1.57%
ACER$_{comb}$ vs. TF$_{comb}$	***58.62%** / 54.34%	***38.19%** / 44.11%	3.20% / 1.54%
ACER vs. TF$_{all}$	***71.05%** / 56.01%	***2.51%** / 41.44%	***26.44%** / 2.55%
ACER$_{msig}$ vs. TF-IDF$_{msig}$	***58.93%** / 45.55%	***37.99%** / 49.88%	3.08% / 4.57%
ACER$_{fsig}$ vs. TF-IDF$_{fsig}$	52.51% / 51.06%	44.57% / 46.77%	2.91% / 2.17%
ACER$_{comb}$ vs. TF-IDF$_{comb}$	***58.62%** / 50.35%	***38.19%** / 47.25%	3.20% / 2.40%
ACER vs. TF-IDF$_{all}$	***71.05%** / 52.17%	***2.51%** / 45.13%	***26.44%** / 2.70%

* = Statistically significant difference between ACER measures and their counterparts

TF-IDF (i.e., 52% query improvement). However, our term weight–CodeRank–is still better alone (i.e., 59%), and improves significantly higher (i.e., *p-value=1.717e-06*) fraction (i.e., 71%) of the baseline queries when employed with our proposed reformulation algorithm–ACER.

Fig. 3 shows how *CodeRank* and traditional term weights perform in reformulating the baseline queries with their (a) Top-10 and (b) Top-30 terms. We see that TF reaches its peak performance pretty quickly (i.e., $K = 3$), and then shows a stationary or irregular behaviour. That means, TF identifies frequent terms for query reformulation, and few of them (e.g., Top-3) could be highly effective. On the contrary, our method–CodeRank– demonstrates a gradual improvement in the performance up to Top-12 terms (i.e., K=12, Fig. 3-(b)), and crosses the performance peak of TF with a large margin (i.e., *paired t-test, p-value=0.004<0.05, Cohen's D=3.77>1.00 (large))*, for K=10 to K=15). CodeRank emphasizes on the votes from other important terms (i.e., by leveraging co-occurrences) for determining weight of a term, and as demonstrated in Fig. 3, this weight is found to be more reliable than TF. TF-IDF is found relatively less effective according to our investigation.

Thus, to answer **RQ$_2$**, CodeRank performs significantly better than traditional methods in identifying effective terms for query reformulation from the source code.

Answering RQ$_3$–Do Document Structures Matter? While most of the earlier reformulation techniques miss or ignore the structural aspect of a source document, we consider such aspect as an important paradigm of our technique. We consider a source document as a collection of structured entities (e.g., signatures, methods, fields) [38] rather than a regular text document. Thus, we make use of method signatures and field signatures rather than the whole source code for query reformulation given that they are likely to contain more salient terms and less noise [23]. Fig. 4 demonstrates how incorporation of document structures into a technique could be useful for query reformulations. We see that reformulations using method signatures and field signatures improve two different sets of baseline queries, and this happens with both term weighting methods–(a) CodeRank and (b) TF. While these sets share about half of the queries (49%–57%), reformulations based on each signature type also improve a significant amount (i.e., 19% (73+136+24) – 25% (105+152+46)) of unique baseline queries. In Fig. 4-(c), when these signatures (i.e., along with ACER) are contrasted with the whole source code (i.e., along with TF), we even found that the signature-based reformulations outperform the whole code-based reformulations by a large margin (i.e., (25.2%–8.39%) ≈ 17% more query improvement). That is, the use of the whole source code introduces additional noise, and diminishes the strength or salience of the individual structures (i.e., signatures). Most of the existing methods [16, 21, 40] suffer from this limitation. On the contrary, our technique ACER exploits document structures (i.e., signatures), and carefully chooses the best among all the candidate reformulations derived from such structures using query quality analysis and machine learning.

Thus, to answer **RQ$_3$**, document structures improve the suggestion of query reformulation terms from the source code.

Answering RQ$_4$– Impact of Stemming, Query Length, and Relevance Feedback: From Table VIII, we see that stemming generally degrades the effectiveness of our reformulated queries. Similar findings were also reported by earlier studies [28, 45]. Fig. 5 shows how (a) Top-10 and (b) Top-30 reformulation terms improve the baseline queries. We see that

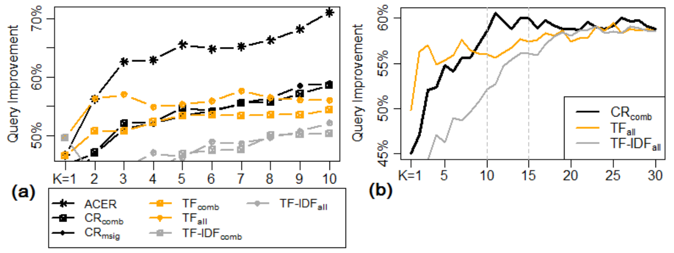

Fig. 3. Comparison of query improvement between CodeRank and traditional term weights for (a) Top-10 and (b) Top-30 reformulated query terms

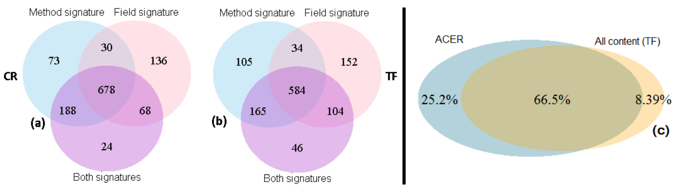

Fig. 4. Improved queries by reformulation from method signatures and field signatures using (a) CodeRank (CR) and (b) Term Frequency (TF). (c) ACER vs. TF (all content)

TABLE VIII
IMPACT OF STEMMING ON QUERY EFFECTIVENESS

Source	Query	Improved (MRD)	Worsened (MRD)	Preserved
Method signature	ACER$_{msig,stem}$	**52.66%** (-58)	44.73% (+127)	2.61%
	ACER$_{msig}$	***58.93%** (-61)	***37.99%** (+131)	3.08%
Field signature	ACER$_{fsig,stem}$	48.14% (-53)	47.47% (+151)	4.39%
	ACER$_{fsig}$	52.51% (-51)	44.57% (+151)	2.91%
Both signatures	ACER$_{comb,stem}$	52.68% (-57)	44.38% (+128)	2.94%
	ACER$_{comb}$	*58.62% (-51)	*38.19% (+136)	3.20%
Both signatures	ACER$_{stem}$	**68.11%** (-78)	5.37% (+67)	**26.51%**
	ACER	**71.05%** (-81)	***2.51%** (+104)	**26.44%**

* = Statistically significant difference between two measures from the same signature, **MRD** = Mean Rank Difference between ACER and baseline queries

TABLE IX
COMPARISON OF QUERY EFFECTIVENESS WITH EXISTING TECHNIQUES

Technique	#Queries	Improvement							Worsening							Preserving
		#Improved	Mean	Q1	Q2	Q3	Min.	Max.	#Worsened	Mean	Q1	Q2	Q3	Min.	Max.	#Preserved
Hill et al. [23]	1,675	631 (37.67%)	157	18	48	161	1	2,264	760 (45.37%)	261	54	119	300	4	4,819	284 (16.96%)
Rocchio [43]	1,675	895 (53.43%)	219	15	49	188	1	4,609	739 (44.11%)	333	65	170	429	3	3,489	41 (2.45%)
RSV [13]	1,675	**914 (54.57%)**	216	15	52	195	1	4,611	723 (43.16%)	307	63	160	415	7	3,387	38 (2.27%)
Sisman and Kak [49]	1,675	759 (45.31%)	207	17	61	213	1	3,707	642 (38.33%)	273	59	147	345	8	2,545	274 (16.36%)
Refoqus [21]	1,675	**895 (53.43%)**	217	15	51	188	1	4,609	737 (44.00%)	332	65	170	429	3	3,489	43 (2.57%)
Refoqus$_{sampled}$ [21]	1,675	**1,154 (68.90%)**	156	11	33	141	1	4,609	**487 (29.07%)**	325	63	166	406	6	3,489	**34 (2.03%)**
ACER$_{msig}$	1,675	969 **(57.85%)**	208	14	49	192	1	3,649	662 (39.52%)	272	52	139	341	2	4,825	44 (2.63%)
ACER$_{comb}$	1,675	958 (57.19%)	216	15	49	194	1	4,117	674 (40.24%)	275	52	139	336	4	3,360	43 (2.57%)
ACER	1,675	***1,169 (69.79%)**	156	**11**	35	130	1	3,162	***57 (3.40%)**	260	53	140	375	13	1,369	***449 (26.81%)**
Baseline	1,675	-	227	32	88	258	3	4,787	-	113	24	49	162	1	718	-
ACER$_{ext}$	1,755	***1,192 (67.92%)**	149	**10**	34	124	1	3,162	***48 (2.74%)**	301	50	145	327	13	1,782	***515 (29.34%)**

Mean = Mean rank of first correct results returned by the queries, $\mathbf{Q}_i$= i^{th} quartile of all ranks considered, * = Statistically significant difference between ACER measures and their counterparts

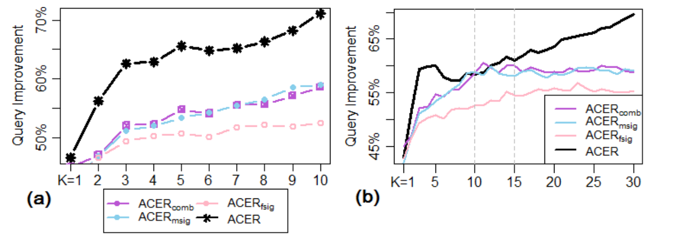

Fig. 5. Effectiveness of ACER queries for (a) Top-10 and (b) Top-30 reformulated terms

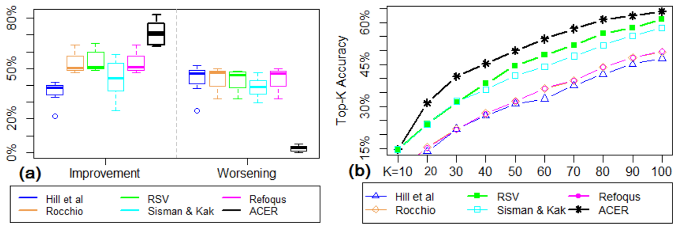

Fig. 6. Comparison of (a) query effectiveness, and (b) retrieval performance

our reformulations perform the best (i.e., about 60% query improvement) with Top-10 to 15 search terms collected from each signature type. However, when query quality analysis [19] is employed, our technique–ACER–can improve 71% of the baseline queries with only Top-10 reformulation terms. We also repeat the same investigation with Top-30 terms, and achieved the same top performance (i.e., Fig. 5-(b)). Thus, our choice of returning Top-10 reformulation terms is justified. We also investigate how the size of pseudo-relevance feedback influences our performance, and experimented with Top-30 documents. We found that reformulations for ACER reach the performance peak when Top-10 to 15 feedback source documents (i.e., returned by the baseline queries) are analyzed for candidate terms. This possibly justifies our choice of Top-10 documents as the pseudo-relevance feedback.

Thus, to answer **RQ$_4$**, stemming degrades the query effectiveness of ACER. Reformulation size and relevance feedback size gradually improve the performance of ACER as long as they are below a certain threshold (i.e., $K = 15$).

E. Comparison with Existing Approaches

Answering RQ$_5$: While the empirical evaluation in terms of performance metrics above clearly demonstrates the promising aspects of our query reformulation technique, we still compare with five closely-related existing approaches [13, 21, 23, 43, 49]. Hill et al. [23] suggest relevant phrases from method signatures and field signatures as query reformulations. While Sisman and Kak [49] focus on term co-occurrences with query keywords, Rocchio [43] and RSV [13] apply TF-IDF based term weights for choosing query reformulation terms. Refoqus [21] is closely related to ours and is reported to perform better than RSV and other earlier approaches, which probably makes it the state-of-the-art for our research problem. We replicate each of Hill et al., Rocchio, RSV, Sisman and Kak, and Refoqus in our working environment by carefully following their algorithms, equations and methodologies given that their implementations are not publicly available. In the case of Refoqus, we implement 27 metrics (20 pre-retrieval [19] and 7 post-retrieval [21]) that estimate query difficulty. We develop a machine learning model using CART algorithm (i.e., as used by them) and 10-fold cross validation. Then, we use the model to return the best reformulation out of four candidates of Refoqus– *query reduction, Dice expansion, Rocchio's expansion* and *RSV expansion*–for each baseline query. Table IX and Fig. 6 summarize our comparative analyses.

From Table IX, we see that RSV and Refoqus perform better than the other existing approaches. They improve about 55% and about 53% of the baseline queries respectively. Such ratios are also pretty close to the originally reported performances by Haiduc et al. on a different dataset, which possibly validates the correctness of our implementation. While 55% query improvement is the maximum performance provided by any of the existing approaches, our technique–ACER–improves about 70% of the baseline queries (i.e., 1% difference between Table V and Table IX due to rounding error) which is significantly higher, i.e., *paired t-test, p-value=6.663e-06<0.05, Cohen's D=2.43>1.00 (large)*. Refoqus adopts a similar methodology like ours. Unfortunately, the approach is limited due to possibly the low performance of its candidate reformulations. One might argue about the data resampling step (i.e., Step 9, Fig. 2) of ACER for the high performance. However, we also apply data resampling to Refoqus using the same settings as ours for further investigation. We see that Refoqus$_{sampled}$ has a similar improvement ratio like ours, but it still worsens a significant amount of queries, 29%, compared to our 3.40%. Thus, our technique still performs better than Refoqus in the equal settings. Our quantile measures and mean ranks are more promising than those from the baseline or competing methods as reported in Table IX. Table **V** and **RQ$_1$** also suggest that our queries have high potential for reducing human efforts. We also experiment with an extended dataset (i.e., 1,755=1,675 + 8x10) containing 80 very good queries. As reported in Table

IX, ACER$_{ext}$ mostly preserves the good quality queries rather than worsening, which also demonstrates its high potential.

Fig. 6-(a) shows the box plots of query improvement and query worsening ratios by all the techniques under study. We see that ACER outperforms the existing techniques including the state-of-the-art [21] by a large margin. Our median improvement ratio is about 75%, which is higher than even the maximum improvement ratios of the counterparts, which demonstrates the promising aspect of ACER. Fig. 6-(b) shows the Top-K accuracy of the query reformulation techniques. We see that our accuracy is relatively higher than that of each of the existing approaches across various Top-K (i.e., 10–100) values. The best performing existing method is RSV. However, our performance is significantly higher than that of RSV for various K values according to statistical significance tests (i.e., *paired t-test, p-value=0.0001<0.05, Cohen's D=0.34*).

Thus, to answer $\mathbf{RQ_5}$, our technique outperforms the state-of-the-art techniques in terms of reformulation query effectiveness, and performs significantly better than each of the existing techniques in terms of document retrieval accuracy.

IV. Threats to Validity

Threats to *internal validity* relate to experimental errors and biases [55]. Although CodeRank and document structures play a major role, the data resampling step (Section II-F, Step 9, Fig. 2) has a significant role behind the high performance of our technique. Unfortunately, to the best of our knowledge, Refoqus [21] does not have such a step. Thus, the performance comparison might look like a bit unfair. Besides, models based on data resampling are sometimes criticized for intrinsic biases [5]. However, we apply data resampling to Refoqus as well (i.e., Refoqus$_{sampled}$), and demonstrate that our technique still performs better in terms of worsening ratio.

Threats to *external validity* relate to the generalization of the obtained results [21]. All of our subject systems are Java-based. So, there might be different results with systems from other programming languages. However, we experimented with eight different systems with promising performance, and the comparison with the state-of-the-art techniques demonstrates the superiority of our approach.

V. Related Work

There exist a number of studies in the literature that reformulate a given query for concept location in the context of software change tasks. Existing studies apply relevance feedback from developers [16], pseudo-relevance feedback from IR tools [21], partial phrasal matching [23, 44], and machine learning [21, 34] to query reformulation. They also make use of context of query terms from source code [25, 40, 49, 53], text retrieval configuration [21, 34], and quality of queries [19, 20] in suggesting the reformulated queries. Hill et al. [23] consider the presence of query terms in the method or field signatures as an indicator of their relevance, and suggest natural language phrases from them as reformulated queries. Sisman and Kak [49] choose such terms for query reformulation that frequently co-occur with query terms within a fixed size of window in the code. Rocchio [43] and RSV [13] determine importance of a term using TF-IDF based metrics. Haiduc et al. [21] identify the best of four reformulation candidates for any given query using a machine learning model with 28 metrics. All these five studies are highly relevant to ours, and we directly compare with them using experiments. Readers are referred to Section III-E for comparison details.

Other related studies [39, 41, 54] explore graph-based methods for term weighting. Rahman and Roy [39, 41] simply use TextRank on *change request texts* for suggesting initial queries for concept location. Yao et al. [54] build a term augmented tuple graph and use a random walk approach to reformulate queries for structured bibliographic DBLP Data (i.e., non-source code). Ours is significantly different from these studies in the sense that we reformulate the initial queries not only by employing our term weighting method–CodeRank for *source code*, but also by applying source code document structures, query quality analysis and machine learning. Besides, their reported best performance (i.e., 58%–62% query improvement over baseline [41]) is quite lower than our performance (i.e., 71%, even with difficult queries). Given that reformulation is often performed on the initial queries, our technique can potentially complement theirs. Howard et al. [25] map method signatures to associated comments for query reformulation, and thus, might not work well with source code without comments. Rahman and Roy [40] exploit crowd sourced knowledge for query reformulation, and their method is also subject to the availability of a third party information source. Thus, while earlier studies adopt various methodologies or information sources, our technique not only employs a novel and promising term weight –*CodeRank*, but also exploits structures of the source documents for identifying the best reformulation to a given query for improved concept location.

VI. Conclusion & Future Work

To summarize, we propose a novel technique–ACER–for improved query reformulation for concept location. It takes an initial query as input, identifies appropriate search terms from the source code using a novel term weight, and then suggests the best reformulation to the initial query using document structures, query quality analysis and machine learning. Experiments with 1,675 baseline queries from eight systems report that our technique can improve 71% of the baseline queries and preserve 26% of them, which are highly promising. Comparison with five closely related approaches including the state-of-the-art not only validates our empirical findings but also demonstrates the high potential of our technique. In future, we plan to apply our term weighting method, CodeRank, to other SE text retrieval tasks involving source code such as bug localization and traceability recovery.

Acknowledgement

This research work was supported in part by the Natural Sciences and Engineering Research Council of Canada (NSERC) and the Internal Dean's Scholarship of University of Saskatchewan, Canada.

References

[1] ACER experimental data. URL https://goo.gl/ZkaNvd.

[2] Debbugger source lookup does not work with variables. URL https://bugs.eclipse.org/bugs/show_bug.cgi?id=31110.

[3] Example code snippet. URL https://goo.gl/WSZHiC.

[4] Samurai prefix and suffix list. URL https://hiper.cis.udel.edu/Samurai.

[5] Resampling. URL http://www.creative-wisdom.com/teaching/WBI/resampling.shtml.

[6] Stop word list. URL https://code.google.com/p/stop-words.

[7] A. Bachmann and A. Bernstein. Software Process Data Quality and Characteristics: A Historical View on Open and Closed Source Projects. In *Proc. IWPSE*, pages 119–128, 2009.

[8] B. Bassett and N. A. Kraft. Structural Information based Term Weighting in Text Retrieval for Feature Location. In *Proc. ICPC*, pages 133–141, 2013.

[9] R. Blanco and C. Lioma. Graph-based Term Weighting for Information Retrieval. *Inf. Retr.*, 15(1):54–92, 2012.

[10] S. Brin and L. Page. The Anatomy of a Large-Scale Hypertextual Web Search Engine. *Comput. Netw. ISDN Syst.*, 30(1-7):107–117, 1998.

[11] D. Carmel and E. Yom-Tov. *Estimating the Query Difficulty for Information Retrieval*. Morgan & Claypool, 2010.

[12] D. Carmel, E. Yom-Tov, A. Darlow, and D. Pelleg. What Makes a Query Difficult? In *Proc. SIGIR*, pages 390–397, 2006.

[13] C. Carpineto and G. Romano. A Survey of Automatic Query Expansion in Information Retrieval. *ACM Comput. Surv.*, 44(1):1:1–1:50, 2012.

[14] E. Enslen, E. Hill, L. Pollock, and K. Vijay-Shanker. Mining Source Code to Automatically Split Identifiers for Software Analysis. In *Proc. MSR*, pages 71–80, 2009.

[15] G. W. Furnas, T. K. Landauer, L. M. Gomez, and S. T. Dumais. The Vocabulary Problem in Human-system Communication. *Commun. ACM*, 30(11):964–971, 1987.

[16] G. Gay, S. Haiduc, A. Marcus, and T. Menzies. On the Use of Relevance Feedback in IR-based Concept Location. In *Proc. ICSM*, pages 351–360, 2009.

[17] S. Haiduc and A. Marcus. On the Use of Domain Terms in Source Code. In *Proc. ICPC*, pages 113–122, 2008.

[18] S. Haiduc and A. Marcus. On the Effect of the Query in IR-based Concept Location. In *Proc. ICPC*, pages 234–237, 2011.

[19] S. Haiduc, G. Bavota, R. Oliveto, A. De Lucia, and A. Marcus. Automatic Query Performance Assessment During the Retrieval of Software Artifacts. In *Proc. ASE*, pages 90–99, 2012.

[20] S. Haiduc, G. Bavota, R. Oliveto, A. Marcus, and A. De Lucia. Evaluating the Specificity of Text Retrieval Queries to Support Software Engineering Tasks. In *Proc. ICSE*, pages 1273–1276, 2012.

[21] S. Haiduc, G. Bavota, A. Marcus, R. Oliveto, A. De Lucia, and T. Menzies. Automatic Query Reformulations for Text Retrieval in Software Engineering. In *Proc. ICSE*, pages 842–851, 2013.

[22] S. Haiduc, G. De Rosa, G. Bavota, R. Oliveto, A. De Lucia, and A. Marcus. Query Quality Prediction and Reformulation for Source Code Search: the Refoqus Tool. In *Proc. ICSE*, pages 1307–1310, 2013.

[23] E. Hill, L. Pollock, and K. Vijay-Shanker. Automatically Capturing Source Code Context of NL-queries for Software Maintenance and Reuse. In *Proc. ICSE*, pages 232–242, 2009.

[24] E. Hill, S. Rao, and A. Kak. On the Use of Stemming for Concern Location and Bug Localization in Java. In *Proc. SCAM*, pages 184–193, 2012.

[25] M. J. Howard, S. Gupta, L. Pollock, and K. Vijay-Shanker. Automatically Mining Software-based, Semantically-Similar Words from Comment-Code Mappings. In *Proc. MSR*, pages 377–386, 2013.

[26] K. S. Jones. A Statistical Interpretation of Term Specificity and Its Application in Retrieval. *Journal of Documentation*, 28(1):11–21, 1972.

[27] T. Kaneishi and T. Dohi. Parametric Bootstrapping for Assessing Software Reliability Measures. In *Proc. PRDC*, pages 1–9, 2011.

[28] K. Kevic and T. Fritz. Automatic Search Term Identification for Change Tasks. In *Proc. ICSE*, pages 468–471, 2014.

[29] K. Kevic and T. Fritz. A Dictionary to Translate Change Tasks to Source Code. In *Proc. MSR*, pages 320–323, 2014.

[30] D. Liu, A. Marcus, D. Poshyvanyk, and V. Rajlich. Feature Location via Information Retrieval based Filtering of a Single Scenario Execution Trace. In *Proc. ASE*, pages 234–243, 2007.

[31] A. Marcus, A. Sergeyev, V. Rajlich, and J.I. Maletic. An Information Retrieval Approach to Concept Location in Source Code. In *Proc. WCRE*, pages 214–223, 2004.

[32] R. Mihalcea and P. Tarau. Textrank: Bringing Order into Texts. In *Proc. EMNLP*, pages 404–411, 2004.

[33] L. Moreno, J. J. Treadway, A. Marcus, and W. Shen. On the Use of Stack Traces to Improve Text Retrieval-based Bug Localization. In *Proc. ICSME*, pages 151–160, 2014.

[34] L. Moreno, G. Bavota, S. Haiduc, M. Di Penta, R. Oliveto, B. Russo, and A. Marcus. Query-based Configuration of Text Retrieval Solutions for Software Engineering Tasks. In *Proc. ESEC/FSE*, pages 567–578, 2015.

[35] C. Parnin and A. Orso. Are Automated Debugging Techniques Actually Helping Programmers? In *Proc. ISSTA*, pages 199–209, 2011.

[36] L. Ponzanelli, A. Bacchelli, and M. Lanza. Seahawk: Stack Overflow in the IDE. In *Proc. ICSE*, pages 1295–1298, 2013.

[37] L. Ponzanelli, G. Bavota, M. Di Penta, R. Oliveto, and M. Lanza. Mining StackOverflow to Turn the IDE into

a Self-confident Programming Prompter. In *Proc. MSR*, pages 102–111, 2014.

[38] M. M. Rahman and C. K. Roy. On the Use of Context in Recommending Exception Handling Code Examples. In *Proc. SCAM*, pages 285–294, 2014.

[39] M. M. Rahman and C. K. Roy. TextRank Based Search Term Identification for Software Change Tasks. In *Proc. SANER*, pages 540–544, 2015.

[40] M. M. Rahman and C. K. Roy. QUICKAR: Automatic Query Reformulation for Concept Location Using Crowdsourced Knowledge. In *Proc. ASE*, pages 220–225, 2016.

[41] M. M. Rahman and C. K. Roy. STRICT: Information Retrieval Based Search Term Identification for Concept Location. In *Proc. SANER*, pages 79–90, 2017.

[42] P. C. Rigby and M.P. Robillard. Discovering Essential Code Elements in Informal Documentation. In *Proc. ICSE*, pages 832–841, 2013.

[43] J.J. Rocchio. *The SMART Retrieval System—Experiments in Automatic Document Processing*. Prentice-Hall, Inc.

[44] 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 *Proc. ICSM*, pages 512–515, 2013.

[45] R. K. Saha, M. Lease, S. Khurshid, and D. E. Perry. Improving Bug Localization using Structured Information Retrieval. In *Proc. ASE*, pages 345–355, 2013.

[46] G. Salton, A. Wong, and C. S. Yang. A Vector Space Model for Automatic Indexing. *Commun. ACM*, 18(11): 613–620, 1975.

[47] T. Savage, M. Revelle, and D. Poshyvanyk. FLAT3: Feature Location and Textual Tracing Tool. In *Proc. ICSE*, pages 255–258, 2010.

[48] D. Shepherd, Z. P. Fry, E. Hill, L. Pollock, and K. Vijay-Shanker. Using Natural Language Program Analysis to Locate and Understand Action-Oriented Concerns. In *Proc. ASOD*, pages 212–224, 2007.

[49] B. Sisman and A. C. Kak. Assisting Code Search with Automatic Query Reformulation for Bug Localization. In *Proc. MSR*, pages 309–318, 2013.

[50] M. Tan, L. Tan, S. Dara, and C. Mayeux. Online Defect Prediction for Imbalanced Data. In *Proc. ICSE*, volume 2, pages 99–108, 2015.

[51] C. Vassallo, S. Panichella, M. Di Penta, and G. Canfora. CODES: Mining Source Code Descriptions from Developers Discussions. In *Proc. ICPC*, pages 106–109, 2014.

[52] I. Vessey. Expertise in Debugging Computer Programs: An Analysis of the Content of Verbal Protocols. *TSMC*, 16(5):621–637, 1986.

[53] J. Yang and L. Tan. Inferring Semantically Related Words from Software Context. In *Proc. MSR*, pages 161–170, 2012.

[54] J. Yao, B. Cui, L. Hua, and Y. Huang. Keyword Query Reformulation on Structured Data. In *Proc. ICDE*, pages 953–964, 2012.

[55] T. Yuan, D. Lo, and J. Lawall. Automated Construction of a Software-specific Word Similarity Database. In *Proc. CSMR-WCRE*, pages 44–53, 2014.