STRICT: Information Retrieval Based Search Term Identification for Concept Location

Mohammad Masudur Rahman   Chanchal K. Roy

Department of Computer Science, University of Saskatchewan, Canada

{masud.rahman, chanchal.roy}@usask.ca

Abstract—**During maintenance, software developers deal with numerous change requests that are written in an unstructured fashion using natural language. Such natural language texts illustrate the change requirement involving various domain related concepts. Software developers need to find appropriate search terms from those concepts so that they could locate the possible locations in the source code using a search technique. Once such locations are identified, they can implement the requested changes there. Studies suggest that developers often perform poorly in coming up with good search terms for a change task. In this paper, we propose a novel technique– STRICT–that automatically identifies suitable search terms for a software change task by analyzing its task description using two information retrieval (IR) techniques– TextRank and POSRank. These IR techniques determine a term's importance based on not only its co-occurrences with other important terms but also its syntactic relationships with them. Experiments using 1,939 change requests from eight subject systems report that STRICT can identify better quality search terms than baseline terms from 52%–62% of the requests with 30%–57% Top-10 retrieval accuracy which are promising. Comparison with two state-of-the-art techniques not only validates our empirical findings and but also demonstrates the superiority of our technique.**

Index Terms—**Concept location, TextRank, POSRank, search term identification, information retrieval**

I. INTRODUCTION

During maintenance, software developers deal with numerous change requests as a part of software change implementation. Identification of exact source location (i.e., also called *concept location*) during software code change (e.g., new feature addition, bug fixation) is a major challenge, even for a medium sized system [46]. Change requests are often made by the users of a software system, and these requests are generally written using unstructured natural language texts [25]. While the software users might be familiar with the application domain of a software system, they generally lack the idea of how a particular software feature is implemented in the source code. Hence, a change request from them generally involves one or more *"high level"* concepts (i.e., functional requirement) related to the application domain. A developer needs to map these concepts to the relevant source locations within the project for implementing the change [25, 28]. Such mapping is possibly trivial for the developer who has substantial knowledge on the target project. However, developers involved in the maintenance might not be aware of the low-level architecture of the software project, and the design documents might not be available either [10, 33]. Thus, they often experience difficulties in identifying the exact source locations (e.g., methods) to be changed. The mapping task generally starts with a search within the project which requires one or more suitable search terms. Previous studies [12, 24, 27] suggest that on average, developers perform poorly in coming up with good search terms for a change task regardless of their experience. For example, Kevic and Fritz [24] report from a user study that only 12.20% of the search terms from the developers were able to retrieve relevant results. According to Furnas et al. [12], there is a small chance (10%–15%) that developers guess the exact words used in the source code. One way to help them overcome this challenge is to automatically suggest suitable search terms for the change task at hand, and our work addresses this research problem.

Existing studies from relevant literature apply various lightweight heuristics [24] and query reformulation strategies [13, 18, 42, 46]. They also perform different query quality analyses [14, 15, 17, 18] and data mining activities [22, 25, 46]. However, most of these approaches expect a developer to provide the initial search query which they can improve upon. Unfortunately, preparing such a query is often a non-trivial task for the developers as well [12, 24]. One can think of the whole change request as the initial query. However, this might lead the developers to major query reformulation tasks [13] or suboptimal solutions. Kevic and Fritz propose the only heuristic model for automatically identifying initial search terms for a change task where they consider different heuristics associated with *frequency, location, parts of speech* and *notation* of the terms from the change request. Although their model is found promising according to the preliminary evaluation, it suffers from two major limitations. First, their model is neither trained using a large dataset (i.e., uses only 20 change requests) nor cross-validated using the change requests from multiple software projects (i.e., uses only one project). Thus, the model is yet to be matured and reliable. Second, *tf–idf* is the most dominating feature of their model which is subject to the size of the change request dataset for *inverse document frequency (idf)* calculation. Hence, their model is likely to be affected significantly by the size of the dataset. That is, it might require frequent re-training to keep itself useful for search term identification. Thus, an approach that overcomes such limitations and yet can identify appropriate search terms from a given change request is warranted.

In this paper, we propose a novel query recommendation technique–STRICT–that automatically identifies and recommends good quality search terms from a change request for concept location. We determine importance of each term by employing two graph-based term weighting algorithms
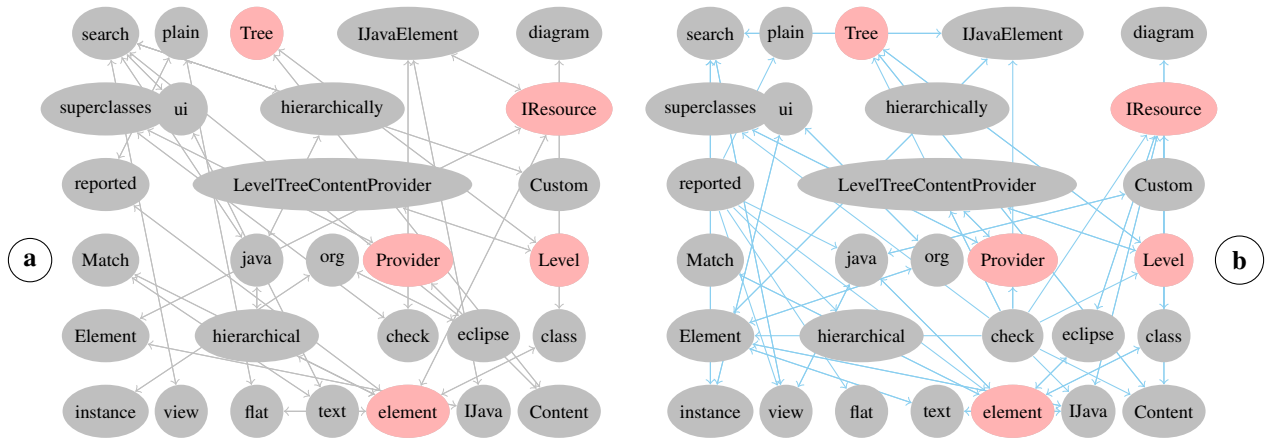

Fig. 1. Text Graphs of change request in Table I – (a) using word co-occurrences, and (b) using syntactic dependencies

(from information retrieval domain)–*TextRank* and *POSRank*–on the content of a change request, and then suggest the most important terms as the search query. Both TextRank and POSRank are adaptations of *PageRank* algorithm [7] for natural language texts, and they extract the most important terms from a change request by analyzing co-occurrences [30] and syntactic dependencies [6, 30] among the terms respectively. Unlike Kevic and Fritz [24], they are not subject to the size of subject systems under study, and also do not require any training. In fact, our technique is highly suited for the target research problem from several perspectives. First, our technique considers the content of a change request as a text graph (e.g., Fig. 1) based on either term co-occurrences or syntactic dependencies rather than plain text. Thus, it has a higher potential for revealing important semantic relationships among the terms which might lead one to choosing appropriate search terms. Second, both TextRank and POSRank determine importance of a term in the graph (i.e., change request) not only based on its connectivity but also by considering the weights (i.e., importance) of the connected terms. That is, a term would be considered important only if it connected to other important terms [20, 30]. Thus, our technique has a better chance of locating the chunk of important terms from the change request which can be suggested as the search terms.

Table I shows an example change request from `eclipse.jdt.ui` system that reports a concern with custom search result display in Eclipse IDE. Our technique–STRICT–first converts the textual content of the request into two text graphs by capturing (a) co-occurrences of the terms in the request (i.e., Fig. 1-(a)) and (b) dependencies among the terms due to their parts of speech (POS) (i.e., Fig. 1-(b)) respectively. Then, it identifies the most important terms by recursively analyzing the topological characteristics (i.e., term connectivity) of both graphs. STRICT returns the following Top-5 search terms (i.e., highlighted, Fig. 1)– *'element', 'IResource', 'Provider', 'Level'* and *'Tree'*–which return the first correct result at the Top-1 position. On the contrary, the baseline queries– *Title*, *Description* and *Title+Description*–return such result at 559th, 71st and 211th positions. Thus, our suggested search terms (1) can provide a meaningful

TABLE I
AN EXAMPLE CHANGE REQUEST (ISSUE #:303705, ECLIPSE.JDT.UI)

Field	Content
Title	[search] Custom search results not shown hierarchically in the java search results view
Descripton	Consider an instance of org.eclipse.search.ui.text.Match with an element that is neither an IResource nor an IJavaElement. It might be an element in a class diagram, for example. When such an element is reported, it will be shown as a plain, flat element in the otherwise hierarchical java search results view. This is because the LevelTreeContentProvider and its superclasses only check for IJavaElement and IResource.

starting point for code search during code change, and (2) can potentially reduce the query reformulation efforts spent by the developers. It should be noted that this paper is a significantly extended version of our preliminary work on search term identification [35]. While the earlier work explores the potential of *TextRank* using a limited dataset, this work (1) extends that idea by applying another appropriate term-weighting technique–*POSRank*, (2) proposes a novel search term ranking algorithm, and (3) then evaluates and validates the technique extensively using a larger dataset [1].

Experiments using 1,939 change requests from eight subject systems (i.e., three *Apache* systems and five *Eclipse* systems) report that our technique–STRICT–can provide better quality search terms than 52%–62% of the baseline queries which is highly promising according to relevant literature [18, 32]. Our suggested queries can retrieve relevant source code files for 30%–57% of the change tasks with about 30% mean average precision@10 where the first relevant file is mostly found within the Top-4 positions, which are also promising [2, 5]. Comparison with two state-of-the-art techniques–Kevic and Fritz [24] and Rocchio [39]– not only validates our empirical findings but also demonstrates the superiority of our technique. We thus make the following contributions:

- A novel and promising search term identification technique–STRICT–that identifies good quality search terms for a change task from its change request.
- Comprehensive evaluation of the technique using 1,939 change requests from eight subject systems and four state-of-the-art performance metrics.

- Comprehensive validation of the technique using comparisons with two state-of-the-art techniques.

II. GRAPH BASED TERM-WEIGHTING

In information retrieval (IR), natural language text is often transformed into a graph where unique words are denoted as vertices, and meaningful relations among those words are represented as the edges [30]. Such relation can be statistical, syntactic or semantic in nature [6]. In our research, we represent a software change request as a text graph where we consider both statistical and syntactic relations among words as the edges in the graph. To capture statistical relation, we consider co-occurrence of the words within a fixed window (e.g., *window size = 2*) across all sentences from the request. For example, Fig. 1-(a) shows the text graph for the showcase change request (i.e., Table I) based on co-occurrence relationships among the words within a window of two. In order to capture syntactic relation, we consider grammatical modification of words from the request using Jespersen's Rank Theory [23]. According to Jespersen, words belonging to different parts of speech from a sentence can be provided with three major ranks– primary (i.e., nouns), secondary (i.e., verbs, adjectives) and tertiary (i.e., adverbs)– where a word from a higher rank modifies another word from the same or lower ranks. We thus encode such modification relations into edges in the text graph. Fig. 1-(b) shows the text graph of the example change request (i.e., Listing 1) based on grammatical dependencies among the words. Once text graphs are developed, we apply two adapted versions of the popular algorithm by Brin and Page (for web link analysis)– *PageRank* [7]–for term weighting. In particular, we identify the most important words from the text graphs by exploiting their topological properties (e.g., connectivity) in the graphs.

III. STRICT: PROPOSED TECHNIQUE FOR SEARCH TERM IDENTIFICATION FROM A CHANGE REQUEST

Given that appropriate search term identification is a major challenge for the developers and existing studies are limited in certain contexts, we introduce a novel IR-based technique. Fig. 2 shows the schematic diagram of our proposed technique– STRICT–for search term identification and suggestion from a change request. We first turn a software change request into two text graphs (e.g., Fig. 1) based on word co-occurrence and grammatical modification of words. Then, we employ two graph-based term-weighting algorithms–*TextRank* and *POS-Rank*–on those graphs, estimate term weights, and identify the suitable search terms for the change task. In this section, we discuss different steps of our technique as follows:

A. Data Collection

Our technique accepts user-provided texts of a change request as the input (i.e., Step 1, Fig. 2), and returns a ranked list of search terms as the output (i.e., Step 8, Fig. 2). We collect change requests from two popular bug tracking systems–*BugZilla* and *JIRA*. Each change request is submitted as a semi-structured report written using natural language texts, and it contains several fields such as *Issue ID* (e.g., 303705), *Product* (e.g., JDT), *Component* (e.g., UI), *Title* and *Description*. We extract the last two fields from each report for analysis, as was also done by literature [24]. *Title* summarizes a requested change task whereas *Description* contains the user's detailed explanation of the task in natural language texts.

B. Text Preprocessing

We analyze *Title* and *Description* of a software change request, and perform several preprocessing steps on them (i.e., Step 2, Fig. 2). We consider *sentence* as a logical unit for the change request texts, and collect each of the sentences from both fields. Then we perform standard natural language preprocessing (i.e., stop word removal and splitting of dotted terms) on each of the sentences, and extract the candidate search terms. A dotted term (e.g., `org.eclipse.ui.part`) often contains multiple technical artifacts, and splitting helps one to analyze each of them in isolation [11]. We also split each camel case (e.g., `createPartControl`) term into simpler terms (i.e., `create`, `Part` and `Control`) and keep both term types for our analysis [11, 18]. It should be noted that we avoid term stemming (i.e., extracting root form of a given term) since it degrades the performance of our technique, as was also reported by Kevic and Fritz [24].

C. Text Graph Development

Using Word Co-occurrence: After preprocessing steps, we get a list of sentences from each change request where each of the sentences contains an ordered list of candidate search terms. We then use those sentences to develop a *text graph* (e.g., Fig. 1-(a)) for the change request (Step 3, Fig. 2). In the text graph, each unique term is represented as a vertex and the *co-occurrence* of terms in the sentence is denoted as the edges among the vertices. The underlying idea is that all the terms that co-occur in the text within a fixed window have some relationships or dependencies [6, 30]. For example, if we consider the sentence– *"Custom search results not shown hierarchically in the java search results view"*–from the example request texts (i.e., Table I), the preprocessed version forms an ordered list of terms– *"Custom search hierarchically java search view."* Please note that the transformed sentence contains several phrases such as *"custom search"* and *"search view"*, and the terms in those phrases are semantically dependent on each other for comprehensive meaning. Use of term co-occurrence captures such dependencies in a statistical sense. We thus consider a *sliding window* of *window size=2* (as recommended by Mihalcea and Tarau [30]) as a semantic unit of words, and derive the following relationships: *Custom⟷search*, *search⟷hierarchically*, *hierarchically⟷java*, *java⟷search* and *search⟷view*. Then such relationships are encoded into the connecting edges between the corresponding vertices in the text graph (i.e., Fig. 1-(a)).

Using POS Dependence: Term co-occurrence models relationship between terms statistically which might not be always effective for determining term-weight (i.e., term's
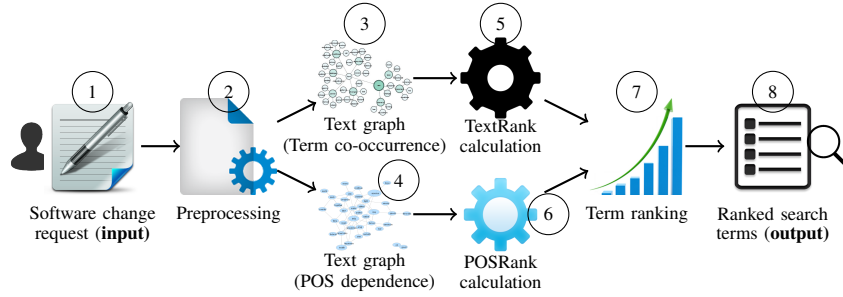
Fig. 2. Schematic diagram of the proposed technique–STRICT

importance). We thus apply another type of relationship– syntactic dependencies– among the terms based on grammatical modification. According to Jespersen's Rank Theory [23], words from a sentence can be provided with three major ranks– primary (i.e., nouns), secondary (i.e., verbs, adjectives), and tertiary (i.e., adverbs). Jespersen [23] suggests that a word from a higher rank generally defines (i.e., modifies) another word from the same or lower ranks in a sentence. Thus, a noun can modify only another noun whereas a verb can modify another noun, verb or adjective but not an adverb. We consider this principle of grammatical modification of words, and represent such dependencies as directed edges in the text graph (i.e., Step 4, Fig. 2). We first annotate each of the sentences from a change request using Stanford POS tagger [44], and organize the annotated words into various ranks. For instance, the example statement—*"element reported plain flat element hierarchical java search view"*–can be organized into two ranks–*primary* (*"search", "view", "java", "element"*), and *secondary* (*"plain", "flat", "hierarchical"*, and *"reported"*). We derive the following relationships based on their grammatical modifications– *search⟷view*, *view⟷java*, *java⟷element*, *reported⟶search*, *reported⟶view*, *reported⟶java*, *reported⟶element*, *reported⟶plain*, *reported⟶flat*, *reported⟶hierarchical*, and, then encode them into connecting edges in the text graph (i.e., Fig. 1-(b)).

D. TextRank (TR) Calculation

Once text graph (i.e., using co-occurrence) for a change request is developed, we consider it as a regular connected network, and apply a popular graph-based algorithm–PageRank [7]–for ranking its nodes (i.e., terms) (Step 5, Fig. 2). PageRank was originally proposed by Brin and Page for web link analysis, and the algorithm exploits topological properties of a graph to estimate the weight (i.e., importance) of each of the vertices. TextRank is an adaptation of PageRank for text graph. It analyzes the connectivity (i.e., connected neighbours and their weights) of each term v_i in the graph recursively, and then calculates the term's weight, $TR(v_i)$, as follows:

$$TR(v_i) = (1 - \phi) + \phi \sum_{j \in V(v_i)} \frac{TR(v_j)}{|V(v_j)|} \quad (0 \leq \phi \leq 1)$$

Here, $V(v_j)$ and ϕ denote node list connected to v_i and *dumping factor* respectively. In the text graph (e.g., Fig. 1-(a)), co-occurrences among terms are represented as bi-directional connecting edges between the nodes. In the context of web surfing, dumping factor, ϕ, is considered as the probability of randomly choosing a web page by the surfer, and $1 - \phi$ is the probability of jumping off that page. Mihalcea and Tarau [30] use a heuristic value of $\phi = 0.85$ for natural language texts in the context of keyword extraction, and we also use the same value for our TextRank calculation. We initialize each of the terms in the graph with a default value of 0.25, and run an iterative version of the algorithm [7]. It should be noted that the initial value of a term does not affect its final score [30]. The computation iterates until the scores of all the terms converge below a certain threshold or it reaches the maximum iteration limit (i.e., 100 as suggested by Blanco and Lioma [6]). As Mihalcea and Tarau [30] suggest, we use a heuristic threshold of 0.0001 for the convergence checking.

TextRank applies the underlying mechanism of a recommendation (i.e., voting) system, where a term (e.g., *"Custom"*) recommends (i.e., votes) another term (e.g., *"search"*) if the second term complements the semantics of the first term in any way (e.g., *"Custom search"*) [30]. The algorithm captures recommendation for a term by analyzing its connected edges (i.e., both incoming and outgoing) in the text graph (e.g., Fig. 1-(a)) with other terms both in local (i.e., same sentence) and global (i.e., entire task description) contexts, and thus determines importance of that term. Once computation is over, each of the terms in the graph is found with a final score which is considered to be the weight or importance of that term within the user provided texts from the change request.

E. POSRank (POSR) Calculation

While TextRank operates on a text graph based on word co-occurrence (e.g., Fig. 1-(a)), POSRank determines term-weight by operating on the text graph (e.g., Fig. 1-(b)) that represents the grammatical dependencies among words as the connecting edges (Step 6, Fig. 2). POSRank is another adaptation of PageRank algorithm [7] for natural language texts. Similar to TextRank, it also analyzes connectivity of each term in the graph but considers the links according to their directions. Incoming links and outgoing links of the term are treated differently. Incoming links represent votes cast for the term by other terms and vice versa. Thus, POSRank $POSR(v_i)$ of each term v_i is calculated as follows:

$$POSR(v_i) = (1 - \phi) + \phi \sum_{j \in In(v_i)} \frac{POSR(v_j)}{|Out(v_j)|} \quad (0 \leq \phi \leq 1)$$

Here $In(v_i)$ and $Out(v_i)$ denote the node lists that are connected to node v_i through incoming and outgoing links respectively. Since the underlying mechanism of PageRank-

based algorithms is recommendation (i.e., votes) from other nodes of the graph, POSRank also determines the weight (i.e., importance) of a term by capturing and analyzing the weights of the incoming links recursively. It should be noted that not only frequent votes but also the votes from other high scored nodes from the graph are essential for a node (i.e., term) to be highly scored (i.e., important). Given the similar topological properties of the text graph using grammatical modifications (i.e., Fig. 1-(b)), we apply the same settings– damping factor (ϕ), iteration count, initial score, and convergence threshold– of TextRank (Section III-D) for POSRank calculation as well.

Algorithm 1 Search Term Identification using IR Methods

1: **procedure** STRICT(CR) ▷ CR: change request
2: $L \leftarrow \{\}$ ▷ list of search terms
3: ▷ collecting task details from the change request
4: $T \leftarrow$ collectTitle(CR)
5: $D \leftarrow$ collectDescription(CR)
6: $TD \leftarrow$ preprocess(combine(T, D))
7: ▷ developing text graphs from the task details
8: $G_{coc} \leftarrow$ developTGUsingCo-occurrence(TD)
9: $G_{pos} \leftarrow$ developTGUsingPOS-dependence(TD)
10: ▷ calculating TextRank and POSRank
11: $TR \leftarrow$ calculateTR(G_{coc})
12: $TR_{norm} \leftarrow$ normalize(sortByValue(TR))
13: $POSR \leftarrow$ calculatePOSR(G_{pos})
14: $POSR_{norm} \leftarrow$ normalize(sortByValue($POSR$))
15: ▷ calculating additional weights for title terms
16: $TT \leftarrow$ assignUniformWeight(T)
17: ▷ determining term importance
18: $CST \leftarrow$ getUniqueTerms(TD)
19: **for** CandidateSearchTerm $CST_i \in CST$ **do**
20:   $S_{TR} \leftarrow TR_{norm}[CST_i]$
21:   $S_{POSR} \leftarrow POSR_{norm}[CST_i]$
22:   $S_{TT} \leftarrow TT[CST_i]$
23:   ▷ calculating final term-weight
24:   $S[CST_i] \leftarrow S_{TR} + S_{POSR} + S_{TT}$
25: **end for**
26: ▷ ranking and then returning Top-K search terms
27: $SST \leftarrow$ sortByFinalTermWeight(S)
28: $L \leftarrow$ getTopKSearchTerms(SST)
29: **return** L
30: **end procedure**

F. Search Term Ranking and Selection

Fig. 2 shows the schematic diagram and Algorithms 1, 2 show the pseudo code of our proposed technique–STRICT– for search term identification from a change request. We first collect *Title* and *Description* of the request submitted by the user, combine them to prepare a complete request text, and then perform standard natural language preprocessing (Lines 3–6, Algorithm 1). Then we develop two text graphs based on co-occurrences and grammatical dependencies among the terms from the preprocessed text (Lines 7–9, Algorithm 1). The goal is to identify the most important terms that could be used as a search query for concept location. We then analyze

the topological properties of both graphs, and determine TextRank and POSRank for each of the terms from the request (Lines 10–14, Algorithm 1, Steps 5, 6, Fig. 2).

Algorithm 2 Candidate Score Normalization Algorithm

1: **procedure** NORMALIZE(R)
2: ▷ R: candidates search terms sorted by scores
3: **for** CandidateSearchTerm $t \in R.keys$ **do**
4:   $R[t] \leftarrow 1 - \frac{position(t)}{size(R)}$
5: **end for**
6: **return** R
7: **end procedure**

Since TextRank and POSRank estimate term importance from different perspectives, we normalize both scores for each of the candidate terms using Algorithm 2. Normalization step can reduce the potential bias of any specific score component. In particular, we sort the candidate terms based on TextRank or POSRank, and derive a score between 0 and 1 for each term by using its position in the sorted list. Such normalization technique is common in the relevant literature [9, 36], and often called as *degree of interest* [9]. Given that title of the request often contains good quality search terms, we assign the highest degree of interest to the candidate terms found in the *Title* field (Lines 15–16). Thus, each of the candidate terms gets three normalized scores from three types of ranking– TextRank, POSRank and title-based heuristic. Please note that such scores for the candidates are calculated by recursively analyzing the votes (i.e., connections) from their surrounding terms in the text graphs. Now, we iterate through the unique candidate search terms, and add up those three scores for each of the candidate terms (Lines 17–25, Algorithm 1). Then we rank the candidates based on their final accumulated scores, and select the Top-K ($K = 10$) candidates as the search terms for the change request (Lines 26–30, Algorithm 1, Step 7, 8, Fig. 2). We also expand the camel case search terms into simpler forms, and keep both forms in the search query [11].

Example: Table II shows a working example of our technique–STRICT–for the showcase change request in Table I. Our technique suggests the following Top-5 terms– *'element', 'IResource', 'Provider', 'Level'* and *'Tree'*–that returns the first correct result at the Top-1 position. We first calculate TextRank and POSRank for each of the candidate terms from the text graphs–Fig. 1-(a) and Fig. 1-(b)– respectively by carefully analyzing their topological properties (Sections III-D, III-E). Table II shows the normalized scores for the top terms. We see that these terms are highly connected in the text graphs, i.e., frequently voted by other important terms across the request text. Our algorithms translate such connectivity into meaningful equivalent scores, and identify the top-scored terms carefully. Please note that none of these terms comes from the title of the request which is often copied and pasted by the developers as the initial query. Besides, the title returns the first correct result at the 559th position. Thus, our suggested terms can significantly reduce the query reformulation efforts spent by the developers, and can provide them with a meaningful starting point for concept location.

TABLE II
STRICT: A WORKING EXAMPLE

Search Term	TR	POSR	TH	Score
element	0.98	0.98	0	1.96
IResource	0.93	1.00	0	1.93
Provider	0.68	0.95	0	1.63
Level	0.83	0.80	0	1.63
Tree	0.78	0.75	0	1.53

TR: TextRank, **POSR:** POSRank, **TH:** Title Heuristic

TABLE III
EXPERIMENTAL DATASET

System	#MD	#CR	System	#MD	#CR
eclipse.jdt.core-4.7.0	64K	292	ecf-170.170	21K	262
eclipse.jdt.debug-4.6.0	16K	63	log4j-1.2.17	3K	60
eclipse.jdt.ui-4.7.0	57K	419	sling-0.1.10	30K	76
eclipse.pde.ui-4.7.0	32K	329	tomcat-7.0.70	24K	438

MD: **M**ethod **D**efinitions, **CR:** **C**hange **R**equests selected for experiments

IV. EXPERIMENT

Given two methods–*pre-retrieval* and *post-retrieval*–in the literature [14, 15], we chose post-retrieval method for the evaluation and validation of our suggested search queries. This method directly evaluates results returned by a query, and thus is more reliable than the other. Besides, past relevant studies [4, 18, 32] also adopted this method for evaluation and validation. We evaluate our technique–STRICT–using 1,939 software change requests with four appropriate performance metrics, and compare with two state-of-the-art techniques [24, 39]. Thus, we answer four research questions as follows:

- **RQ$_1$:** Are our suggested queries significantly better in term of effectiveness than the baseline search queries from the software change requests?
- **RQ$_2$:** How do our suggested search queries actually perform in retrieving correct/relevant results?
- **RQ$_3$:** How effective are the information retrieval algorithms–*TextRank* and *POSRank*–in identifying good quality search terms from a change request?
- **RQ$_4$:** Can our proposed technique–STRICT– outperform the state-of-the-art techniques in identifying good quality search terms from a change request?

A. Experimental Dataset

Data Collection: We collect 1,939 software change requests from eight Java-based subject systems (i.e., five *Eclipse* systems and three *Apache* systems) for our experiments. Table III shows the details of our selected systems. Each of these change requests was marked as *RESOLVED*, and we follow a careful approach in their selection. We first collect the *RESOLVED* change requests from *BugZilla* and *JIRA* bug repositories of those systems, and then map them to the commit history of their corresponding projects at GitHub and SVN. We analyze the commit messages from each project, and look for specific Issue ID (i.e., identifier of a change task) in each of those messages [3]. Then, we include any change request in the experimental dataset only if there exists a corresponding commit in the collected commit history. Such evaluation approach is regularly adopted by the relevant literature [4, 18, 24, 48], and we also follow the same. In order to ensure a fair evaluation, we also discard the change requests from our dataset for which

no source code files (i.e., Java classes) were changed or the relevant code files were missing from the system.

Goldset Development: In GitHub and SVN, we note that any commit that either solves a software bug or implements a feature request generally mentions the corresponding Issue ID in the very title of the commit. We identify such commits from the commit history of each of the selected projects using suitable regular expressions, and select them for our experiments [3]. Then, we collect the *changeset* (i.e., list of changed files) for each of those commit operations, and develop solution set (i.e., *goldset*) for the corresponding change tasks. Thus, for experiments, we collect not only the actual change requests from the reputed subject systems but also their solutions which were applied in practice by the developers [17]. We use several utility commands such as *git, clone, rev-list* and *log* on GitHub and SVN consoles for collecting those information.

Replication: All experimental data and supporting materials are hosted online [1] for replication or third-part reuse.

B. Search Engine

We use a Vector Space Model (VSM) based search engine–*Apache Lucene* [15, 18]–to search files that were changed for implementing the change requests. Search engines generally index the files in a corpus prior to search. Since Lucene indexer is targeted for simple text documents (e.g., news article) and the source code files in our selected projects contain items beyond regular texts, i.e., code segments, we apply limited preprocessing on them. In particular, we extract method bodies from each of the Java classes (i.e., source files), and consider each method as an individual document in the corpus (Table III). We remove all programming keywords and punctuation characters, and split all dotted and camel case tokens [11]. Please note that we avoid stemming of the tokens for aforementioned reasons as described in Section III-B [24]. This preprocessing step transforms the source files into text like files, and the indexer becomes able to perform more effectively, especially in choosing meaningful index terms. Once a search is initiated using a query, the search engine collects relevant documents from the corpus using a *Boolean Search Model*, and then applies a *tf-idf* based scoring technique to return a ranked list of relevant documents. As existing studies suggest [24, 26], we consider the Top-10 results from the search engine for calculating the performance of our suggested queries.

C. Performance Metrics

We choose four performance metrics for evaluation and validation of our suggested search queries. These metrics are frequently used by the relevant literature [18, 31, 40], and thus are highly appropriate for our experiments as well.

Effectiveness (E): It is a measure that approximates the developer's effort for locating a concept in the source code [31]. In short, the measure returns the rank of the first occurred file (i.e., from the gold set) in the search result list. The lower the effectiveness value is, the more effective a query is.

Mean Reciprocal Rank@K (MRR@K): Reciprocal rank@K refers to the multiplicative inverse of the rank of the

TABLE IV
EFFECTIVENESS DETAILS OF STRICT QUERY VS. BASELINE (TITLE)

System	#Queries	Improvement							Worsening							Preserving
		#Improved	Mean	Q1	Q2	Q3	Min.	Max.	#Worsened	Mean	Q1	Q2	Q3	Min.	Max.	#Preserved
ecf	262	**145 (55.34%)**	220	**7**	27	140	1	3,450	100 (38.17%)	648	65	195	815	2	6,417	**17 (6.49%)**
jdt.core	292	**154 (52.74%)**	421	**6**	46	207	1	6,636	122 (41.78%)	1050	112	346	932	3	9,419	**16 (5.48%)**
jdt.debug	63	**45 (71.43%)**	352	**10**	47	155	1	4,401	15 (23.81%)	682	66	195	1,008	9	2,942	**3 (4.76%)**
jdt.ui	419	**229 (54.65%)**	282	**7**	26	108	1	9,851	179 (42.72%)	974	55	222	916	2	8,596	**11 (2.63%)**
pde.ui	329	**169 (51.37%)**	163	**10**	34	125	1	2,746	128 (38.91%)	1033	87	330	1,272	7	7,602	**32 (9.72%)**
log4j	60	**30 (50.00%)**	74	**7**	13	73	1	788	20 (33.33%)	197	47	124	296	5	630	**10 (16.67%)**
sling	76	**46 (60.53%)**	257	**7**	24	236	1	2,849	24 (31.58%)	501	25	141	564	7	2,904	**6 (7.89%)**
tomcat70	438	**292 (66.67%)**	152	**4**	14	100	1	5,846	128 (29.22%)	555	70	229	675	3	5,918	**18 (4.11%)**
	Total=1,939	**Avg = 57.84%**							Avg = 34.94%							Avg = 7.22%

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 relevant document in the search result, Q_i= Rank value for i^{th} quartile of all result ranks

TABLE V
COMPARISON OF STRICT'S EFFECTIVENESS WITH BASELINE QUERIES

Query Pairs	Improved	Worsened	p-value	Preserved	MRD
STRICT vs. Title	**57.84%**	34.94%	*(<0.001)	**7.22%**	-147
STRICT vs. Title (10 keywords)	**62.49%**	32.26%	*(<0.001)	**5.25%**	-201
STRICT vs. Description	**53.84%**	38.21%	*(<0.001)	**7.95%**	-329
STRICT vs. (Title + Description)	**52.36%**	39.94%	*(<0.001)	**7.70%**	-265

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

first correctly returned changed file (i.e., from gold set) within the Top-K results. Mean Reciprocal Rank@K (MRR@K) averages such measures for all change requests.

Mean Average Precision@K (MAP@K): Precision@K calculates precision at the occurrence of every single relevant result in the ranked list. Average Precision@K (AP@K) averages the precision@K for all relevant results in the list for a search query. Thus, Mean Average Precision@K is calculated from the mean of average precision@K for all queries.

Top-K Accuracy: It refers to the percentage of the change requests for which at least one changed file is correctly returned within the Top-K results by the search queries.

D. Evaluation of STRICT

We conduct experiments using 1,939 change requests from eight subject systems (Table III) where the above four metrics (Section IV-C) are applied to performance evaluation. We run each of our suggested queries with a search engine (Section IV-B), check their results against the collected gold set files (Section IV-A), and then compare with baseline queries from those requests. In this section, we discuss our evaluation details and answer **RQ₁**, **RQ₂**, and **RQ₃** as follows.

Answering RQ₁–Comparison with Baseline Queries: Developers often copy and paste the text from a software change request on an ad-hoc basis for searching the source code that needs to be changed. Thus, change requests can be considered as the baseline queries for evaluation or validation of our suggested queries [4, 18, 24]. We consider three types of baseline queries from the fields of each request–*Title, Description*, and *Title+Description*, run them against our search engine, and then collect the rank of the first correctly returned result for each of those queries. We also collect similar ranks returned by our suggested queries for each of those change requests, and compare with that of baseline queries. Tables V and IV report our comparative analysis. From Table V, we see that our suggested queries are more effective than the baseline queries. Our queries provide better ranks than the baseline

queries for 52% -62% of the cases while preserving the quality of 5%–8% which are promising according to the relevant literature [18, 32]. They provide relatively lower ranks than baseline for 35% of the requests on average. We also compare the query improvement ratios and worsening ratios by our technique using statistical significance test– *Mann-Whitney U (MWU) test*, and found that our improvements are significantly higher (i.e., all $p-values < 0.05$, Table V) with each type of baseline queries. For example, when compared with *Title* from the request, our queries provide 58% improvement ratio on average with 35% worsening ratio. More interestingly, when first 10 keywords (i.e., average size of our request titles) are chosen from *Title* like we suggest Top-10 search terms, the improvement ratio (i.e., 62%) is almost *twice* the worsening ratio (i.e., 32%) by our suggested queries, which demonstrates the high potential of our technique. Mean Rank Difference (MRD) between our provided queries and baseline queries is negative (e.g., -329) which suggests that our queries generally return the goldset files at upper positions of the result list than the baseline queries. Table IV provides further details on the *effectiveness* of our queries. We see that our queries provide better ranks (i.e., 50%–71%) than the baseline ranks of *Title* for each of the eight subject systems, which is promising. More importantly, the 25% quantile (Q_1) of the improved ranks for each of the subject systems is $Q_1 \leq 10$. That means, for one-fourth of the improved cases from each system, STRICT provides the relevant results within the Top-10 positions. When we consider all returned ranks (i.e., improved, worsened and preserved) for those systems, we found that 15%–33% of our first correct ranks are within Top-10 positions whereas such statistic for the baseline queries is only 02%–11%, which demonstrates the potential of our technique.

Earlier studies report significant benefit in including source code tokens in the search query during bug localization [31] and feature location [4]. We also thus compare with code-only baseline queries where each query comprises of camel case tokens extracted from the change request. We found that our queries provide better ranks than 63% of the code-only baseline queries while returning lower ranks for only 29%. We manually investigated, and found that 37% of our baseline queries do not contain any source code tokens, which possibly explains such finding. Thus, while source code tokens might make a good search query, their presence in the change request cannot be guaranteed and thus, queries based on them could also be limited. On the contrary, we suggest relatively better

queries from the careful analysis of any available textual information in the change request. We also manually investigate the worsened queries by STRICT and corresponding requests, and found two important observations. First, the extent of our rank worsening is relatively smaller than that of rank improvement. Second, most of those requests contain structural content (e.g., stack traces), and STRICT could have performed even better if such structures were properly incorporated in the text graphs.

Thus, to answer **RQ$_1$**, our suggested queries are significantly more effective than baseline queries from the change requests. They provide better ranks than baseline queries for 52%–62% of the requests which is promising.

Answering RQ$_2$–Retrieval Performance: Although our suggested queries provide significantly better ranks than baseline queries in RQ$_1$, detailed retrieval performance could be another indicator of their quality. We thus run each of our queries, analyze the Top-10 results (i.e., as many existing studies do [18, 24, 43]), and calculate Top-K accuracy, mean average precision@K and mean reciprocal rank@K of our technique. Table VI, Fig. 3 and Fig. 4 report our performance. From Table VI, we see that our queries perform correctly for 45% of the change requests with 28% mean average precision@10 and a mean reciprocal rank@10 of 0.28. That is, one can get the relevant files (i.e., that need to be changed) with our queries within the Top-4 results for 45% of the times, which is promising. Such performance is also comparable to other existing findings from the literature [2, 4, 8]. Although our retrieval performance is not much higher, our technique–STRICT–has more potential than the baseline queries for practical usage. It achieves relatively higher performance with a limited number of search terms. For example, baseline queries–*Description* and *Title+Description*–perform close (i.e., 38%–41% Top-10 accuracy) to ours (i.e., 45% Top-10 accuracy). However, their performance is achieved at the cost of about 150 terms in each query which is extremely difficult for any developer to reformulate or choose keywords from. On the other hand, 5–10 suggested terms from STRICT provide 40%–45% Top-10 accuracy which demonstrates their higher potential. From the viewpoint of reformulation efforts, *Title* can be considered as a potential baseline opponent, and we compare with this type of queries extensively. We perform statistical significance test, and found that our performance is significantly higher than the baseline query–*Title*–for all three performance metrics–mean average precision@10 (i.e., $p - value : 0.010 < 0.05$), mean reciprocal rank@10 (i.e., $p - value : 0.007 < 0.05$) and Top-10 accuracy (i.e., $p - value : 0.007 < 0.05$). The box plots in Fig. 3 also demonstrate that STRICT performs better than the baseline in each of the measures with higher medians. We also investigate how Top-K values affect the retrieval performance of the queries where K takes up values from 10 to 100. From Fig. 4, we see that STRICT provides a Top-100 accuracy of 77% whereas such statistics for the baseline queries are 66%–70%. More importantly, our accuracy measures always remained significantly higher than the baseline measures– *Title* (i.e., $p - value : 0.012 < 0.05$), *Description* (i.e., $p - value : 0.005 < 0.05$), and *Title+Description* (i.e.,

TABLE VI
RETRIEVAL PERFORMANCE OF STRICT QUERIES

Query	#Keywords	MAP@10	MRR@10	Top-10 Accuracy
Title	10	**18.40%**	**0.18**	**31.14%**
Title (10 keywords)	09	16.63%	0.16	27.95%
Description	146	22.60%	0.23	38.00%
Title + Description	156	**24.54%**	**0.24**	**41.40%**
Title + Description (Code tokens only)	15	19.98%	0.20	29.51%
Title + Description (Without code tokens)	135	18.87%	0.18	34.03%
STRICT$_{T3}$	03	17.93%	0.17	31.32%
STRICT$_{T5}$	05	24.38%	0.24	39.90%
STRICT	10	**28.09%***	**0.28***	**45.34%***

* = Statistically significant difference between suggested and baseline (i.e., title) queries
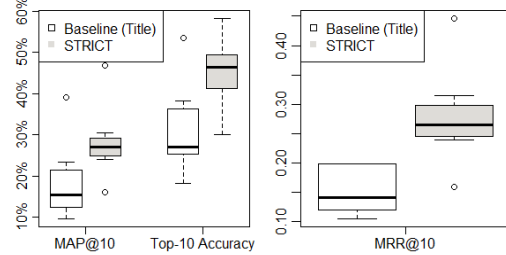

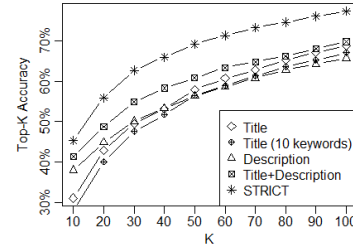Fig. 3. Comparison of retrieval performance with baseline query (i.e., title)

Fig. 4. Comparison of Top-K accuracy with all baseline queries

$p - value : 0.031 < 0.05$)–for various K-values which demonstrate the actual strength of the proposed technique.

We also investigate the retrieval performance of baseline queries containing source code tokens only and without any code tokens. From Table VI, we see that combination of both token types produced better quality search queries (41% Top-10 accuracy) than any single type alone (i.e., 30%–34% Top-10 accuracy). However, our technique provides even better queries in terms of all retrieval performance metrics.

Thus, to answer **RQ$_2$**, our suggested queries provide 45% Top-10 accuracy and 77% Top-100 accuracy with moderate mean average precision and mean reciprocal rank which are significantly higher than those of baseline queries (e.g., title of a change request) and are also promising.

Answering RQ$_3$–Role of IR Algorithms: We investigate how the two term-weighting techniques–TextRank and POSRank– perform in identifying good quality search terms from a software change request. Table VII summarizes our investigation details. We see that both TextRank and POSRank perform almost equally well in terms of all four performance metrics. Our technique–STRICT– provides about 45% Top-10 accuracy with each term-weight considered which shows their potential. However, their combination in STRICT provides marginal improvement in precision (i.e., MAP@10) and reciprocal rank (i.e., MRR@10), which possibly justifies our choice for the combination. Similar scenario is observed in the case of query effectiveness. Combination of TextRank and POSRank

TABLE VII
ROLE OF TEXTRANK AND POSRANK IN STRICT QUERIES

Term weight	Retrieval Performance		Effectiveness	
	Metric	Value	Metric	Value
TextRank	Top-10 Accuracy	46.02%	Improved	54.85%
	MRR@10	0.26	Worsened	37.05%
	MAP@10	26.75%	Preserved	8.10%
POSRank	Top-10 Accuracy	45.31%	Improved	54.52%
	MRR@10	0.27	Worsened	37.13%
	MAP@10	27.63%	Preserved	8.35%
TextRank + POSRank	Top-10 Accuracy	**45.34%**	Improved	**57.84%**
	MRR@10	**0.28**	Worsened	34.94%
	MAP@10	**28.09%**	Preserved	**7.22%**

reduces worsening ratio and increases the improvement ratio to 58% whereas the individual ratio is 55% for each of them.

Thus, to answer **RQ₃**, both TextRank and POSRank are found effective in identifying good quality search terms from a change request. Each of them alone provides such effective search queries that can return better ranks than 55% of the baseline queries on average, which is promising.

E. Comparison with Existing Techniques

STRICT can be considered both as (1) a search term identification technique and (2) a search query reformulation technique. It not only identifies good quality search terms from a change request but also, in essence, reformulates the baseline queries by discarding low quality search terms. We thus compare our technique with two relevant existing studies–Kevic and Fritz [24] and Haiduc et al. [18]–that identify search terms and reformulate search queries from software change requests respectively. To the best of our knowledge, Kevic and Fritz [24] is the only available study in the literature for search term identification, and Rocchio's expansion [39] is reported to be one of the best performing strategies for query reformulation by Haiduc et al. [18]. Thus, these two can be considered as the state-of-the-art techniques. We select them for comparison using our experimental dataset (Section IV-A), and Table VIII, Fig. 5 and Fig. 6 report our comparative analyses as follows.

Answering RQ₄–Comparison with the State-of-the-Art: Kevic and Fritz [24] use a heuristic-based regression model in identifying search terms from a change request where they consider frequency (i.e., *tf-idf*), location (i.e., *inSumAndBody, isInMiddle*) and notation (i.e., *isCamelCase*) of the terms from the request content. Rocchio [39] expands a baseline query by collecting candidate terms from the top $K = 5$ source files returned by that query and then applying *tf-idf* based term weighting to each of them [18]. We implement both methods using their provided settings (e.g., metric weight), collect their search queries for the change requests, and then evaluate those queries using the same search engine they used (i.e., Lucene).

From Table VIII, we see that our queries are more effective than the queries provided by Kevic and Fritz [24] or Rocchio's method [39]. In the case of retrieval performance, the close competitor is Rocchio's method whereas Kevic and Fritz performs the better (of the two) in the case of query effectiveness. They provide 29% Top-10 accuracy and 40% rank improvement (than baseline ranks) at best.

TABLE VIII
COMPARISON WITH EXISTING TECHNIQUES

Technique	Retrieval Performance		Effectiveness	
	Metric	Value	Metric	Value
Kevic and Fritz [24]	Top-10 Accuracy	24.90%	Improved	**40.09%**
			Worsened	53.95%
	MRR@10	0.12	Preserved	5.96%
	MAP@10	12.45%	MRD	+101
Rocchio's Method [18, 39]	Top-10 Accuracy	**29.48%**	Improved	37.59%
			Worsened	56.38%
	MRR@10	0.16	Preserved	6.03%
	MAP@10	**17.66%**	MRD	+45
STRICT (Proposed approach)	Top-10 Accuracy	45.34%*	Improved	57.84%*
			Worsened	34.94%*
	MRR@10	**0.28***	Preserved	**7.22%**
	MAP@10	**28.09%***	MRD	**-147**

* = Statistically significant difference between proposed and existing techniques
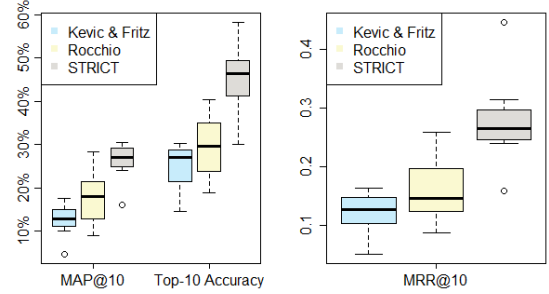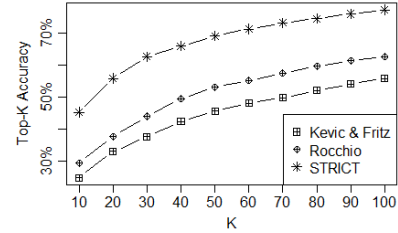

Fig. 5. Comparison with existing techniques

Fig. 6. Comparison of Top-K accuracy with existing techniques

On the contrary, our technique–STRICT–provides 45% Top-10 accuracy with 28% mean average precision@10 and a mean reciprocal rank@10 of 0.28 which are promising. It also improves 58% of 1,939 baseline queries which is highly promising according to relevant literature [18, 32]. We also found that all of our reported measures–*accuracy, precision, reciprocal rank, improvement* and *worsening*–are significantly higher (i.e., all $p - values < 0.05$) than that of these two competing techniques. When the retrieval performance measures are plotted using box plots in Fig. 5, we also see that our performance is quite higher than the state-of-the-art with higher medians and lower variances. More interestingly, when we plot Top-K accuracy against various K-values in Fig. 6, our accuracy always remained significantly higher than Kevic and Fritz [24] (i.e., $p - value :< 0.001$) and Rocchio [39] (i.e., $p - value : 0.001 < 0.05$). The state-of-the-art provides 63% Top-100 accuracy whereas that measure for STRICT is 77%, which clearly demonstrates the superiority of our technique.

Thus, to answer **RQ₄**, our technique–STRICT– performs significantly better than the state-of-the-art in identifying good quality search terms from a given change request. Such terms not only provide better retrieval performance (e.g., 77% Top-100 accuracy) but also reformulate a baseline query more effectively (e.g., 58% improvement) than the state-of-the-art.

V. Threats to Validity

Threats to internal validity relate to experimental errors and biases [47]. Re-implementation of the existing techniques is a possible source of such threat. Due to the lack of reliable or directly applicable prototypes, both existing techniques–Kevic and Fritz [24] and Rocchio [39]–were re-implemented. However, these techniques are based on two different equations with clearly stated independent and dependent variables, and we implemented them carefully. Besides, we ran them in our experiments multiple times, and compared with their best performance. Thus, such threat might be mitigated.

Threats to external validity relate to the generalizability of a technique [37]. So far, we experimented with only Java-based systems. However, given the simplicity in our project corpus creation (Section IV-B), our technique can be easily replicated for subject systems using other programming languages.

The POS tagging might contain a few false positives given that preprocessed sentences are used instead of original sentences. However, its impact might be low since stemming is not performed that affects the individual words, and highly connected terms are chosen as search terms.

VI. Related Work

Search Query Suggestion & Reformulation: A number of studies in the literature attempt to support software developers in *concept/feature/concern location* tasks using search query suggestion. They apply different lightweight heuristics [24], structural analyses [4, 31] and query reformulation strategies [13, 18, 19, 42]. They also perform different query quality analyses [14, 15, 16, 17] and data mining activities [22, 25, 46]. However, most of these approaches (1) expect a developer to provide the initial search query which they can improve upon, and (2) their main focus is improving a given query from the change request. Unfortunately, as existing studies [12, 24] suggest, preparing an initial search query is equally challenging, and those approaches do not provide much support in this regard. In this study, we propose a novel technique–STRICT–that suggests a list of suitable terms as an *initial search query* from a change request. Kevic and Fritz [24] consider a list of heuristics such as *frequency, location, parts of speech* and *notation* of the terms in the task description, and employ a prediction model for identifying search terms from a change request. Our work is closely related to theirs, and we compare with it directly in our experiments. In essence, our work is also aligned with query reformulation domain since it reformulates a baseline query by discarding the low quality search terms from the query. According to Haiduc et al. [18], Rocchio's method [39] is found to be one of the best query reformulation strategies, and thus is also closely related to ours. We also directly compare with this existing technique, and the detailed comparison can be found in Sections IV-E. Haiduc et al. and colleagues conduct several other studies on how to reformulate a given search query that apply machine learning [18] and query quality analyses [15, 17]. Given that a well-formed initial query requires less reformulation and preparing the initial query is already challenging [24], our technique can complement their techniques. Howard et al. [22] suggest semantically similar query for a given query by mining comment-code mapping from a source code repository. It could also possibly perform better if the initial query is prepared carefully which our technique does, rather than the query is chosen randomly. Bassett and Kraft [4] apply structural term weighting to feature location by emphasizing on source code tokens during query formulation. However, as our finding suggests (RQ_1, Section IV-D), code tokens might not be always available, and thus queries based on them could be limited in performance. Other related studies apply ontology [45], query-based configurations [4, 31, 32] and phrasal concepts [21, 42] in concept location.

Search Mechanisms: There also exist a number of studies on search mechanisms [2, 27, 29, 34, 38, 41] in the literature that apply the search queries to actually locate the concepts, features, concerns or bugs within the source code. These studies adopt static analyses, dynamic analyses or perform both analyses on the source code to identify the items of interest (e.g., methods to be changed). Revelle et al. [38] combine information from three different processes–textual analysis, dynamic analysis and web mining–and apply *PageRank* algorithm [7] like ours. However, they apply that algorithm in a different context–ranking methods within the project source–rather than search term identification from a change request. Other studies are not strongly related to our work–search term identification, and we did not compare with them directly in our experiments.

Thus, from a technical perspective, we adapt two information retrieval techniques–TextRank and POSRank–in the context of concept location, and identify a list of suitable search terms from a change request. We exploit not only the co-occurrences but also the grammatical dependencies among the words from the request text carefully using a graph-based term weighting approach, and then capture the relative importance of those terms. Such idea was considered by no relevant existing studies, and the experimental findings also confirm the high potential of our idea.

VII. Conclusion & Future Work

Studies suggest that preparing a search query from a change request is challenging for the developers. In this paper, we propose a novel technique–STRICT–that automatically suggests suitable search terms from a change request. It employs two term-weighting techniques–TextRank and POSRank– from information-retrieval to capture a term's importance. Experiments with 1,939 change requests from 8 subject systems show that our technique can identify better quality search terms than baseline for 52%–62% of the requests, and retrieves relevant results with 30%–57% Top-10 accuracy and about 30% mean average precision, which are promising. Comparison with two state-of-the-art techniques validates our empirical findings and also confirms the superiority of our technique. In future, we plan to further extend our evaluation and implement the technique as an Eclipse plug-in tool. Experimental data along with supporting materials are available elsewhere [1].

REFERENCES

[1] STRICT: Experimental Data. URL http://homepage. usask.ca/~masud.rahman/strict.

[2] G. Antoniol, G. Canfora, G. Casazza, A. De Lucia, and E. Merlo. Recovering Traceability Links between Code and Documentation. *TSE*, 28(10):970–983, 2002.

[3] 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.

[4] B. Bassett and N. A. Kraft. Structural information based term weighting in text retrieval for feature location. In *Proc. ICPC*, pages 133–141, 2013.

[5] G. Bavota, A. De Lucia, R. Oliveto, A. Panichella, F. Ricci, and G. Tortora. The Role of Artefact Corpus in LSI-based Traceability Recovery. In *Proc. TEFSE*, pages 83–89, 2013.

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

[7] 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.

[8] G. Capobianco, A. D. Lucia, R. Oliveto, A. Panichella, and S. Panichella. Improving IR-based Traceability Recovery via Noun-Based Indexing of Software Artifacts. *Journal of Software: Evolution and Process*, 25(7):743–762, 2013.

[9] J. Cordeiro, B. Antunes, and P. Gomes. Context-based Recommendation to Support Problem Solving in Software Development. In *Proc. RSSE*, pages 85 –89, 2012.

[10] B. Dagenais and M. P. Robillard. Creating and Evolving Developer Documentation: Understanding the Decisions of Open Source Contributors. In *Proc. FSE*, pages 127–136, 2010.

[11] B. Dit, L. Guerrouj, D. Poshyvanyk, and G. Antoniol. Can better identifier splitting techniques help feature location? In *Proc. ICPC*, pages 11–20, 2011.

[12] 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.

[13] 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.

[14] S. Haiduc. Automatically Detecting the Quality of the Query and its Implications in IR-based Concept Location. In *Proc. ASE*, pages 637–640, 2011.

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

[16] 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.

[17] 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.

[18] 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.

[19] 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.

[20] S. Hassan, R. Mihalcea, and C. Banea. Random-Walk Term Weighting for Improved Text Classification. In *Proc. ICSC*, pages 242–249, 2007.

[21] E. Hill, L. Pollock, and K. Vijay-Shanker. Improving Source Code Search with Natural Language Phrasal Representations of Method Signatures. In *Proc. ASE*, pages 524–527, 2011.

[22] 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.

[23] Otto Jespersen. The Philosophy of Grammar. 1929.

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

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

[26] Andrew J. Ko, Brad A. Myers, Michael J. Coblenz, and Htet Htet Aung. An Exploratory Study of How Developers Seek, Relate, and Collect Relevant Information During Software Maintenance Tasks. *TSE*, 32(12):971–987, 2006.

[27] 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.

[28] A. Marcus and S. Haiduc. Text Retrieval Approaches for Concept Location in Source Code. In *Software Engineering*, volume 7171, pages 126–158. 2013.

[29] 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.

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

[31] 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.

[32] 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.

[33] C. Parnin and C. Treude. Measuring API Documentation on the Web. In *Proc. Web2SE*, pages 25–30, 2011.

[34] D. Poshyvanyk and A. Marcus. Combining Formal Concept Analysis with Information Retrieval for Concept Location in Source Code. In *Proc. ICPC*, pages 37–48, 2007.

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

[36] M. M. Rahman, S. Yeasmin, and C. K. Roy. Towards a Context-Aware IDE-Based Meta Search Engine for Recommendation about Programming Errors and Exceptions. In *Proc. CSMR-WCRE*, pages 194–203, 2014.

[37] M. M. Rahman, C. K. Roy, and D. Lo. RACK: Automatic API Recommendation using Crowdsourced Knowledge. In *Proc. SANER*, pages 349–359, 2016.

[38] M. Revelle, B. Dit, and D. Poshyvanyk. Using data fusion and web mining to support feature location in software. In *Proc. ICPC*, pages 14–23, 2010.

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

[40] 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.

[41] G. Scanniello and A. Marcus. Clustering Support for Static Concept Location in Source Code. In *Proc. ICPC*, pages 1–10, 2011.

[42] 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.

[43] P. Thongtanunam, R. G. Kula, N. Yoshida, H. Iida, and K. Matsumoto. Who Should Review my Code ? In *Proc. SANER*, pages 141–150, 2015.

[44] K. Toutanova, D. Klein, C.D. Manning, and Y. Singer. Feature-Rich Part-of-Speech Tagging with a Cyclic Dependency Network. In *Proc. HLT-NAACL*, pages 252–259, 2003.

[45] L.A. Wilson. Using Ontology Fragments in Concept Location. In *Proc. ICSM*, pages 1–2, 2010.

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

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

[48] J. Zhou, H. Zhang, and D. Lo. Where should the bugs be fixed? - more accurate information retrieval-based bug localization based on bug reports. In *Proc. ICSE*, 2012.