Assisting Code Search with Automatic Query Reformulation for Bug Localization

Bunyamin Sisman, Avinash C. Kak
Purdue University
West Lafayette, IN
{bsisman, kak}@purdue.edu

Abstract—**Source code retrieval plays an important role in many software engineering tasks. However, designing a query that can accurately retrieve the relevant software artifacts can be challenging for developers as it requires a certain level of knowledge and experience regarding the code base. This paper demonstrates how the difficulty of designing a proper query can be alleviated through automatic Query Reformulation (QR) — *an under-the-hood operation for reformulating a user's query with no additional input from the user*. The proposed QR framework works by enriching a user's search query with certain specific additional terms drawn from the highest-ranked artifacts retrieved in response to the initial query. The important point here is that these additional terms injected into a query are those that are deemed to be "close" to the original query terms in the source code on the basis of *positional proximity*. This similarity metric is based on the notion that terms that deal with the same concepts in source code are usually proximal to one another in the same files. We demonstrate the superiority of our QR framework in relation to the QR frameworks well-known in the natural language document retrieval by showing significant improvements in bug localization performance for two large software projects using more than 4,000 queries.**

Index Terms—**Query Expansion, Query Reformulation, Pseudo Relevance Feedback, Bug Localization; Software Maintenance**

I. Introduction

During software development and maintenance, developers mainly depend on a source code search engine to locate the different parts of a software library [1]. The tasks for which code search is carried out include concept localization, bug localization, traceability link recovery, code comprehension, etc. Over the years, of particular focus has been code search and retrieval for the purpose of automatic bug localization. That is not surprising since locating the buggy parts of a software project is a critical element of software development and maintenance.

Our particular interest is in code search as employed in Information Retrieval (IR) techniques for locating the parts of a software library that are relevant to a bug report. With IR based approaches, we treat a bug report as a textual query that is compared with the files in a software library — the comparison technique depending on which IR technique is used — and the files ranked in the order of their relevance to the query [2], [3], [4], [5]. It was recently demonstrated by us [2] that the retrieval precision for bug localization in an IR-based framework can be improved dramatically if we include temporally weighted version histories of the files in the retrieval process. Another recent contribution that also reports improved bug localization with an IR-based approach is by Zhou et al. [3]; their BugLocator tool leverages the similar bugs resolved in the past for enhanced bug localization accuracy.

Obviously, the success of these studies depended much on the quality of the queries — in other words, the quality of the bug reports when IR-based search was used for automatic bug localization. In general, in code search, a poorly designed query is likely to be indiscriminate in assigning high ranks to the relevant and the irrelevant documents, which would make it difficult to locate the desired software components reliably. And, in general, constructing a good query for retrieving the relevant files and/or artifacts with high reliability is no easy task in the domain of software.

The problem of constructing a good query for code search is exacerbated by the fact that, despite the naming conventions in all programming languages, arbitrary abbreviations and concatenations are frequently used in source code for the naming of concepts, objects, artifacts, and so on [6]. Therefore, searching a code base for concepts, objects, artifacts, etc., using terms that are oblivious to the abbreviation and concatenations actually used can, in the worst case, miss out entirely on the files highly relevant to a given search, and, in the best, result in poor values for the relevancies. While an experienced developer — especially one who is already familiar with the code base — may be able to anticipate the peculiarities of the naming conventions used in a software library, it is easy to imagine how much harder it would be for an inexperienced developer to do the same [7].

Obviously, code search needs effective techniques for what is known as *Query Reformulation* (QR). As it turns out, researchers in the software engineering community have looked at QR in the past — *but in a way that either places additional burden on the developer in constructing a query or that are based on general statistical properties of a software library.* To elaborate, as an example of QR that places additional burden on the developer, we have the study of Explicit Relevance Feedback (ERF) by Gay et al. [8]. They used Rocchio's method for QR to locate a target file by reformulating the original query [9] through an iterative interaction with the user. And, as an example of QR that automatically reformulates a query on the basis of general statistical properties of a library (without regard to the retrieval effectiveness of the terms in the original query), we have the work by Marcus et al. [10]. They used Latent Semantic Indexing (LSI) to determine what terms were more likely to co-occur with what other terms in software artifacts. The term co-occurrence information obtained in this manner was later used to automatically expand a given query that has a single term initially. Along the same lines, Yang and Tan proposed an approach to infer semantically related terms automatically via a pairwise comparison of the code segments and the comments in the source code [11].

In this paper, we present a QR Framework for *automatic* query reformulation in the software context in a manner that requires no further input from a user and that, at the same time, is attuned to the original query. We believe this framework is ideally suited for automatic QR for bug localization in order to improve the quality of what comes back from an IR-based search engine for the following reasons:

- Bug reports by their very nature contain terms that can be expected to result in the retrieval of a set of documents with a reasonable chance that the set will contain at least some documents directly relevant to the bug. The issue then becomes how to analyze this set of initially retrieved documents for modifications to the original query so that the set of documents retrieved with the reformulated query will give us improved retrievals. Obviously, while a bug report will contain terms relevant to the intended search task, in general it may also contain terms that are extraneous to what a user is looking for.
- Within the naming conventions that may be recommended for a particular programming language, the programmers are generally free to use any abbreviations and concatenations at all that, at least in their minds, convey some information regarding the purpose served by a name. This obviously presents hurdles in code search for those not already familiar with the code base. And even by those who are familiar with the code base, the relative importance of the different names with regard to their ability to serve as discriminating identifiers for a software artifact may not be well understood.

Our goal is to analyze the documents retrieved for an initial query for additional terms that are conceptually related to the original query terms on the basis of term-term and term-document relationships in the highest-ranked artifacts. The proposed QR framework exploits the fact that the terms that consistently appear in close *proximity* to the query terms in the retrieved software artifacts are likely to be related to the query. For the evaluation of the proposed proximity based model, we compared it to the well-known QR methods in the literature with the following questions in mind:

- **Question1:** Does the term proximity based QR technique we propose improve the accuracy of source code retrieval, if so, to what extent?
- **Question2:** How do the QR techniques that are currently in the literature perform for source code retrieval?
- **Question3:** How does the initial retrieval performance affect the performance of QR?
- **Question4:** What are the conditions under which QR may perform poorly?

Our experimental validation presents answers to all these questions. We also show that our term proximity based approach to QR for bug localization extracts terms more accurately and outperforms the other QR methods significantly.

This paper is organized as follows. In Section II, a brief background on relevance feedback is presented. Section III presents an overview of the more prominent of the QR methods in the literature today. In Section IV, we present our term proximity based approach to automatic query reformulation. Section V describes the retrieval and the evaluation framework for QR. We evaluate the performance of QR in Section VI. Section VII describes the threats to the validity of the approach and Section VIII concludes the paper.

II. Background on Relevance Feedback

Locating a piece of information in a large repository of documents is a challenging task as it requires formulating a query that can successfully distinguish the relevant documents from the irrelevant ones. If the original query is found to yield unsatisfactory results, it could be refined by either engaging in an explicit query-response session with the search engine, or, more automatically, by analyzing the documents retrieved for the initial query in order to find ways to augment it for further retrieval. Relevance Feedback frameworks are commonly used to assist users in this process.

Although the main ideas underlying relevance feedback are straightforward, there is no single feedback strategy that works for different types of queries and for different domains. As a result, relevance feedback has continued to be an active area of research, with its focus being on (1) how to best acquire the feedback; and (2) how to best utilize it to improve the retrieval accuracy [9], [12], [13], [14]. In the next two subsections, we provide a brief survey of this research as it has been used in the past for QR in software context. Our comments to follow include a comparison of this framework with ours.

A. Explicit Relevance Feedback

In explicit relevance feedback (ERF), after seeing the first set of retrieval results returned by the search engine, the user provides his/her judgments *explicitly* by marking the relevant and irrelevant set of results. In [8], Gay et al. showed that ERF can assist developers in locating the target file(s) by reformulating the original query iteratively. At each iteration, the reformulated query is submitted and a new set of retrieval results is presented to the user to obtain the next round of feedback. This process is repeated until all the target files are located.

The most commonly used method for query reformulation is perhaps Rocchio's method. With Rocchio's method, both the query and the documents are regarded as term vectors with the size of the vocabulary V. Given a query Q, and the set of top X retrieved files D with respect to Q, the user populates the set of relevant files $D_{REL} \subset D$ and the irrelevant files $D_{IR} \subset D$. Then the query is reformulated as follows:

$$Q^{ref} = \alpha \cdot Q + \beta \cdot \sum_{f \in D_{REL}} \frac{f}{|D_{REL}|} - \gamma \cdot \sum_{f \in D_{IR}} \frac{f}{|D_{IR}|}. \quad (1)$$

Although ERF is the most accurate type of feedback, its use in source code retrieval is highly limited as it prolongs the query session and developers can be expected to find it burdensome to have to repeatedly mark the relevant and irrelevant documents in an iterative session. Evaluating the performance gain obtained with ERF is another drawback since the user spends a considerable amount of time judging the relevancy of the retrievals and some of the relevant files are already located during this process.

B. *Pseudo Relevance Feedback*

In comparison to ERF, Pseudo (Blind) Relevance Feedback (PRF) employs a different approach in the sense that the user input is not required. With this method, after getting the first set of retrieval results, the set D of the top X retrieved documents is regarded as an approximation to the set of relevant documents and the "best" terms from these documents are chosen to reformulate the query. As this type of RF does not require user involvement, it can be performed instantly even without user noticing it.

The main intuition behind PRF is that since the files in D receive the highest retrieval scores, they are highly likely to contain further informative terms that are conceptually related to the original query. Obviously, PRF can only work if the original query is reasonably strong in its power to retrieve at least some of the relevant documents. As it turns out, and as we argue later in this paper, this assumption is well satisfied in the domain of bug localization on account of how the bug reports are generally constructed.

III. Past Work on Automatic Query Reformulation

In this Section, we present two currently well-known methods for automatic QR. We include them here since we will use them in a comparative evaluation of automatic QR for bug localization in Section VI.

A. *Rocchio's Formula for Automatic QR*

Rocchio's Formula (ROCC) for automatic QR is well known in natural-language document retrieval [9]. It is therefore an automatic candidate to try in software context also. ROCC is based on the following rationale: Use the set D of the top files retrieved for the initial query Q to create a vector space model (VSM) of the term-term and term-document relationships in the set of documents in D. For each word w in D, ROCC calculates a weight that expresses its importance in a reformulated version Q^{ref} of the query Q according to:

$$tf(w, Q^{ref}) = (1-\beta) \cdot tf(w, Q) + \beta \cdot \sum_{f \in D} \frac{tf(w, f)}{|D|} \quad (2)$$

where $f \in D$ is a file in the set D, $tf(w, Q)$ the term frequency of w in the initial query Q, and $tf(w, f)$ is the same in the file f. As illustrated by the formula, the frequencies of the terms in D are used to express the importance of those terms for a reformulation of Q. The greater the frequency of a term in D, the higher its weight in the reformulated query (Q^{ref}). The interpolation parameter $\beta \in [0, 1]$ adjusts the weights of the expansion terms with respect to the original query terms. Although ROCC is a robust performer, competitive probabilistic models with richer semantics have been proposed over the years.

B. *Automatic QR Using the Relevance Model*

Another approach to automatic QR, again in the natural language context, is the *Relevance Model* (RM) [12], [15]. What makes RM based QR different from Rocchio's formula is that as a formal probabilistic model, it incorporates the ranking scores of the documents in D as an additional component for a more accurate selection of the terms to be used for reformulating the original query. The goal is to estimate a co-occurence probability distribution $p'(w|Q)$ for the terms in the vocabulary vis-a-vis the query terms. Given the first pass retrieval results D, RM estimates this probability by:

$$p'(w|Q) = \frac{p(w, Q)}{p(Q)} \quad \propto \quad \sum_{f \in D} p(w|f) p(f) \prod_{q \in Q} p(q|f) \quad (3)$$

where the right-hand-side formula is based on the reasonable assumption that, given a document $f \in D$, we can consider w and Q to be independent. The query likelihood component in the above formula, $p(Q|f) = \prod_{q \in Q} p(q|f)$, gives the score of a file f with respect to the original query and different retrieval models can be used to compute it. Subsequently, the query Q is reformulated using the following interpolation formula:

$$p(w|Q^{ref}) = (1-\beta) \cdot p(w|Q) + \beta \cdot p'(w|Q) \quad (4)$$

where β is an experimental parameter that determines the extent one should mix the original query as presented by the user and the new terms as made evident by their associated probabilities.

IV. The Proposed Approach to Query Reformulation For Source Code Retrieval

The QR methods we described in Section III consider all the terms in a file $f \in D$ equally for reformulation. However, given the distinct structure and the term relationships in source code, one would think the importance of a term in a file to an existing term in a query ought to depend on the likelihood that the former normally shows up *proximally* to the latter in the same file. The proximity we are referring to may relate to the different terms appearing in the same long name through an underscore or a camel-case based concatenation. Proximity also refers to the different terms appearing close to one another in the same method or class definition and so on.

A number of different approaches have been proposed to capture the term-term proximity effect in the context of natural-language [13], [16], [17]. Inspired by these studies, we propose a simple probabilistic model, which we call

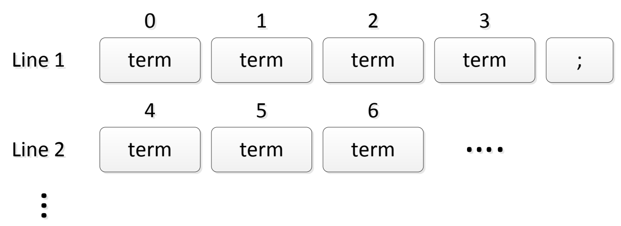


Fig. 1: *An illustration of indexing the positions of the terms in a source file.*

Spatial Code Proximity (SCP) model for measuring term-term positional proximity that is particularly appropriate for source code. SCP estimates a proximity probability distribution $p'(w|Q)$ over each term w in the vocabulary V of the system on the basis of their *proximity frequencies* in D with respect to Q.

In order to use proximity for query reformulation, we index the position of each term in every file $f \in D$ as demonstrated in Fig. 1. Vis-a-vis a given query Q, we next associate a term proximity frequency with each term in the file f in the following manner: (1) We initialize the value of the proximity frequency to 0 for each term in f. (2) We then identify every position in f where the term is the same as one of the terms in Q. (3) Subsequently, we place a window of size $2W + 1$ at each position identified in the previous step. The window is placed symmetrically around the identified positions. (4) The proximity frequency associated with every term in the window is now incremented by 1. The proximity frequencies (pf) obtained in this manner can be expressed as:

$$pf(w, f|Q) = \sum_{q \in Q} \sum_{i \in P_f[w]} \sum_{j \in P_f[q]} I_W(i,j) \tag{5}$$

where $P_f[\cdot]$ is the position operator which returns the set of position indices of a given term in f. The window operator $I_W(i,j)$ returns 1 if the distance between the positions i and j is less than or equal to W:

$$I_W(i,j) = \begin{cases} 1 & \text{if } |i-j| \le W \\ 0 & \text{otherwise.} \end{cases} \tag{6}$$

Based on these term frequencies, we may estimate the proximity probability distribution $p'(w|Q)$ using the top retrieved documents D:

$$p'(w|Q) = \frac{\sum_{f \in D} pf(w, f|Q)}{\sum_{w' \in V} \sum_{f \in D} pf(w', f|Q)} \tag{7}$$

Using Eq. 4, we reformulate the query by interpolating the original query with the proximity probability distribution. With this formulation, the terms that tend to co-occur with the query terms consistently in close proximity across the files in D will receive a high probability mass whereas the terms that appear far away from the query terms will be discarded in QR. Fig. 2 presents the Query Reformulation process.

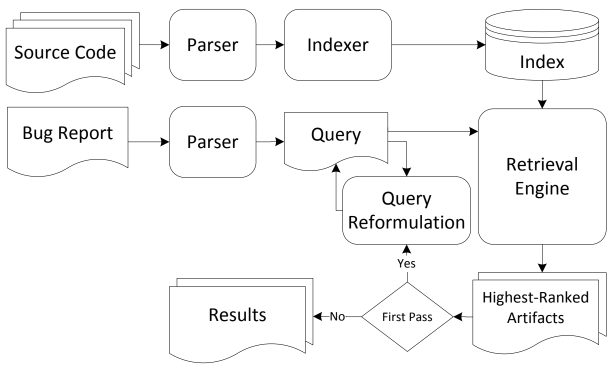


Fig. 2: *An illustration of data flow in QR process.*

TABLE I: *QR achieved with the proposed SCP method vis-a-vis the same achieved with the ROCC and RM methods for the bug #20750 filed for the Chrome project.*

SCP		RM		ROCC	
Original Query Terms					
tab	0.1965	tab	0.1201	tab	1.0000
animation	0.0650	load	0.0444	animation	0.2867
load	0.0444	animation	0.0359	load	0.2000
drag	0.0340	drag	0.0302	drag	0.1000
cause	0.0222	cause	0.0222	cause	0.1000
separate	0.0222	separate	0.0222	separate	0.1000
Expansion Terms					
strip	0.0231	bookmark	0.0157	bookmark	0.1721
content	0.0161	strip	0.0145	gtk	0.1631
pin	0.0131	content	0.0132	browser	0.1524
gtk	0.0127	gtk	0.0121	content	0.1367
model	0.0111	model	0.0117	strip	0.1358
control	0.0094	drop	0.0116	model	0.1154
bound	0.0084	browser	0.0111	bar	0.1100
tabstrip	0.0083	window	0.0099	window	0.1040
layout	0.0072	control	0.0099	node	0.0820
Average Precision (Baseline: 0.1667)					
1.0000		0.1429		0.2000	

A. A Motivating Example

The bug #20750 filed for Google Chrome[1] has a title that reads: "drag a loading tab will cause the loading animation separate with the tab". The target file that has been modified to fix this bug is "chrome/browser/gtk/tabs/tab_strip_gtk.cc". The reformulated versions of this query along with the normalized term frequencies as estimated by the three QR methods are given in Table I.

Looking at the relevant file, we see that the most important terms of the query to retrieve this file are tab, animation and load; while the terms cause and separate are not related to the information need primarily. As can be seen in Table I, all three methods are able to capture this notion of relevancy by reweighing the original query terms accordingly. Additionally, the QR methods are also able to extract relevant terms from the initial retrieval results to expand the query. For instance, the terms gtk and strip are good expansion terms that are conceptually related to the query. Interestingly, differing from the other methods, SCP extracts the terms tabstrip, bound and pin from the feedback files. These proximity terms are

[1]http://code.google.com/p/chromium/issues/detail?id=20750

the key terms that help the SCP method achieve a perfect average precision.

To better understand the affect of term proximity on QR, note that the rank of the term strip as computed by the SCP method is relatively high in comparison to the other two methods because this term frequently appears right next to the term tab in the feedback files hence receiving a higher probability mass. The term tabstrip is another interesting proximity term extracted by only the SCP method. As a compound term constructed without using punctuation characters or camel casing, it may be difficult for the developer to think of this term for retrieval purposes.

V. Retrieval & Evaluation Framework

A. Retrieval Model

Obviously, the underlying retrieval function has a prominent effect on QR since it is first used to rank the files in the collection to obtain the set D of feedback files. Then, after the reformulation, a second pass of retrieval with respect to the updated query is performed to obtain the final retrieval set. Over the years, several retrieval frameworks have been evaluated for bug localization [2], [3], [5]. In these studies, it has been shown that the models that use Inverse Document Frequency (IDF) produce strong empirical results. Motivated by the prior studies, we use the TF-IDF framework [18] as our baseline retrieval model where the score of a query with respect to a file is computed by the following weight function:

$$p(Q|f) \propto score_{\text{TFIDF}}(Q, f) = \sum_{q \in Q} tf(q, f) \cdot idf(q). \tag{8}$$

We compute the frequency of a term in a file by Robertsons's TF:

$$k \cdot \frac{tf}{(tf + k \cdot (1 - b + b \cdot |f|/avg\_l))} \tag{9}$$

and the inverse document frequency of the terms by Spark Jones' IDF: $log_2(|C|/(N_q + 1))$, where $|f|$ is the length of a file f, avg_l is the average document length in the collection, $|C|$ is the total number of files and N_q is the number of files that the term q appears in. The model constants k and b provide non linearity to the term frequencies and we empirically set $k = 1.2$ and $b = 0.75$ for the QR experiments.

B. Query Performance Prediction (QPP) Metrics

To evaluate the effect of QR on the quality of the retrievals, we employ the following QPP metrics [19], [20]:

1) Average Inverse Document Frequency (avgIDF): Inverse Document Frequency (IDF) of a term determines the specificity of the term with respect to a collection. A query consisting of a set of terms with high IDF values is more likely to locate the relevant files as the size of the retrieved set becomes smaller for such queries. The average for this metric is given by

$$avgIDF(Q) = \sum_{q \in Q} log_2[|C|/(N_q + 1)]/|Q|. \tag{10}$$

2) Average Inverse Collection Term Frequency (avgICTF): Inverse Collection Term Frequency (ICTF) measures the specificity of a term on the basis of the overall term frequency in the entire software library. Its average is given by

$$avgICTF(Q) = \sum_{q \in Q} log_2[NT/(tf(q, C) + 1)]/|Q| \tag{11}$$

where NT is the number of terms in the collection and $tf(q, C)$ is the frequency of a term q in the collection.

3) Query Scope (QS): In [20], Query Scope (QS) is defined as the percentage of the documents that contain at least one query term. A small value of QS for a query with respect to a given collection indicates that the query is discriminatory. Note that as the length of the queries increase this metric becomes less effective in measuring the query specificity.

4) Simplified Clarity Score (SCS): Simplified Clarity Score (SCS) is given by the Kullback-Leibler (KL) divergence between the query and the collection:

$$SCS(Q) = \sum_{q \in Q} p(q|Q) log_2[p(q|Q)/p(q|C)]. \tag{12}$$

SCS measures the discriminatory power of the query while also taking into account the query length.

VI. Experimental Evaluation

Our goal in this section is to compare the power of query reformulation that is based on the spatial code proximity analysis presented in this paper with two other approaches drawn from natural language research. Our comparative results are based on two large software projects, namely Eclipse[2] and Google Chrome[3]. In order to evaluate the presented algorithms, we need a set of bug reports, B and the source files modified to fix the corresponding bugs as the ground truth. Unfortunately, the bug tracking databases such as Bugzilla[4] do not store the actual modifications committed for the reported bugs. The common approach to link the modifications to the bug reports is to look for pointers in the commit messages to the bug tracking database [3], [21], [22]. For Eclipse, we follow a similar approach and employ regular expressions for an accurate reconstruction of the links between the bug reports and the corresponding modifications as follows:

1) Scanning the repository logs, group the files that are modified by the same author with the same commit message with a time fuzziness of 200 seconds [23]. This step is necessary as CVS repositories store the changes made to each file separately.
2) For the target version of the software, use regular expressions to extract the bug IDs from the commit messages. Our regular expressions match the following generic phrases in the commit messages: Fix for ID, Fix ID, Fixed ID, Fixing ID, Bug ID and they are

[2]www.eclipse.org
[3]www.google.com/chrome
[4]https://bugs.eclipse.org/bugs/

TABLE II: *Evaluated Projects*

Project, Description	Language	$\|B\|$	$\#Files$	$\#Terms$	Analysis Period
ECLIPSE v3.1, Integrated Development Environment	Java	4,035	12,825	19,955	2001-04-28 - 2010-05-21
CHROME v4.0, WEB browser	C/C++	358	8,420	349,958	2008-07-25 - 2010-05-20

insensitive to the punctuation characters and the spacing within the identified phrase i.e. phrases such as BugID, Bug: ID or Bug #ID and so on are also matched.

3) Check if the extracted ID exists in the bug tracking database for the target version of the software.

For Google Chrome, the commit messages follow a more specific form. They contain a separate line to indicate whether the commit fixes any bugs and if so the bug IDs are given in that line. Note that a bug may need more than one set of modifications to be finally resolved. Therefore, we accumulate the set of modifications that are committed for the same bug over the analysis period. Finally, we consider only the bug reports that are marked as "FIXED" in the bug tracking database, which means a fix has been checked into the tree and tested for the corresponding bug. The result of this reconstruction process is a dataset that we call *BUGLinks*[5]. Table II presents the evaluated projects from this dataset and some of their statistics.

Establishing the ground truth for the retrievals in the presence of the links as constructed above is an important step in the evaluation process. For a fair evaluation, we use the set of files that are mentioned in the corresponding commits of a given bug and that are also present in the indexed version of the software as the relevant files to be retrieved. As a result, we discard the bug reports for which there are no files to be located in the corpora of the evaluated projects. For index creation, we used Eclipse version 3.1 and Chrome version 4.0.305.0.

A. Indexing Source Code

We use a fairly standard set of preprocessing steps to index the code base of each project. Firstly, the compound terms are tokenized using punctuation characters and camel case characters. The tokens thus generated are subject to a set of stemming and stopping rules. We use Porter's stemming algorithm to trim the terms to their common root forms [24]. Regarding the stopping rules, these delete the programming-language specific terms and the terms in a list of standard-English stop-words that has 733 words in it. The positions of each term in the files are recorded for the SCP method after these pre-processing steps.

The bug reports are subject to the same preprocessing steps as the source files. That is, each bug report goes through compound term splitting, and the extracted tokens are subject to stemming and stopping. Each bug report is also divided into two parts: its title and its description. The title of a bug report constitutes an explanation, albeit very brief, of the buggy behavior, with further elaboration provided by the description part. In this paper, we use the titles of the bug reports to populate the set of queries B as they are less noisy and they resemble the real world search queries better than the descriptions. In order to index the source code and perform the retrievals with QR, we extended Terrier[6], a flexible open source platform developed in Java [25].

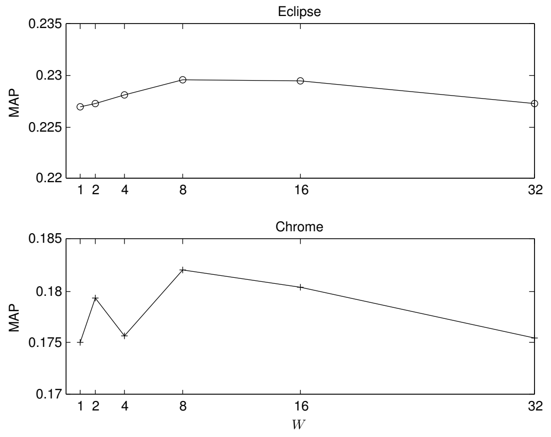


Fig. 4: *The effect of varying window size parameter W.*

B. Evaluation Metrics

Retrieval engines are commonly evaluated using precision, and recall based metrics [26]. Besides mean precision (P@) and mean recall (P@), we also present the performance of the retrievals in terms of Mean Average Precision (MAP). In order to compare the performance of the original queries and the reformulated queries, we use pairwise student's t-test on the average precisions of each query.

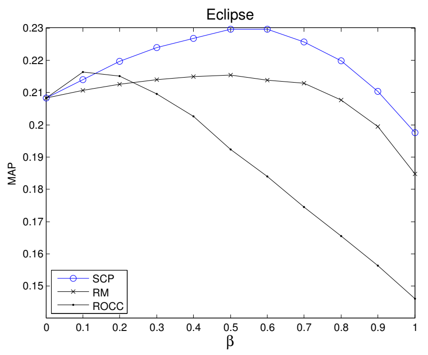
Note that any QR strategy can only be expected to give us improvements on the average, implying that we must admit to the possibility of a reformulated query giving us worse results on occasion [13]. We define the set of queries for which QR leads to improvements as *positive* queries and denote this set by B^+. Similarly, the set of queries for which QR results in a decreased performance is defined as *negative* queries and it is denoted by B^-. The remaining queries in B preserve their initial performance after QR. We consider these queries as constituting the set of *neutral* queries and we denote it by B^o. Obviously, $|B^+| + |B^-| + |B^o| = |B|$. For comparing QPP metrics on these sets, we used unpaired two-sample t-test.

C. Parameter Sensitivity Analysis

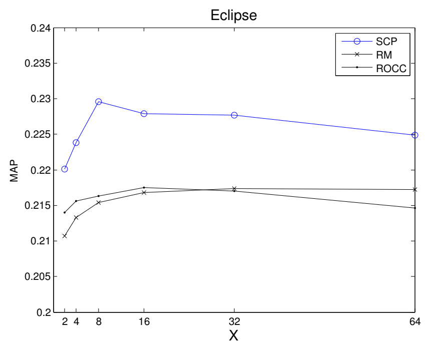
The four experimental parameters that affect the quality of retrieval from reformulated queries are: (1) The interpolation parameter β that determines the weight to be accorded to the new terms to be added to a query vis-a-vis the original terms;

[5]https://engineering.purdue.edu/RVL/Database/BUGLinks/

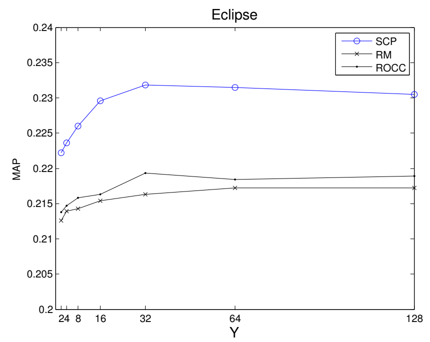
[6]http://terrier.org/



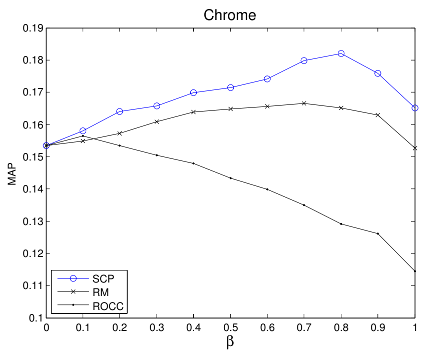
(a) The interpolation parameter.



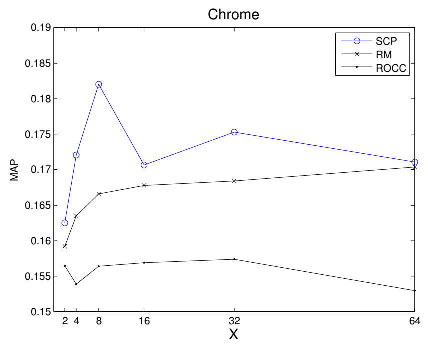
(b) The number of top files.



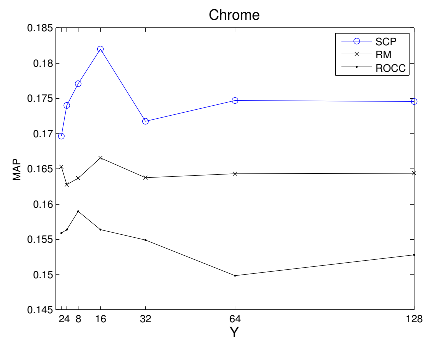
(c) The number expansion terms.



(d) The interpolation parameter.



(e) The number of top files.



(f) The number expansion terms.

Fig. 3: *The effects of varying the experimental parameters.*

(2) The number of highest ranked files to be analyzed for QR; (3) The number of terms to be added to the original query; and, finally, (4) The size parameter W of the window to be used for proximity analysis. Fig. 3 and 4 show how the retrieval performance depends on these four parameters.

With regard to β, as can be seen in Fig. 3a and 3d, while ROCC reaches its peak point for $\beta = 0.1$, RM and SCP achieve better accuracies for higher values of β. Fig. 3b, 3c, 3e and 3f present the retrieval accuracies as the number of feedback files and the feedback terms to expand the queries are varied for $X \in \{2, 4, 8, 16, 32, 64\}$ and $Y \in \{2, 4, 8, 16, 32, 64, 128\}$ for the analyzed projects. Fixing these parameters at certain chosen values empirically is a common approach to evaluating the QR methods [15]. Following this tradition, we set these parameters as $X = 8$ and $Y = 16$ unless stated otherwise. Note that SCP outperforms the other two methods consistently in these experiments. In addition to these parameters common to all three methods, Fig. 4 plots the effect of varying the window size parameter W for the SCP method. Interestingly, in both software projects, we reached the best retrieval accuracy for $W = 8$. Notice that the retrieval accuracy starts to degrade as far away terms are considered for QR.

D. Retrieval Results & Discussions

Table III presents the average performance of the three QR methods. The baseline retrieval accuracy without QR is presented in the last row of the table. The best retrieval accuracy in each column is given in bold. The column "p-value" gives the p-value of the pairwise significance testing with respect to the baseline accuracy. Table IV, on the other hand, present the retrieval accuracies on the sets B^+, B^- and B^o. For each QR method, we present the baseline performance (with the original queries in each set) and the performance of the reformulated queries along with the size of the corresponding query set. All of the differences between the baselines and the performances of the respective QR methods in this table are statistically significant at $\alpha = 0.01$.

It is obvious from the results presented in Table III that the queries for the Eclipse project perform much better than the queries for the Chrome project, indicating that there is a higher correlation between the terms used in the bug reports and the source code of Eclipse in comparison to Chrome. This could be due to the differences in the naming conventions of the respective programming languages. In the rest of this section, we will now provide answers to the four questions listed at the end of Introduction.

1) Answer to Question1: Based on the results in Table III, we conclude that QR, in general, leads to significant improvements over the baseline retrieval accuracy for bug localization. Even more significantly, the improvements obtained with the new SCP approach are noticeably higher than those obtained by the other QR methods. While all three QR methods lead to large improvements for the Eclipse project at a significance level of $\alpha = 0.01$, only the proposed SCP method

TABLE III: *Retrieval accuracy with QR for the three QR methods*

QR Method	Eclipse						Chrome					
	MAP	P@1	P@5	R@5	R@10	p-value	MAP	P@1	P@5	R@5	R@10	p-value
SCP	**0.2296**	**0.1893**	**0.1011**	**0.2849**	**0.3739**	1.03e-17	**0.1820**	**0.1788**	**0.0933**	**0.2021**	0.2775	0.0023
RM	0.2154	0.1670	0.0970	0.2729	0.3631	1.56e-04	0.1666	0.1480	0.0844	0.1966	**0.2853**	0.0140
ROCC	0.2163	0.1713	0.0961	0.2743	0.3663	4.01e-05	0.1564	0.1397	0.0793	0.1816	0.2779	0.5369
Baseline	0.2089	0.1653	0.0921	0.2632	0.3559	↑	0.1535	0.1453	0.0832	0.1838	0.2601	↑

TABLE IV: *QR for positive (B^+), negative (B^-) and neutral (B^o) queries.*

Retrieval Accuracy for B^+

QR Method	Eclipse						Chrome					
	MAP (Imp.)	P@1	P@5	R@5	R@10	$\|B^+\|$	MAP (Imp.)	P@1	P@5	R@5	R@10	$\|B^+\|$
SCP	0.2530 (+66%)	0.2168	0.1325	0.3041	0.4405	1,674	0.2655 (+90%)	0.2911	0.1544	0.2934	0.4071	158
Baseline	0.1524	0.0920	0.0957	0.2086	0.3501		0.1401	0.1392	0.0937	0.1715	0.2841	
RM	0.2128 (+45%)	0.1707	0.1286	0.2669	0.4009	1,459	0.1999 (+50%)	0.2179	0.1244	0.2200	0.3392	156
Baseline	0.1463	0.1165	0.0949	0.1755	0.3103		0.1337	0.1795	0.0872	0.1296	0.2352	
ROCC	0.2143 (+44%)	0.1794	0.1206	0.2600	0.3920	1,449	0.1604 (+32%)	0.1656	0.0954	0.1557	0.2964	151
Baseline	0.1486	0.0994	0.0962	0.1935	0.3320		0.1219	0.1126	0.0861	0.1290	0.2380	

Retrieval Accuracy for B^-

	MAP (Imp.)	P@1	P@5	R@5	R@10	$\|B^-\|$	MAP (Imp.)	P@1	P@5	R@5	R@10	$\|B^-\|$
SCP	0.1314 (-40%)	0.0681	0.0821	0.1965	0.3046	955	0.0901 (-51%)	0.0495	0.0554	0.0988	0.1833	101
Baseline	0.2205	0.1853	0.1087	0.2723	0.3870		0.1850	0.1683	0.1149	0.2247	0.3140	
RM	0.1228 (-35%)	0.0577	0.0704	0.1659	0.2902	1,057	0.0932 (-38%)	0.0306	0.0510	0.1061	0.2318	98
Baseline	0.1898	0.1258	0.0986	0.2556	0.3870		0.1508	0.0816	0.1061	0.2032	0.3053	
ROCC	0.1372 (-34%)	0.0764	0.0803	0.1925	0.3110	942	0.1397 (-30%)	0.0988	0.0914	0.1874	0.2928	81
Baseline	0.2065	0.1741	0.1011	0.2474	0.3588		0.1988	0.2222	0.1259	0.2466	0.3230	

Retrieval Accuracy for B^o

	MAP	P@1	P@5	R@5	R@10	$\|B^o\|$	MAP	P@1	P@5	R@5	R@10	$\|B^o\|$
SCP	0.2682	0.2390	0.0767	0.3220	0.3416	1,406	0.1426	0.1313	0.0343	0.1616	0.1667	99
RM	0.2823	0.2396	0.0852	0.3530	0.3776	1,519	0.1857	0.1538	0.0558	0.2468	0.2548	104
ROCC	0.2634	0.2184	0.0835	0.3337	0.3752	1,644	0.1622	0.1349	0.0524	0.2090	0.2460	126

achieves a significant improvement on the Chrome project at this confidence level.

Looking at the sizes of the query sets B^+, B^- and B^o, as presented in Table IV, we observe that SCP either preserves or improves the performance for 76% of the 4,393 queries for the analyzed projects. Additionally, Table IV shows that the SCP method improves a larger number of queries in comparison to the other two QR methods. Notice that the amount of improvement is also very large for the positive queries with SCP, becoming as large as 66% for the Eclipse project and 90% for the Chrome project in terms of MAP.

2) Answer to Question2: This question is regarding the effectiveness of the two natural-language based approaches to QR, the one based on Relevance Model (RM) and the other based on using the Rocchio's formula (ROCC). Based on the results presented in Table III, we conclude that these do improve the retrieval results for bug localization. Of the two, RM is a robust performer; it achieves a significantly superior retrieval accuracy over the baseline for both projects. However, the improvements obtained via ROCC are not significant on the Chrome project. The main difference between these two methods is that RM incorporates the ranking scores of the files in the first pass retrieval into the QR process. With the help of these scores, the higher the rank of a file in the first pass retrieval, the higher the weights of the terms in these files for QR. Note that because of this weighting, the performance of RM is also less sensitive to the number of feedback files X.

3) Answer to Question3: In order to answer this question, we analyze the sets B^+, B^- and B^o for their differences in terms of the retrieval performance. As shown in Table IV, the initial retrieval performance has a significant correlation with the QR performance. Note that the baseline retrieval accuracy for the B^+ queries is significantly lower compared to that of the B^- queries for all QR methods. From these results, we may conclude that QR improves poorly performing queries, while it may lead to retrieval degradation for the queries that are performing relatively well to begin with. Also note that a large number of queries preserve their initial performance and the average retrieval accuracies of those queries are also high. Especially, P@1 for Eclipse is significantly high for these sets.

4) Answer to Question4: With regard to this question, Table V presents the mean values for the QPP metrics together with the average query lengths ($|Q|$) and the average number of Relevant Files (#RF) per bug for the respective query sets. We only present these metrics for the query sets as obtained via the new SCP method for lack of space. However, we obtained similar results for all three methods. We use the '†' symbol to indicate the statistical significance in the differences

TABLE V: *Query Statistics on the query sets as constructed by the SCP method*

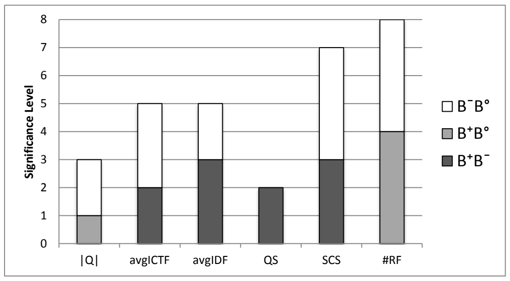
Set	Eclipse						Chrome					
	$\|Q\|$	avgICTF	avgIDF	QS	SCS	#RF	$\|Q\|$	avgICTF	avgIDF	QS	SCS	#RF
B^+	5.80	10.19††	3.94†††	0.49††	7.78†††	3.09	6.35	11.38	4.03	0.53	8.84	5.17††
B^-	5.90**	10.02††***	3.80†††**	0.51††	7.57†††****	3.69****	6.14	11.22*	3.98*	0.51	8.73	3.13††*
B^o	5.72**	10.20***	3.91**	0.50	7.83****	1.73****	6.44	11.61*	4.27*	0.50	9.10	2.36*

between the sets B^+ and B^-, and the '$*$' symbol for the same purpose between the sets B^- and B^o. The number of symbols indicates the level of significance at 10%, 5%, 1% and 0.1% respectively. That is, one such symbol indicates a 10% level of significance, two symbols indicate 5%, and so on. Additionally, Fig. 5 shows the significance levels of the differences for a pairwise comparison on the query sets. The heights of the bars in these plots give the significance level i.e. the number of symbols. We do not show the significance levels for the pairwise comparison of the sets B^+ and B^o in Table V in order to keep the table uncluttered. See Fig. 5 for these differences.

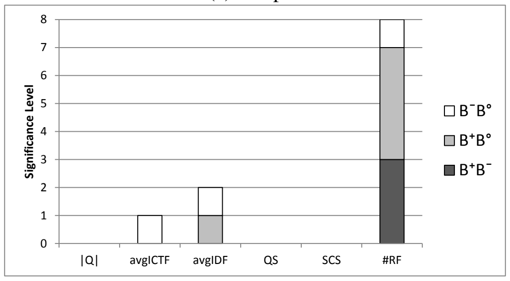
Comparing the queries in B^+ and B^- for Eclipse, we see that the differences in query lengths and the number of relevant files are *not* significant while the differences between the other metrics are. Based on these differences, we conclude that reformulating non-discriminatory queries that consist primarily of generic terms leads to query degradation for this project. These results are not surprising since the terms extracted by SCP become unpredictable when there is no pattern in the term distribution of the first pass retrieval results.

The number of relevant files (#RF) has a prominent effect on SCP based QR for both projects. Note that the bug fixes for Chrome touch a relatively higher number of files in comparison with Eclipse and this nature of the bug fixes is the most important factor that distinguishes the three query sets for this project. We observe that #RF for B^+ queries are significantly higher than those for the other sets for Chrome, indicating that our SCP based QR is more likely to improve the retrieval accuracy for the bugs for which the fixes have to touch a high number of files on this project. For B^o queries, on the other hand, #RF is the lowest among the three sets for both projects. Therefore, we conclude that our SCP based QR is more likely to preserve the query performance than to decrease it when the number of relevant files are lower than the average for the analyzed projects.

5) Remarks: The retrieval results for the query sets B^+, B^- and B^o reveal many important characteristics of QR for bug localization. These results clearly show that QR with the evaluated methods improves poorly performing queries. Obviously, then, we can claim that the presented QR framework helps out when it is most needed by a user. Additionally, looking at Table IV, we observe that if the original query is already performing well, QR is likely to just preserve the accuracy rather than to decrease it on average. This is evident from the fact that compared to the negative queries, the number of the neutral queries is larger and the average retrieval accuracies of those queries are also high. Based on the differences presented in Table V, we may conclude that these properties of QR are more likely to occur as the discriminatory power of the queries increases.



(a) Eclipse



(b) Chrome

Fig. 5: *Significance levels on the differences between the query sets obtained with the new SCP method. The heights of the bars indicate how significant the difference is: Longer bars mean more significant at the significance levels of 10%, 5%, 1% and 0.1%.*

VII. Threats to Validity

A drawback of our query reformulation approach is the number of parameters that need to be tuned to reach the best possible accuracy. The number of feedback files and the feedback terms, the interpolation parameter, and the window size parameter play a critical role in the reformulation of the queries. Training these parameters using a held-out query set may be a feasible approach. However, the quality of the retrievals with such an approach would obviously depend much on the queries chosen for training.

Another important point to mention is that, in order to evaluate the proposed approach, we used the titles of the bug reports filed in the past as natural text queries and the set of files modified to fix the corresponding bugs as our "ground-truth". Although we linked a large number of bug reports to the actual modifications in the software repositories, the extracted set of queries may be biased since not all bug fixes are clearly indicated in the repository logs [27].

VIII. Conclusion & Future Work

Retrieving files and documents from a source code library is particularly challenging for a novice user on account of the fact that developers use made-up words as identifiers for variables, method names, class names, etc. A user not already familiar with the library would have a difficult time conjuring up these names. The challenge then becomes how to use what is likely to be a weak initial query as a jumping off point for a stronger query. Our Query Reformulation (QR) approach solves this problem effectively.

Our novel QR framework exploits spatial proximity between the terms in source code files in order to decide how to reformulate a given query to increase retrieval effectiveness. The proposed method examines the files retrieved for the initial query supplied by a user and then selects from these files only those additional terms that are in close proximity to the terms in the initial query. Our experimental evaluation on two large software projects using more than 4,000 queries showed that the proposed approach leads to significant improvements for bug localization and outperforms the well-known QR methods in the literature. As is true of all information retrieval methods based on statistical modeling of underlying database, on occasion a query that yields good results as originally formulated may lead to not-so-good results after it has been reformulated by the approach presented in this paper. Nonetheless, on the average, a user interacting with a retrieval engine fitted with our query reformulation framework would be much better with our QR approach than without it.

Future work includes the evaluation of different term proximity metrics in source code. We believe imposing a functional distance metric may lead to even better QR for source code retrieval. We also plan to analyze the effect of stop-word removal on the proximity based metrics as it changes the positional distributions of the terms in the software artifacts.

Acknowledgment

This work was supported by Infosys.

References

[1] B. Dit, M. Revelle, M. Gethers, and D. Poshyvanyk, "Feature location in source code: a taxonomy and survey," *Journal of Software: Evolution and Process*, vol. 25, no. 1, pp. 53–95, 2013.

[2] B. Sisman and A. Kak, "Incorporating version histories in information retrieval based bug localization," in *Proceedings of the 9th Working Conference on Mining Software Repositories (MSR'12)*. IEEE, 2012, pp. 50–59.

[3] 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 *34th International Conference on Software Engineering (ICSE'12)*. IEEE, 2012, pp. 14–24.

[4] S. Rao and A. Kak, "Retrieval from software libraries for bug localization: a comparative study of generic and composite text models," in *Proceedings of the 8th Working Conference on Mining Software Repositories (MSR'11)*. ACM, 2011, pp. 43–52.

[5] S. Lukins, N. Kraft, and L. Etzkorn, "Source code retrieval for bug localization using Latent Dirichlet Allocation," in *15th Working Conference on Reverse Engineering (WCRE'08)*. IEEE, 2008, pp. 155–164.

[6] F. Deissenboeck and M. Pizka, "Concise and consistent naming," *Software Quality Control*, vol. 14, no. 3, pp. 261–282, Sep. 2006.

[7] T. Biggerstaff, B. Mitbander, and D. Webster, "Program understanding and the concept assignment problem," *Communications of the ACM*, vol. 37, no. 5, pp. 72–82, 1994.

[8] G. Gay, S. Haiduc, A. Marcus, and T. Menzies, "On the use of relevance feedback in IR-based concept location," in *International Conference on Software Maintenance (ICSM'09)*, sept. 2009, pp. 351 –360.

[9] J. Rocchio, "Relevance feedback in information retrieval," 1971.

[10] A. Marcus, A. Sergeyev, V. Rajlich, and J. I. Maletic, "An information retrieval approach to concept location in source code," in *Proceedings of the 11th Working Conference on Reverse Engineering (WCRE'04)*. IEEE Computer Society, 2004, pp. 214–223.

[11] J. Yang and L. Tan, "Inferring semantically related words from software context," in *Proceedings of the 9th Working Conference on Mining Software Repositories (MSR'12)*. IEEE, 2012, pp. 161 –170.

[12] V. Lavrenko and W. Croft, "Relevance based language models," in *Proceedings of the 24th annual international ACM SIGIR conference on Research and development in information retrieval*. ACM, 2001, pp. 120–127.

[13] G. Cao, J. Nie, J. Gao, and S. Robertson, "Selecting good expansion terms for pseudo-relevance feedback," in *Proceedings of the 31st annual international ACM SIGIR conference on Research and development in information retrieval*. ACM, 2008, pp. 243–250.

[14] D. Kelly and J. Teevan, "Implicit feedback for inferring user preference: a bibliography," *SIGIR Forum*, vol. 37, no. 2, pp. 18–28, Sep. 2003. [Online]. Available: http://doi.acm.org/10.1145/959258.959260

[15] Y. Lv and C. Zhai, "A comparative study of methods for estimating query language models with pseudo feedback," in *Proceedings of the 18th ACM conference on Information and knowledge management*. ACM, 2009, pp. 1895–1898.

[16] ——, "Positional relevance model for pseudo-relevance feedback," in *Proceedings of the 33rd international ACM SIGIR conference on Research and development in information retrieval*. ACM, 2010, pp. 579–586.

[17] J. Miao, J. X. Huang, and Z. Ye, "Proximity-based Rocchio's model for pseudo relevance feedback," in *Proceedings of the 35th international ACM SIGIR conference on Research and development in information retrieval*. ACM, 2012, pp. 535–544.

[18] S. Robertson and K. Spärck Jones, "Simple, proven approaches to text retrieval," University of Cambridge, Computer Laboratory, Tech. Rep. UCAM-CL-TR-356, Dec. 1994. [Online]. Available: http://www.cl.cam.ac.uk/techreports/UCAM-CL-TR-356.pdf

[19] S. Haiduc, G. Bavota, R. Oliveto, A. De Lucia, and A. Marcus, "Automatic query performance assessment during the retrieval of software artifacts," in *Proceedings of the 27th International Conference on Automated Software Engineering (ASE'12)*. ACM, 2012, pp. 90–99.

[20] B. He and I. Ounis, "Inferring query performance using pre-retrieval predictors," in *Proc. Symposium on String Processing and Information Retrieval*. Springer Verlag, 2004, pp. 43–54.

[21] R. Wu, H. Zhang, S. Kim, and S. Cheung, "Relink: recovering links between bugs and changes," in *Proceedings of the 19th ACM SIGSOFT symposium and the 13th European conference on Foundations of software engineering (ESEC/FSE'11)*. ACM, 2011, pp. 15–25.

[22] V. Dallmeier and T. Zimmermann, "Extraction of bug localization benchmarks from history," in *Proceedings of the 22nd IEEE/ACM international conference on Automated software engineering (ASE'07)*. ACM, 2007, pp. 433–436.

[23] T. Zimmermann and P. Weißgerber, "Preprocessing cvs data for fine-grained analysis," in *Proceedings of the 1st International Workshop on Mining Software Repositories (MSR'04)*, 2004, pp. 2–6.

[24] M. Porter, "An algorithm for suffix stripping," *Program: electronic library and information systems*, vol. 14, no. 3, pp. 130–137, 1980.

[25] C. Macdonald, B. He, V. Plachouras, and I. Ounis, "University of Glasgow at TREC 2005: Experiments in Terabyte and Enterprise tracks with Terrier," in *Proceedings of TREC 2005*, 2005.

[26] C. Manning, P. Raghavan, and H. Schutze, *Introduction to information retrieval*. Cambridge University Press Cambridge, 2008, vol. 1.

[27] A. Bachmann, C. Bird, F. Rahman, P. Devanbu, and A. Bernstein, "The missing links: Bugs and bug-fix commits," in *Proceedings of the 18th ACM SIGSOFT international symposium on Foundations of software engineering (FSE'10)*. ACM, 2010, pp. 97–106.