Automatic Query Reformulations for Text Retrieval in Software Engineering

Sonia Haiduc[1], Gabriele Bavota[2], Andrian Marcus[1], Rocco Oliveto[3], Andrea De Lucia[2], Tim Menzies[4]
[1]Wayne State University, Detroit MI, USA
[2]University of Salerno, Fisciano (SA), Italy
[3]University of Molise, Pesche (IS), Italy
[4]University of West Virginia, Morgantown VA, USA

***Abstract*—There are more than twenty distinct software engineering tasks addressed with text retrieval (TR) techniques, such as, traceability link recovery, feature location, refactoring, reuse, etc. A common issue with all TR applications is that the results of the retrieval depend largely on the quality of the query. When a query performs poorly, it has to be reformulated and this is a difficult task for someone who had trouble writing a good query in the first place.**

We propose a recommender (called *Refoqus*) based on machine learning, which is trained with a sample of queries and relevant results. Then, for a given query, it automatically recommends a reformulation strategy that should improve its performance, based on the properties of the query. We evaluated *Refoqus* empirically against four baseline approaches that are used in natural language document retrieval. The data used for the evaluation corresponds to changes from five open source systems in Java and C++ and it is used in the context of TR-based concept location in source code. *Refoqus* outperformed the baselines and its recommendations lead to query performance improvement or preservation in 84% of the cases (in average).

***Index Terms*—Text Retrieval, Query Reformulation.**

I. Introduction

Text Retrieval (TR) is one of the most popular technologies used in Software Engineering (SE), where it has been successfully applied to address more than twenty tasks [1], including: concept/feature/concern location, impact analysis, code retrieval and reuse, traceability link recovery, bug triage, requirements analysis, refactoring and restructuring, reverse engineering, defect prediction, coupling and cohesion measurement, etc. Many of these SE tasks are reformulated as TR tasks and involve the formulation of a text query (by the developer or automatically). For example, during feature location, a developer formulates a query which describes the feature to be located in the code. The query is then run by the TR technique and a list of ranked software artifacts (e.g., classes or methods) is retrieved.

For all such TR applications in SE, the performance of the retrieval depends greatly on the textual query and its relationship to the text contained in the software artifacts. Determining what is a good query is a nontrivial problem and it requires intimate knowledge of the vocabulary of the software artifacts to be searched. Ironically, developers could benefit from TR tools especially when such knowledge is missing. The obvious solution for determining if a query is good is for the developer to investigate the retrieved results and decide whether they are relevant or not. When the results are not relevant, the query is usually reformulated (i.e., words are added and deleted).

Rewriting a query in order to improve its performance (i.e., by better performance we mean retrieving the relevant documents closer to the top of the list of results) is often as difficult as writing the query in the first place. This problem has been recognized by SE researchers and two types of approaches have been proposed to assist developers with the query reformulation. The first category of approaches is based on *user relevance feedback* and it has been employed in the context of traceability link recovery [2] and concept location [3]. These reformulation techniques are interactive and rely on the developer analyzing the list of results and marking the top documents as relevant or not relevant. The documents marked by the user are then used to reformulate the query. A second class of approaches is based on automatically adding to the query new terms that are similar to its terms (e.g., synonyms) [4], [5], [6]. The main shortcoming of these approaches is that they ignore the properties of the queries and the reformulation strategy is the same for every query.

The performance of a query depends on many factors and we conjecture that queries with different properties need different reformulation strategies. For example, a query that has a single term will likely need an expansion strategy (i.e., adding terms) to improve its performance, whereas a verbose query may need a reduction strategy (i.e., removing terms).

In this paper, we propose and evaluate an automated recommender that, for a given query, it recommends a reformulation strategy that should improve its performance. We call the recommender *Refoqus* (**REF**orumulation **O**f **QU**erie**S**). *Refoqus* is based on machine learning and requires a training set comprised of queries and their relevant results. For each query, a set of measures are computed (see Section II for details), which capture properties of the query, such as, *specificity*, *coherency*, *similarity*, *term relatedness*, *robustness*, and *score distribution*. These measures have been shown to correlate with the performance of queries in the field of natural language document retrieval [7] and in SE applications [8]. We selected four reformulation strategies proposed in the field of natural language document retrieval (see Section II for details), which perform best in that field and have never been used in SE, yet they are appropriate for SE data. *Refoqus* automatically applies each reformulation strategy for the queries in the training set and learns which reformulation strategy works best for which type of query (based on the relevant properties). For an incoming query, *Refoqus* measures its properties and automatically recommends the appropriate reformulation strategy. The underlying algorithms of *Refoqus* are generic, so the measures and recommendation strategies can be replaced, if needed.

The *Refoqus* recommender is a premiere in SE, as well as in natural language document retrieval. Within SE, it is to date the only automatic query reformulation approach that employs multiple strategies and selects the best one for each query, as opposed to applying a single strategy to all queries.

We performed an empirical evaluation of *Refoqus*, in the context of TR-based concept location in source code. We collected 282 queries corresponding to changes in response to 94 bug reports from five open source systems, written in Java and C++. We compared the results of *Refoqus* with the baseline results provided by the four reformulation strategies on their own, respectively. We found that *Refoqus* outperformed the baselines and its recommendation lead to query performance improvement or preservation in 84% of the cases (in average). In addition, we investigated two ways to perform the training (i.e., *within* a single project vs. *cross* projects) and found that the *within* training is superior. The results support our conjecture that the best reformulation strategies are project and query specific. *Refoqus* also proves to be robust with respect to its training data, as we observed little differences between its performances on different systems. More than that, a relatively modest number of training data (i.e., 57 queries corresponding to 19 bugs per system, in average) is sufficient for good results.

II. Query Properties and Reformulation Strategies Used by Refoqus

We introduce terminology and definitions necessary to understand the measures and reformulation strategies used by *Refoqus*. Most of these measures and strategies come from the field of natural language document retrieval (a.k.a., text retrieval), whereas some are used in SE applications only.

A. Query Reformulation Techniques

The goal of query reformulation is to define a new query, starting from the initial one, which is able to lead to improved retrieval results. What exactly "good search results" means can differ according to context in which the search is used, but it usually refers to the relevant documents being as close as possible to the top of the search results list. This is the interpretation of quality performance we adopt in this work.

Over time, researchers in the field of TR have proposed and investigated a large variety of approaches for producing candidate reformulations for an initial query. These approaches fall in two categories [9]: *query expansion* approaches and *query reduction* approaches. We introduce briefly each category with emphasis on the reformulation strategies used in our proposed approach.

1) Query Expansion: Query expansion is meant to offer a solution to the problem known as "*the vocabulary problem*" [10], where the terms in the query do not match the vocabulary of the relevant documents in the corpus. A variety of query expansion approaches have been proposed in the field of TR [11]. We found, however, that not all these were applicable to our circumstances (i.e., source code based corpora). We selected three existing approaches in the following way. We did not consider approaches that relied on linguistic features or on sources of information external to the corpus, like the web, ontologies, Wikipedia, or Wordnet. Such approaches are designed to work for natural language documents as they rely on word relationships that exist in natural language. Since we target source code-based corpora and previous studies [12] have shown that words do not share the same relationships in source code as they do in natural language, we decided to ignore such strategies (in this work at least). Some approaches are based on algorithms with high computational complexity to produce reformulations for a query. Since our end goal is to produce a recommender which can be used by developers during their daily tasks, we did not consider such approaches practical and thus, we did not select them. Finally, from all other available strategies we selected seven strategies that are reported to perform best in the TR literature. We performed a study on SE data[1]and selected the best three to be used by *Refoqus*. Note that *Refoqus* is built in such a way that any strategy can be replaced or additional ones can be added. We decided to use fewer strategies because the classification with more strategies would require larger training data sets, which impacts the usability of *Refoqus*.

All three selected strategies are based on some form of *pseudo-relevance feedback*, in that they consider the top *K* documents from the list of results as relevant documents to the query. Then they use different techniques to order the terms in these *K* documents and select the top *N* ones to use for the query expansion.

The first strategy is similarity-based and orders the terms in the top *K* documents based on their *Dice similarity* (see below) with the individual query terms.

$$Dice = \frac{2df_{u \wedge v}}{df_u + df_v}$$

where *u* is a term from the query, *v* is a term from the top *K* documents, and *df* denotes the number of documents in the corpus containing *u*, *v*, or both *u* and *v*, respectively.

The idea behind Dice similarity is that two terms are related if they appear in the same documents in the corpus, a common assumption in all TR engines.

The other two techniques do not rely on similarities with the terms in the query. The idea is to use the first *K* documents retrieved in response to the original query as a more detailed description of the underlying query topic. Therefore, descriptive terms for this topic can be used for expansion, and can be determined by identifying the most representative terms for the set of top retrieved documents.

[1]http://www.cs.wayne.edu/~severe/ICSE2013

One of the approaches is based on Rocchio's method for relevance feedback [13] and assigns a score to each term in the top K documents based on the sum of the *Tf-Idf* scores of the term in each of the K documents. *Tf-Idf* is a score often used in the field of TR to determine the importance of a term for a particular document relative to the corpus. The first component of the measure is the Term Frequency (*Tf*), which is the number of times the term appears in a document and it is an indicator of the importance of the term in the document compared to the rest of the terms in that document. The Inverse Document Frequency (*Idf*), on the other hand, is the inverse of the number of documents in the corpus containing that term and indicates the specificity of that term for a document containing it.

$$Rocchio = \sum_{d \in R} TfIdf(t,d)$$

where *R* is the set of top K relevant documents in the list of retrieved results, *d* is a document in R, and *t* is a term in d.

The last approach uses the Robertson Selection Value (*RSV*), described below, as an ordering function for the terms in the top K documents.

$$RSV = \sum_{d \in R} TfIdf(t,d)[p(t|R) - p(t|C)]$$

where *C* denotes the collection of documents in the corpus, *R* is the set of top K relevant documents in the list of retrieved results, *d* is a document in R, and *t* is a term in d. Also,

$p(t|R)$ = *freq(t in R) / number of terms in R*

$p(t|c)$ = *freq(t in C) / number of terms in C* where *freq (t in R)* is the number of times t appears in the top K documents in the list of results (R) and freq(t in C) is the number of times t appears in the whole document collection.

RSV also uses *Tf-Idf* as part of its formula, but considers in addition the probability of a term occurring in a relevant document in order to determine its importance for the query topic (i.e., for the top K documents).

2) Query Reduction: Query reduction is based on the idea that the query contains both important information as well as noise, i.e., words that do not contribute to the main intent of the query and may hinder the retrieval of relevant documents. Therefore, query reduction should help improve the results of a query. In the absence of user feedback and information about the semantics of the query, automated query reduction needs to be done with care, as intrusive reduction strategies may actually harm the results [9].

We adopt a conservative reduction strategy, previously used in SE, in the context of user-provided relevance feedback [3]. More specifically, *Refoqus* eliminates the terms that appear in more than 25% of the documents in the corpus, as they are considered non-discriminating [3].

B. Query Properties

Measures of query properties have been used in the field of TR for assessing the query performance [7] and have been shown to correlate with the mean average precision of a query. These properties focus on linguistic and statistical properties of the terms in the query and of the documents returned in the result list. The properties fall in two main categories, depending if they are computed based on information available before or after the retrieval of the results takes place. The two categories and the specific measures we use from each category in our approach are presented in the following subsections.

1) Pre-Retrieval Properties and Measures: Pre-retrieval measures are computed before the query is run, and measure linguistic and statistical properties of the query (e.g., coherence) and its relationship with the document collection (e.g., the similarity between the query and the entire document collection). They are considered lightweight, as they do not require the list of results to be computed. Pre-retrieval measures assess different properties of a query: *specificity*, *similarity*, *coherency*, and *term relatedness* [7].

Specificity refers to the ability of the query to represent the current information need and discriminate it from others. A query composed of non-specific terms, i.e., commonly used in the collection of documents, is considered having low specificity, as it is hard to differentiate the relevant documents from non-relevant ones based on its terms. For example, when searching source code, the query "initialize members" could have low specificity if a comment containing this text would be found in most class constructors in a system.

The *similarity* between the query and the entire document collection is another property that reflects an aspect of query quality. The argument behind this type of measure is that it is easier to retrieve relevant documents for a query that is similar to the collection since high similarity potentially indicates the existence of many relevant documents to retrieve from.

Another property for queries is their *coherency*, which measures how focused a query is on a particular topic. The coherency of a query is usually measured as the level of inter-similarity between the documents in the collection containing the query terms. The more similar the documents are, the more coherent the query is.

Finally, *term relatedness* measures make use of term co-occurrence statistics in order to assess the performance of a query. The terms in a query are assumed to be related to the same topic and are, thus, expected to occur together frequently in the document collection.

Refoqus uses 21 measures that capture the four pre-retrieval quality properties mentioned above: Average Inverse Document Frequency, Maximum Inverse Document Frequency, Standard Deviation of the Inverse Document Frequency, Average Inverse Collection Term Frequency, Maximum Inverse Collection Term Frequency, Standard Deviation of the Inverse Collection Term Frequency, Average Entropy, Median Entropy, Maximum Entropy, Standard Deviation of the Entropy, Query Scope, Simplified Clarity Score, Average Variance of Query Term Weights, Maximum Variance of Query Term Weights Sum of the Variance of Query Term Weights, Coherence Score, Average Similarity Collection-Query Term, Maximum Simi-

TABLE I
POST-RETRIEVAL MEASURES USED BY *Refoqus*

Name	Description
Subquery Overlap	Captures the extent of the overlap between the result set retrieved by the entire query and the result sets retrieved by individual query terms.
Robustness Score	The terms' weights in the top relevant documents are slightly perturbed and the resulting documents are re-ranked. The correlation between the initial rank and that after modification is considered.
First Rank Change	Captures the probability of a document found on the first position in the list of results to still remain on the first position after a perturbation is applied to it.
Clustering Tendency	Measures the cohesion of the top retrieved documents as the textual similarity between them.
Spatial Autocorrelation	Changes the retrieval-scores of each top relevant document as the average of the scores of its most similar documents. Then, the linear correlation of the new scores with original ones is used.
Weighted Information Gain	Measures the divergence between the mean retrieval score of top-ranked documents and that of the entire corpus.
Normalized Query Commitment	Measures the standard deviation of retrieval scores in the list of top relevant documents, normalized by the score of the whole collection.

larity Collection-Query Term, Sum of Similarity Collection-Query Term, Average Pointwise Mutual Information, and MaxPMI. We have previously introduced the usage of these measures in SE and have successfully applied them for predicting the quality of queries in the context of concept location in source code [14], [8]. Four of these measures were introduced specifically in SE context [14], [8], whereas the others come from TR [7]. More details about what each of the measures captures and the formulas used to compute them can be found in [7], [8], [14].

2) Post-Retrieval Properties and Measures: Pre-retrieval measures miss some aspects of the query, which are reflected in the results that it returns. For example, the coherence of the search results, i.e., how focused they are on aspects related to the query, is not captured by the query text and is hard to assess it without an analysis of the results list.

Post-retrieval measures rely on the analysis of the search results, that is, the list of documents ranked highest in response to the query. These measures are categorized into three main paradigms [7].

Robustness-based methods evaluate how robust the results are to perturbations in the query and the documents in the result list. The measures based on query perturbation assess the robustness of the result list to small modifications of the query. When small changes in the query result in large changes in the search results, the confidence in the capacity of the query to capture the essential information diminishes. Document perturbation measures, on the other hand, rely on injecting the top documents in the result list with noise and re-ranking them, measuring the difference in their ranks before and after the perturbation. In the case of a high performing queries, small perturbations of the documents in the result list should not result in significant changes in their ranking.

Score distribution-based methods analyze the similarity between the query and the results, which are used to rank the results of the retrieval. For example, the highest retrieval score (i.e., similarity) and the mean of top scores indicate query performance since, in general, low scores of the top-ranked documents indicate some difficulty in retrieval.

Clarity-based methods directly measure the "focus" of the search results with respect to the corpus. While we experimented with clarity-based quality measures, we did not use them in our approach due to their extended execution times, which would make them unpractical in a realistic setting.

Refoqus uses seven post-retrieval measures of robustness and score distribution: Subquery Overlap, Robustness Score, First Rank Change, Clustering Tendency, Spatial Autocorrelation, Weighted Information Gain, and Normalized Query Commitment. They are defined in TR [7] and their use in SE context is a premiere. A brief explanation of each of these measures can be found in Table I. Details about their implementation can be found in [7].

Refoqus uses these 28 measures (21 pre- and 7 post-retrieval) to classify the queries. Obviously, some of these measures are related and correlate to some degree. The classifier used by *Refoqus* (see the next Section for details) employs implicitly a feature selection mechanism that eliminates the properties that are not relevant.

III. REFOQUS - A RECOMMENDER FOR AUTOMATIC QUERY REFORMULATION

Refoqus consists of two main steps: (1) training the classifier; and (2) using the classifier to recommend the best reformulation technique for incoming queries. The following subsections detail each step.

A. Training the Classifier

Refoqus needs a training data set for its classifier. The training data consists of queries and their associated relevant documents. How queries are collected depends on the SE task addressed. Ideally (as the evaluation shows in the next Section), the data should come from the system where *Refoqus* is being used. For example, if the task at hand is traceability link recovery between documentation and source code, the training data would consist of existing validated traceability links, where parts of the documentation are the queries and the relevant documents are the parts of the code they link to.

Refoqus communicates with the TR engine used by the developer. In the current implementation (which we used in the empirical evaluation from the next Section), we used Lucene[2], a popular implementation of the Vector Space Model. Future versions will allow *Refoqus* to work with other TR engines.

[2]http://lucene.apache.org

Refoqus executes the following steps in order to train its classifier:

- *Refoqus* uses the TR engine to rank all the relevant documents for each query in the training data set.
- The values of the 28 query property measures are computed for each of the queries in the training data.
- The four reformulation techniques are applied, one at a time, to each query in the training set and the resulting reformulated queries are run by the TR engine.
- The results obtained by the four reformulation variants are compared and the best performing reformulation is determined for each query.
- If there are queries that led to no relevant document being retrieved by the TR engine after they were run in their original form and in any of the reformulated forms, then these queries are removed from the training set. This is a necessary step, as for such queries Refoqus will not be able to make any recommendation, given that it cannot decide which is the best reformulation strategy.
- The classifier is trained using the collected training data. One data point in the final training data used by the classifier corresponds to a query. Each data point has 29 attributes, 28 attributes corresponding to the query property measures and one corresponding to the best reformulation strategy. We discuss bellow the details of the classifier employed by *Refoqus*.

In *Refoqus* we use *classification trees* [15]. Our choice has several advantages. First, the rules produced by classification trees are easy to understand by humans, which is not the case for other, more complex models. Hence, a developer could interpret easily the recommendation made by Refoqus, before allowing it to automatically reformulate the query, if she chooses to do so. Second, classification trees perform implicitly feature selection. This is a very important property, as it allows Refoqus to be less sensitive to the choice of query property measures. In the current form, it allows us to give as input all 28 measures of a query, as our classification tree will determine automatically the properties relevant for the classification, with little overhead.

Classification trees are suitable to solve problems where the goal is to determine the values of a categorical variable based on one or more continuous and/or categorical variables. In our approach, the categorical dependent variable is represented by the best query reformulation technique for a particular query, while the independent variables are the 28 query property measures described in Section II. The classifier uses the training data to automatically select the independent variables and their interactions that are most important in determining the dependent variable to be explained.

There are two possible approaches to train the classifier, namely *within-project* and *cross-project* training. In the former approach, the classifier is trained independently for each software system, thus using only the training data from one single system. Our evaluation from Section IV indicates that the *within* training is superior to the *cross* training, yet *cross* training is useful when training data may not be obtained from the current system.

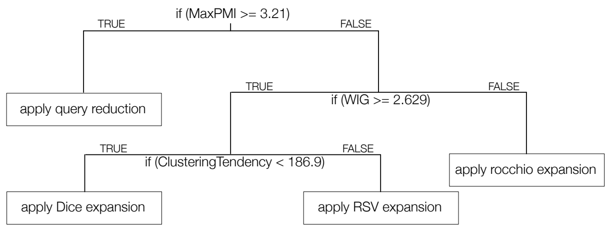

Fig. 1. An example of classification tree.

The output of the training stage is the classification tree, represented by a set of yes/no questions that splits the training sample into gradually smaller partitions that group together cohesive sets of data, i.e., those having the same value for the dependent variable. An example of classification tree built in our study is reported in Figure 1.

B. Using the Classifier for New Queries

Once the classification tree is built, it can be used to recommend the best reformulation technique for a given query. When a new query is issued (manually or automatically) to the TR engine, which returns the results, *Refoqus* computes the 28 measures for the new query. Based on the classification tree and these 28 measures, *Refoqus* determines automatically which reformulation strategy should be applied to the new query and it recommends it to the developer. The recommended reformulation technique is then automatically applied to add and/or remove terms from the query in order to improve its performance.

IV. Evaluation

We conducted an empirical study to investigate the performance of *Refoqus* in the context of TR-based concept location. The replication package, containing also the dataset used in the study is available online[3].

A. Context

According to a recent survey on feature/concept location [16], most techniques make use of TR. We decided to evaluate *Refoqus* in a manner akin to the evaluation of concept location techniques. In addition, a prior user-based reformulation technique was also evaluated in the same context [3].

TR-based concept location [5], [16] is mostly used during software change. During concept location, the developer gets a change request and based on it formulates a query to find a place in the code that she will need to change. If the query does not retrieve a relevant code document (e.g., a method or a class), then the query needs to be reformulated. Once the place of change is located, the developer engages in impact analysis to determine other parts of the code that need to change.

B. Definition

There are several aspects of *Refoqus* that we want to evaluate. First, we want to establish which training strategy (i.e., *within-* or *cross-project*) works better. Second, we want to

[3]http://www.cs.wayne.edu/~severe/ICSE2013

TABLE II
CHARACTERISTICS OF THE FIVE OBJECT SYSTEMS.

System	Version	Language	KLOC	#Methods	#Queries	#Bugs
Adempiere	3.1.0	Java	330	28,355	51	17
ATunes	1.10.0	Java	80	3,481	51	17
FileZilla	3.0.0	C++	410	8,012	72	24
JEdit	4.2	Java	250	5,532	54	18
WinMerge	2.12.2	C++	410	8,012	54	18
Total	-	-	1,310	48,620	282	94

establish whether the reformulations recommended by *Refoqus* improve the queries and if so by how much. Our conjecture is that the strength of *Refoqus* comes from the fact that it selects the best reformulation strategy for each query. Hence, third, we compare *Refoqus* with baseline approaches, based on the individual reformulation strategies used by *Refoqus*. In order to address these issues, we formulated three research questions and conducted three experiments to answer them:

RQ$_1$: *Which training approach works better for Refoqus?*
RQ$_2$: *Does Refoqus improve the performance of the queries?*
RQ$_3$: *Does Refoqus perform better than the baseline reformulation techniques?*

Answering RQ$_1$ allows us to determine and inform future users what is the best way to construct the training data. A positive answer for RQ$_2$ implies that *Refoqus* can be used to improve TR-based concept location approaches (and hopefully TR approaches for other SE tasks). A positive answer for RQ$_3$ confirms our conjecture that selecting the best reformulation strategy for each query is better than applying the same strategy to all queries.

We implemented *Refoqus* to interact with Lucene, to use a classification tree[4], to compute the 28 query property measures, and to apply the four reformulation strategies (as described in the previous section).

C. Data set

Our choice of empirical evaluation is based on reenacting concept location based on past changes. This is a very common evaluation technique used in feature/concept location research [16]. Past changes in software provide us with a change request (or bug description in this case) and the actual changes in the code done in response to the request, named the *change set*. During concept location a user or a tool starts with the change request and finds a place in the code where a change should be made. To verify that this location is correct, the complete change should be implemented and tested. Reenactment based on historical data allows us to assess the correctness of concept location without complete implementation and testing. If concept location results in a place in the code that is in the original change set, then we can conclude that concept location succeeded. If the result of the concept location leads to a place that is not in the change set, then we consider that concept location failed. Changes to software can be made in a variety of ways, so there may be some cases when concept location leads to a place that is not in the original change set, yet could still lead to a complete and correct change. Our assumptions will cause to miss these cases. It is a trade-off we are willing to take given the gains in terms of time and number of changes. This trade-off is commonly undertaken in the field [16].

Reenactment also allows us to automatically formulate queries for TR-based concept location. The bug reports contain both the title of the bugs (a.k.a. the short description) and their (long) description. In this study we automatically created queries considering two different options: (i) the title of the bug; (ii) the description of the bug. In addition, to have a better simulation of an usage scenario of the proposed approach, we also asked a Ph.D. student to manually formulate a query after analyzing only the bug report content. In the end, we obtained three queries for each bug report. For each query formulated for a bug report, the set of relevant documents to be retrieved is defined by the *change set*. The same data set is used when answering each research question. We collected an initial set of 309 queries, corresponding to 103 bugs randomly extracted from the bug tracking systems of five open source systems implemented in Java and C++: Adempiere[5], ATunes[6], FileZilla[7], JEdit[8], and WinMerge[9].

We removed the queries for which no target method was retrieved when running the original query and all of its four reformulated forms (see Section III for details). The data set was reduced to 94 bugs and their corresponding 282 queries. From this point on, we will refer only to these remaining 282 queries. The number of queries extracted from each project are reported, together with some size attributes of the object systems, in Table II. Note that when using other TR engines (e.g., LSI) this filtering set may not be necessary, as some engines rank all the documents from the search space.

D. Planning and Execution

In order to generate term suggestions for query expansion, we used the top five documents in the ranked list of results. Also, when expanding the query, we considered the first 10 term suggestions. These decisions were made based on recommendations found in the domain literature [11]. After the collection of the data, we performed the following steps:

1. *Document corpus creation*. We built the source code corpus by considering each method in the system as a separate document. For each method, we extracted the terms found in its source code identifiers and comments. We then normalized the text using identifier splitting (we also kept the original identifiers), stop words removal (i.e., we removed common English words and programming keywords), and stemming (we used the Porter stemmer).

2. *Query execution and performance measurement.* We performed the same text normalization process adopted for the methods on all the 282 queries and their reformulations. Then, we executed each query on their respective document corpus by using Lucene and measured the query performance

[4]We used the implementation of classification trees provided in the statistical platform R.

[5]http://www.adempiere.org
[6]http://www.atunes.org
[7]http://www.filezilla-project.org
[8]http://www.jedit.org
[9]http://www.winmerge.org

by identifying the position of the first relevant document (i.e., changed method) in the ranked list of search results. The higher the method appears in the result list (i.e., the lower its rank - 1 is best), the better the query performance. This is a common measure used to assess the results of concept location, called *effectiveness*. It represents an approximation of the effort it takes to locate a concept,assuming each document takes a unit of effort to investigate. Effectiveness is one of the most common measures used in empirical studies on comparing concept and feature location techniques [16].

3. *Answering RQ_1*. Refoqus was trained using the *within-* and *cross-project* strategy, respectively. For the *within-project* case, the classification model is trained on each system individually and a 4-fold cross-validation was performed: (i) randomly divide the set of queries for a system into 4 approximately equal subsets, (ii) set aside one query subset as a test set, and build the classification model with the queries in the remaining subsets (i.e., the training set), (iii) use the classification model built on the training set to identify the best reformulation technique for the queries in the evaluation set, (iv) repeat this process, setting aside each query subset in turn. The key element here is that each query is used only once in the test set.

As for the *cross-project* training, the queries from four of the five projects are used for training and the queries from the fifth project is used for evaluation. This is repeated such that the queries in each project are tested.

The 282 queries were reformulated and the performance (i.e., the best rank among the methods in the change set) of the reformulated queries was recorded for each type of training. The two sets of performances were then compared.

4. *Answering RQ_2*. The performance of the reformulated queries based on the *Refoqus*' recommendation were compared with the performance of the original queries.

5. *Answering RQ_3*. We defined four baselines using the reformulation strategies employed by *Refoqus*: *query reduction*, *rocchio expansion*, *RSV expansion*, and *Dice expansion*. Each baseline approach applies a single reformulation strategy to all 282 queries, respectively. For example, the *reduction* baseline applies *query reduction* to all queries.

In order to analyze the comparisons, when comparing *Refoqus* with any of the baselines (or when comparing the two training strategies), we report the number of times the query reformulated by *Refoqus* and by the compared baseline has a better performance (i.e., lower rank of the top changed method) than the original query, the number of times the performances are the same, and the number of times the original query achieves better query performance. We also report the minimum, maximum, median, mean, and the 25% and 75% percentiles values of the differences in performance (i.e., difference in ranking of the top changes method). See Tables III, V, and IV for details.

The sets of results were also analyzed through statistical analysis using the Mann-Whitney test [17]. We chose this test as we cannot assume normality of data and the test does not make normality assumptions. The results are interpreted as statistically significant at $\alpha < 0.05$. However, since we performed multiple tests, we adjusted our p-values using the Holm's correction procedure [18]. This procedure sorts the p-values resulting from n tests in ascending order, multiplying the smallest by n, the next by $n-1$, and so on.

V. Results and Discussion

We present and discuss the results that we used to answer each research question.

A. Research Question 1

Tables III and IV report the results achieved by *Refoqus* when building the classifier using the *within-project* training strategy and the *cross-project* training strategy, respectively. The *within-project* strategy achieves a mean query performance improvement of 262 positions (for 146 queries) and a maximum of 5,285, compared to the mean of 229 (for 113 queries) and the maximum of 5,197 obtained by the *cross-project* training strategy. However, the within-project approach also worsens the results of 6 more queries compared to the cross-project strategy, whereas the number of queries for which the results do not change is significantly higher for the cross-project approach, i.e., 128 compared to 89.

Since the results were mixed, we performed statistical analysis in order to determine if the difference between the results obtained by the two approaches is significant. The Mann-Whitney Test reports statistically significant differences between the performance values of the reformulated queries using the two approaches, in favor of the *within-project* training (p-value=0.002, mean=-40). A mean value of -40 indicates the *within-project* training returns the first relevant method 40 positions on average higher in the results list than the *cross-project* training.

RQ_1 answer. We conclude that the *within-project* training is superior to the *cross-project* training. Nonetheless, *cross-system* training for *Refoqus* still manages to improve the performance of 40% of the original queries, and to preserve the results of other 45%. This indicates that *cross-project* training could be still used, when *within-project* data is not available.

We use the *within-project* training strategy to answer the subsequent research questions.

B. Research Question 2

When compared to the performance obtained by original queries (see Table III), *Refoqus* using within-project training is able to improve the performance of 52% of the queries and to preserve the results obtained for other 32%, for a total of 235 out of 282 (84%) queries for which the performance is preserved or improved. This improvement is in several cases by hundreds or thousands of positions. When analyzing the results, it is important to focus on the performance in the worse cases, as these are the situations where *Refoqus* is most useful (i.e., when the original query is really bad). When the original query is already good (say, the best ranked method is in top 10), reformulations strategies in general lead to small improvements or no improvement. The rather large difference

TABLE III
RESULTS ACHIEVED BY *Refoqus* IN THE WITHIN-PROJECT VALIDATION

System	#Queries	#Improved	Improvement						#Worsened	Worsening						#Preserved
			Mean	Q1	Q2	Q3	Min.	Max.		Mean	Q1	Q2	Q3	Min.	Max.	
Adempiere	51	30	418	3	12	97	1	5,286	10	261	18	26	381	3	970	11
ATunes	51	29	85	5	9	86	1	667	10	54	5	40	100	1	324	12
FileZilla	72	42	383	7	163	611	1	1,409	7	90	10	21	106	1	371	23
JEdit	54	19	64	5	29	56	1	434	9	25	2	12	52	1	83	26
WinMerge	54	26	230	4	18	36	2	4,909	11	43	6	11	53	2	151	17
Total	282	146	262	4	23	166	1	5,286	47	100	5	19	86	1	970	89

TABLE IV
RESULTS ACHIEVED BY *Refoqus* IN THE CROSS-PROJECT VALIDATION

System used as evaluation	#Queries	#Improved	Improvement						#Worsened	Worsening						#Preserved
			Mean	Q1	Q2	Q3	Min.	Max.		Mean	Q1	Q2	Q3	Min.	Max.	
Adempiere	51	15	585	7	11	109	1	5,197	3	71	25	49	107	1	165	33
ATunes	51	25	62	3	9	51	1	413	6	107	41	51	148	4	319	20
FileZilla	72	32	275	11	157	425	1	1,403	12	105	22	27	158	2	437	28
JEdit	54	18	68	7	29	61	1	434	10	112	8	52	71	1	781	26
WinMerge	54	23	242	2	8	46	1	4,603	10	34	2	5	16	1	164	21
Total	282	113	229	4	15	157	1	5,197	41	87	5	28	96	1	781	128

between the median and mean improvements indicates that many ”bad” queries had large performance improvements.

We also performed a Mann-Whitney statistical test, which revealed that the difference between the effectiveness measure as returned by *Refoqus* and that returned by the original query is statistically significant (p-value$<$0.0001, mean = -119). In other words, on average, across all queries, *Refoqus* is able to obtain an improvement (i.e., a lower effectiveness measure) of 119 positions in the list of ranked results and this improvement is statistically significant.

RQ$_2$ answer. We conclude that the query reformulation recommendations formulated by *Refoqus* lead to the improvement or preservation of the query performances in most cases (52% of the queries improved their performance and 32% preserved it). We discuss some examples and observations from the detailed analysis of the data. An example of large improvement in query performance was observed on a query in the FileZilla system. The original query was automatically extracted from the title of the bug report: *set use medium large icon*. Using this query the first target document retrieved was the method `LoadPage` from the `COptionsPageThemes` class, on position 175. *Refoqus* suggested to apply the *Rocchio* expansion, and was reformulated as: *set use medium large icon theme panel scroll preview wx ptheme*. In other words, the terms *theme, panel, scroll, preview, wx, ptheme* were added to the query. The reformulated query retrieved the same target method (i.e., `COptionsPageThemes.LoadPage`) on position 6 of the ranked list. When analyzing the content of this method we observed that all the terms added by the *Rocchio* expansion were present in the body of the method: *wx* (25 occurrences), *theme* (24), *panel* (12), *ptheme* (9), *scroll* (6), and *preview* (2); which explains the improvement.

Further analysis of the queries that preserved their performance after reformulation revealed that, for all of them, *Refocus* recommended query reduction. One observation is that, when applying this technique the query is not always modified (only if it contains “non-discriminatory” terms). We also noticed that 20 of the queries achieving stable performances (22%) were not improvable, that is, they already retrieved the first relevant method on the first position. The fact that *Refoqus* does not decrease the performances of these queries is certainly a good result. Another 22 original queries (25%) retrieved the relevant method in the top ten positions of the ranked list.

There were 47 (17%) cases when the performances of the reformulated queries using *Refoqus* decreased. The decrease was, on average, of 100 positions in the ranked list, which is, less than half of the average improvement obtained by Refoqus on the improved queries. In other words, the potential negative effect of the reformulations are outweighed by the significant improvements. It is also worth noting that we did not observe significant differences between the percentage of manually formulated queries that were improved by Refoqus (51%) and automatically extracted queries that were improved (52%). We also did not observe significant differences between the C++ and the Java systems, which indicates that *Refoqus* is robust with respect to this aspect (clearly more analysis is needed as this issue is not the main focus in this paper).

C. Research Question 3

Table V compares *Refoqus* and the four baseline reformulation techniques. The obvious observations are: the number of queries improved by *Refoqus* is matched by *RSV Expansion* (i.e., 146), the mean improvement is slightly better for the *Dice Expansion* (i.e., 266 vs. 262), and the number of queries with reduced performance after reformulation is better for *Query Reduction* (i.e., 13 vs. 47).

We can see that the number of queries with preserved results when applying the *Query Reduction* is very large (86%). As explained before, these can be explained by the fact that this technique is rather conservative and it only eliminates words from the query in few cases, keeping the query unchanged in many cases. In addition, most individual expansion strategies result in higher decrease in performance in the case of the negative results than *Refoqus*. We conclude that there is a higher risk to use them over *Refoqus*. In further support of that conclusion, Table VI reports the results of the Mann-Whitney Test performed between the results of *Refoqus* and each baseline, respectively. The tests indicate that *Refoqus* achieves statistically significant better results compared to each

TABLE V
COMPARISON BETWEEN *Refoqus* AND THE BASELINE REFORMULATION TECHNIQUES

Technique	#Queries	#Improved	Improvement						#Worsened	Worsening						#Preserved
			Mean	Q1	Q2	Q3	Min.	Max.		Mean	Q1	Q2	Q3	Min.	Max.	
Query Reduction	282	47	78	4	15	33	1	530	13	15	2	4	20	1	59	242
Rocchio Expansion	282	124	166	3	14	148	1	5,286	130	100	6	28	127	1	1,280	28
RSV Expansion	282	146	233	4	21	178	1	4,843	114	148	5	29	103	1	4,529	22
Dice Expansion	282	127	266	4	32	237	1	5,197	137	314	5	52	204	1	12,829	18
Refoqus	**282**	**146**	**262**	**4**	**23**	**166**	**1**	**5,286**	**47**	**100**	**5**	**19**	**86**	**1**	**970**	**89**

TABLE VI
MANN-WHITNEY RESULT (P-VALUE) AND MEAN OF THE DIFFERENCES

Test	p-value	mean
Refoqus vs Reduction	$<$**0.0001**	-112
Refoqus vs Rocchio Expansion	$<$**0.0001**	-92
Refoqus vs RSV Expansion	$<$**0.0001**	-58
Refoqus vs Dice Expansion	$<$**0.0001**	-152

baseline. Indeed, the mean of differences is negative, showing that *Refoqus* achieves, on average, lower (and thus better) effectiveness measures for the queries. **RQ$_3$ answer.** *Refoqus* outperforms all the baseline reformulation approaches.

VI. THREATS TO VALIDITY

This section discusses the threats to validity that could affect this study, namely *construct*, *internal*, *conclusion*, and *external* validity threats.

Threats to *construct validity* concern the relationship between theory and observation. We evaluated *Refoqus* using a query performance measure (i.e., effectiveness), which is widely used in concept/feature location studies since it provides a quite good estimation of the effort that a developer needs to spend in a feature location task.

Threats to *internal validity* concern co-factors that could influence our results. In our study we automatically extracted a set of queries from the online bug tracking system of the object systems. Such queries are approximations of actual user queries. However, developers are often faced with unfamiliar systems, in which cases they must rely on outside sources of information (as bug reports) to formulate queries during TR-based concept location. Thus, we believe that the approach used in our experimentation resembles real usage scenarios. Nevertheless, in order to mitigate such a threat we also asked a Ph.D. student to manually formulate queries as well.

This is the first work that makes use of the 28 measures that capture different properties of a query and the four reformulation techniques. We do not know at this stage how would the results be affected if we use other measures or reformulation strategies. The same is true for the number of documents in the result list used to suggest expansion terms and the number of terms included in the query during expansion. We used the values of 5 and 10, respectively, but we do not know at this stage how using different values would impact the results. We also do not know how the results would change if we increased the size of the training data sets.

Threats to *conclusion validity* concern the relationship between treatment and outcome. Where appropriate, we used non-parametric statistical tests (Mann-Whitney) to show statistical significance for the obtained results.

Threats to *external validity* concern generalization of the obtained results. In order to mitigate this threat, we selected five software systems from diverse domains, implemented in two programming languages, i.e., Java and C++. A larger set of queries and more systems would strengthen the results from this perspective. Also, we only used a single TR engine (i.e., Lucene). The results may differ when using other TR engines.

The last threat to external validity is related to the fact that we only evaluated the proposed approach for the task of TR-based concept location. Thus, we cannot (and do not) generalize the results to other SE tasks.

VII. RELATED WORK

In the field of TR, query reformulation has long been established as a way to improve the results returned by an TR engine [13]. Various approaches have been proposed over time, which fall in two main categories: *query reduction* [19], [20] and *query expansion* [11] approaches.

In SE, a few works have also taken advantage of query reformulation strategies in order to improve SE tasks supported by TR. Each of these works fall in one of two categories. The first category includes approaches which rely on the involvement of the user to reformulate the query, while the second includes automatic techniques.

A few studies have investigated the manual reformulation of queries by developers. Query reformulation using ontology fragments has been investigated in the context of concept location by Petrenko et al. [21]. In this work, developers build and update ontology fragments which capture their knowledge of the system and then reformulate queries based on these fragments, leading to improved results. Starke et al. [22] have studied how developers search source code when performing corrective tasks on an unfamiliar system. Their findings indicate that even after several reformulations some developers are unable to locate the information they need. These studies provide motivation for our work as they support the need for automatic techniques for query reformulation.

A semi-automated (i.e., interactive) approach for reformulating the queries, which requires the intervention of a developer, is based on using *user relevance feedback*. In this approach, the developer needs to analyze the list of results returned by the TR engine and provide feedback about the relevance of the top returned documents. The query is then automatically reformulated, usually by including terms from the relevant documents and excluding terms from irrelevant ones (as marked by the user). The goal of this approach is to get the meaning of the query closer to that of relevant documents [13]. Papers that make use of this approach in SE include [2] and [23], where user relevance feedback is used to improve TR-based traceability link recovery between various types of software artifacts, and [3], which uses the approach in the context of concept location in code. The results suggest that user relevance feedback generally benefits SE tasks. However, they also underline that it is not always the solution.

A few papers have investigated automated query reformulations, as we do in this paper. These approaches are usually based on reformulating the query using words that are either similar or related in some way to the query terms. Some of these approaches determine word relations based solely on their usage in source code. For example, Marcus et al. [5] have used Latent Semantic Indexing in order to determine the most similar terms to the query from the source code and include them in the query. Yang et al. [6] use the context in which query words are found in the source code to extract synonyms, antonyms, abbreviations and related words to include them in the reformulated query. Hill et al. [24] also use word context in order to extract possible query expansion terms from the code. Shepherd et al. [25] build a code search tool that expands search queries with alternative words learned from verb-direct object pairs. Other approaches make use of external sources of information in order to determine the related words that should be included in the query [4].

A common feature of these automated techniques is that they utilize the same reformulation strategy, regardless of the query or system used. In contrast, *Refoqus* chooses and recommends the best reformulation strategy for each given query and system.

VIII. Conclusions and Future Work

We introduced in premiere an approach (*Refoqus*) based on machine learning that automatically recommends the best reformulation strategy for a textual query used in TR applications in SE. We evaluated *Refoqus* in the context of TR-based concept location in source code. *Refoqus* outperformed a set of baselines and its recommendations lead to query performance improvement or preservation in 84% of the cases. We also found that training *Refoqus* with data from the project where the new queries are to be issued is better than using cross-project data for training. Relatively small data sets used for training lead to very good results. The results are even more impressive when considering that *Refoqus* does not use any user feedback and does not consider any external knowledge sources. We expect that better recommenders can be built using *Refoqus* and user feedback. We also believe that *Refoqus* can be used equally well for other TR-based applications in SE, using other type of data. Future work will address this issue.

IX. Acknowledgments

This work was supported in part by grants from the US National Science Foundation (CCF-1017263 and CCF-0845706).

References

[1] A. Marcus and G. Antoniol, "On the use of text retrieval techniques in software engineering," in *Proceedings of 34th IEEE/ACM International Conference on Software Engineering, Technical Briefing*, 2012.

[2] J. H. Hayes, A. Dekhtyar, and S. K. Sundaram, "Advancing candidate link generation for requirements tracing: The study of methods." *IEEE Transactions on Software Engineering*, vol. 32, no. 1, pp. 4–19, 2006.

[3] G. Gay, S. Haiduc, A. Marcus, and T. Menzies, "On the use of relevance feedback in ir-based concept location," in *Proceedings of the International Conference on Software Maintenance*, 2009, pp. 351–360.

[4] M. Gibiec, A. Czauderna, and J. Cleland-Huang, "Towards mining replacement queries for hard-to-retrieve traces," in *Proceedings of the International Conference on Automated Software Engineering*, 2010, pp. 245–254.

[5] A. Marcus, A. Sergeyev, V. Rajlich, and J. I. Maletic, "An information retrieval approach to concept location in source code," in *Proceedings of the Working Conference on Reverse Engineering*, 2004, pp. 214–223.

[6] J. Yang and L. Tan, "Inferring semantically related words from software context," in *Proceedings of 9th Working Conference on Mining Software Repositories*, 2012, pp. 161–170.

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

[8] S. Haiduc, G. Bavota, A. De Lucia, A. Marcus, and R. Oliveto, "Evaluating the specificity of text retrieval queries to support software engineering tasks," in *Proceedings of the 34th IEEE/ACM International Conference on Software Engineering, NIER Track*, 2012, pp. 1273–1276.

[9] X. A. Lu and R. B. Keefer, "Query expansion/reduction and its impact on retrieval effectiveness," *NIST SPecial Publication SP*, vol. 225, pp. 231–239, 1995.

[10] G. W. Furnas, T. K. Landauer, L. M. Gomez, and S. T. Dumais, "The vocabulary problem in human-system communication," *Communications of the ACM*, vol. 30, no. 11, pp. 964–971, 1987.

[11] C. Carpineto and G. Romano, "A survey of automatic query expansion in information retrieval," *ACM Computing Surveys*, vol. 44, pp. 1–56, 2012.

[12] G. Sridhara, E. Hill, L. L. Pollock, and K. Vijay-Shanker, "Identifying word relations in software: A comparative study of semantic similarity tools," in *Proceedings of the International Conference on Program Comprehension*, 2008, pp. 123–132.

[13] J. J. Rocchio, *The SMART Retrieval System – Experiments in Automatic Document Processing*. Prentice Hall, Inc., 1971, ch. Relevance feedback in information retrieval, pp. 313–323.

[14] S. Haiduc, G. Bavota, R. Oliveto, A. D. Lucia, and A. Marcus, "Automatic query performance assessment during the retrieval of software artifacts," in *IEEE/ACM International Conference on Automated Software Engineering, ASE'12*, 2012, pp. 90–99.

[15] L. Breiman, J. Friedman, C. Stone, , and R. A. Olshen, *Classification and Regression Trees*. Chapman and Hall, 1984.

[16] 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.

[17] W. J. Conover, *Practical Nonparametric Statistics*, 3rd ed. Wiley, 1998.

[18] S. Holm, "A simple sequentially rejective Bonferroni test procedure," *Scandinavian Journal on Statistics*, vol. 6, pp. 65–70, 1979.

[19] N. Balasubramanian, G. Kumaran, and V. R. Carvalho, "Exploring reductions for long web queries," in *Proceedings of SIGIR*, 2010, pp. 571–578.

[20] X. Xue, S. Huston, and W. B. Croft, "Improving verbose queries using subset distribution," in *Proceedings of the ACM International Conference on Information and Knowledge Management*, 2010, pp. 1059–1068.

[21] M. Petrenko, V. Rajlich, and R. Vanciu, "Partial domain comprehension in software evolution and maintenance," in *Proceedings of the International Conference on Program Comprehension*, 2008, pp. 13–22.

[22] J. Starke, C. Luce, and J. Sillito, "Searching and skimming: An exploratory study," in *Proceedings of the International Conference on Software Maintenance*, 2009, pp. 157–166.

[23] A. De Lucia, R. Oliveto, and P. Sgueglia, "Incremental approach and user feedbacks: a silver bullet for traceability recovery," in *Proceedings of the International Conference on Software Maintenance*, 2006, pp. 299–309.

[24] E. Hill, L. Pollock, and K. Vijay-Shanker, "Automatically capturing source code context of nl-queries for software maintenance and reuse," in *Proceedings of the International Conference on Software Engineering*, 2009.

[25] D. Shepherd, Z. Fry, E. Gibson, L. Pollock, and K. Vijay-Shanker, "Using natural language program analysis to locate and understand action-oriented concerns," in *Proceedings of the International Conference on Aspect Oriented Software Development*, 2007, pp. 212–224.