Automatic Query Performance Assessment during the Retrieval of Software Artifacts

Sonia Haiduc
Wayne State Univ.
Detroit, MI, USA
sonja@wayne.edu

Gabriele Bavota
Univ. of Salerno
Fisciano (SA), Italy
gbavota@unisa.it

Rocco Oliveto
Univ. of Molise
Pesche (IS), Italy
rocco.oliveto@unimol.it

Andrea De Lucia
Univ. of Salerno
Fisciano (SA), Italy
adelucia@unisa.it

Andrian Marcus
Wayne State Univ.
Detroit, MI, USA
amarcus@wayne.edu

ABSTRACT
Text-based search and retrieval is used by developers in the context of many SE tasks, such as, concept location, traceability link retrieval, reuse, impact analysis, etc. Solutions for software text search range from regular expression matching to complex techniques using text retrieval. In all cases, the results of a search depend on the query formulated by the developer. A developer needs to run a query and look at the results before realizing that it needs reformulating. Our aim is to automatically assess the performance of a query before it is executed. We introduce an automatic query performance assessment approach for software artifact retrieval, which uses 21 measures from the field of text retrieval. We evaluate the approach in the context of concept location in source code. The evaluation shows that our approach is able to predict the performance of queries with 79% accuracy, using very little training data.

Categories and Subject Descriptors
D.2.7 [Software Engineering]: Distribution, maintenance, and Enhancement – corrections, enhancement.

General Terms: Measurement, Experimentation.

Keywords: Query performance, text retrieval, concept location

1. INTRODUCTION
Text-based search and retrieval are frequently employed by developers when looking for software artifacts that might be helpful for their task at hand. For more than two decades, Text Retrieval (TR) -based search is being successfully applied to a multitude of software engineering (SE) tasks, including: concept location [23], impact analysis [13], code reuse [24], traceability link recovery [2], bug triage [12, 32], requirements analysis [6], etc.

Approaches using TR usually require formulating a query and return a list of ranked software artifacts. The developer examines the list of artifacts and for each of them decides if it is relevant to the current task or not. The performance of any TR-based search technique used in SE depends strongly on the text query and its relationship with the text contained in the software artifacts. Writing good queries is not easy, especially when searching for source code. One of the causes is the vocabulary mismatch problem [10], i.e., developers often use different language to describe the code and its behavior than they use to implement it. In addition, most TR techniques rely on complex statistics, which are rather opaque to the user, i.e., it is not always clear to the user why a document is ranked high/low with respect to a query.

The performance of a query reflects how well the query retrieves the desired documents when executed by a TR approach. A high-performing query retrieves the relevant documents on top of the results list. Conversely, a low-performing query either retrieves the desired documents in the bottom part of the list of results, or it does not retrieve them at all. When low-performing queries are executed, the developer will spend time and effort analyzing irrelevant search results, before she decides to reformulate the query.

Our goal is to overcome this problem by estimating the performance of a query before it is executed. We want to identify the queries that are likely to perform poorly, immediately after they are written and notify the developer that a reformulation of the query is likely needed.

In order to solve this problem, we get our inspiration from the field of natural language (NL) document retrieval. In that field, query performance prediction [5] has been actively researched over the past decade. We found that, while similar, the query performance problems for NL documents and software documents exhibit some essential and important differences. In SE, the relevance of a result to the query is much more sensitive to the context (i.e., the SE task at hand) than in NL document retrieval. For example, during concept location in source code [23] it is more important to improve the rank of one relevant artifact (i.e., find something relevant quickly) rather than improving the rank of all the relevant artifacts (as it is common in NL document retrieval). At the same time, solutions in NL are usually based on measuring different properties of the query and of the NL documents, which are not always applicable to SE artifacts (e.g., the text extracted from the source code is not always correct English). Thus, a careful selection of measures that can be applied to software artifacts is needed. Moreover, many solutions in NL require the execution of the query in order to make an assessment of its performance (i.e., post-retrieval techniques). Since our goal is to offer quick feedback to developers about their queries, we focus only on assessments performed before retrieval (i.e., pre-retrieval techniques). Most pre-retrieval techniques rely on measuring some properties of the query (e.g., coherence) and their relationship with the corpus (e.g., the similarity between the query and the entire document collection).

We propose a solution to the problem of query performance assessment in SE by adapting NL-inspired solutions and using them on software data. Our approach uses 21 selected measures, which assess four different aspects that can influence the performance of a query: specificity, similarity, coherency, and term relatedness. Our technique uses classification trees to learn rules (constructed using some of the 21 measures), which classify queries as having high or low performance.
We evaluated the proposed approach in the context of TR-based concept location in source code. The results on five software systems indicate that our approach is able to correctly assess the performance of queries for the task of concept location with an accuracy of 79% (i.e., it classifies correctly 79% of the queries, in average). The technique performs better than expected, as the results are better than those achieved by state of the art approaches in NL document retrieval [15, 30].

The paper is a premiere in SE, as no prior work addressed query performance prediction on software corpora. Considering the good performance of our technique, we anticipate that it can be used to help developers reformulate their queries faster, hence saving effort and time while solving their SE tasks.

2. RELATED WORK
The need for an in-depth analysis of query performance surfaced in the Text REtrieval Conference (TREC) community, as the high variance in performance across different queries became evident for the TR approaches participating in the TREC competitions. In consequence, a special conference track was created between 2004-2005 (i.e., the Robust track), where a new challenge was introduced that required predicting the performance of the participating TR approaches on each of the queries in the competition [31]. The query performance assessments were done based on different measures that captured various properties of the queries, document collections, and list of search results. The prediction power of each measure was determined by correlating its values with the average precision (AP) values achieved by the queries after execution. A high correlation would indicate that the measure is able to assess the performance of a query, in terms of AP. The correlations obtained were, however, very low and even negative in some cases. This outcome of the Robust track indicated that assessing the performance of queries is a challenging problem, and spouted the research on this topic. Since then, numerous approaches for assessing the performance of a query have been proposed in the NL document retrieval field [5], but the main goal has remained the same: predicting the AP of a query based on measures that correlate with it.

A few papers in the field of NL document retrieval have investigated the query performance assessment problem from the perspective of classifying incoming queries into easy to answer (high-performing) and hard to answer (low-performing) queries [15, 30, 34]. In these works, several classification approaches have been used for this purpose, and in each case, decision trees were found to be the most adequate for solving this problem. While we take inspiration from this work in using classifiers, and in particular decision trees for predicting if a query exhibits low- or high-performance, our work is significantly different in terms of the measures used to train the classifiers and the timing of the prediction (i.e., pre-retrieval vs. post-retrieval).

In terms of measures, the approaches in NL document retrieval usually use a combination of pre-retrieval measures (based on the length and some linguistic properties of the query) and post-retrieval measures, collected after the query is executed. While we use pre-retrieval measures in our approach, we chose a different set of measures, for two reasons. First, the measures based on query length were proved to have little to no bearing on the query performance [18]. Second, the rest of the pre-retrieval measures used were all based on linguistic properties of the query, and made use of word relationships as defined in a NL context.

Software artifacts (e.g., source code) do not always follow the rules of discourse found in NL documents [28]. Thus, the linguistic measures are not generally applicable to software and, in consequence, we do not use them in our approach.

Within SE, predicting the performance of queries has not been tackled. The (somewhat) related work in SE deals with the manual or automatic query reformulation and refinement [7, 8, 11, 14, 17, 26, 27], mostly based on relevance feedback mechanisms. Also, some studies have investigated the results of formulating different queries for the same information need [21, 29], which highlighted the strong dependence of the retrieval performance on the query and motivates our work.

3. PROPOSED APPROACH
We propose an approach for automatically assessing the performance of queries before they are executed, in the context of SE tasks. While this bares clear resemblances to the analogous problem in NL, there are some aspects that make SE tasks unique. As mentioned before, in many SE tasks, such as, concept location, it is more important that one relevant artifact is found as soon as possible rather than retrieving all relevant ones. Even when the retrieval of all relevant software artifacts is needed, software data offers additional means for retrieving the complete set of the relevant artifacts once the first one is identified (e.g., dependencies in the code). For example, in source code, a call graph can be used to locate the other relevant artifacts, starting from the first one identified [25]. Query performance assessment for certain SE tasks would, therefore, benefit more from determining if the query leads to quickly identifying the first relevant document than from predicting the AP (i.e., average precision). However, there are SE tasks, such as, traceability link recovery between software artifacts that may gain more by reducing the AP of the retrieval.

We focus on the former category of tasks in this work, which require rethinking and adapting existing work from NL document retrieval. Our goal is to offer a clear and pragmatic answer to developers, indicating if a query is worth pursuing or requires reformulation. The proposed approach offers such an answer by classifying queries in two categories: i.e., high-performing or low-performing, where the latter are queries that require reformulation. The term query performance refers here to the ability of the query to retrieve the relevant software artifacts to the task at hand in such a way that they are easily accessible by developers (i.e., they are placed close to the top of the result list). A query that achieves this is considered a high-performing query, as opposed to low-performing queries, which either fail to retrieve the relevant documents altogether or they place them at high ranks in the list of results, making them hard to reach by developers. Note that the definition of high-performing and low-performing queries may need to be reformulated, based on the current SE task. For example, in some applications, a high-performing query may be considered one that retrieves the highest ranking relevant artifact within the top 20, whereas in other cases this threshold could be set to 50.

Our proposed approach uses classification trees [4] in order to assign queries to one of the two categories. Decision tree learning has been previously applied successfully to query performance prediction in NL [15, 30, 34], and also to analyzing SE data [20] (i.e., for defect prediction). In order to train the classification trees for assessing query performance, we make use of a set of carefully selected, state-of-the-art pre-retrieval performance prediction measures [5] from the field of TR for NL documents. We selected 21 such measures, which refer to four aspects that

---

1 http://trec.nist.gov/
can influence the performance of a query: specificity, similarity, coherency, and term relatedness. The measures were selected such that they can be applied to any type of software artifacts. For example, measures making use of word relationships based on WordNet\(^2\) were not selected, due to the fact that the lexical relationships in software data have a different nature than the ones in English [28].

The rest of this section is organized as follows. Subsection 3.1 presents an overview of the process followed in our approach, followed by subsection 3.2, which presents the 21 measures used, and subsection 3.3 which contains details about the classification approach.

### 3.1 Process

Our approach is applicable to any SE task that makes use of TR-based search, and it consists of several steps, described below.

#### 3.1.1 Collect Training Data

The first step in our approach deals with collecting the data needed for training the classification trees. This data consists of sets made of tuples, each containing: the query, the 21 pre-retrieval measures for the query, and the class of the query (i.e., high-performing or low-performing). The queries may be collected from different sources, depending on the SE task. For example, in the case of traceability link recovery the queries may be requirements documents or fragments of requirements, source code, design documents, etc. In the case of concept location, the queries are either formulated by developers or are automatically extracted from bug and feature repositories. Once the queries are collected, the 21 performance measures are computed based on the text of the query and on the software artifacts in the collection. Last, the queries are executed using a TR engine and based on the results their class is determined (i.e., high-performance or low-performance).

#### 3.1.2 Build the Classification Tree

At this point, the classification tree can be trained using the data collected. After this initial training, the classifier rules are built and they can be used to assess the performance of new queries. There are two different approaches that can be used for training the classifier. The first is based on training the classifier independently for each new software system, thus using only the training data from one system at the time. While this approach might be able to assess the performance of queries better for a particular system, as it adapts to its specific features, it may not be applicable to new systems. The second approach is based on training the classifier based on data from a set of systems, with the purpose of creating a general model that can be applied to a variety of software systems. We evaluate both approaches for the task of concept location in our study, presented in Section 4. More details about the classifier used by our approach can be found in Section 3.3.

#### 3.1.3 Assess the Performance of New Queries

Once the classifier is built, it can be applied for assessing the performance of new queries, based only on computing the subset of the 21 measures of query performance which are included in the rules of the classification tree. Thus, the classifier may be able to determine if a query needs reformulation immediately after it was formulated by the developer.

### 3.2 Query Performance Aspects and Measures

This section presents the 21 pre-retrieval query performance assessment measures used by our approach. More details, including the formulas used to compute each of the 21 measures, can be found in [5] and [16] and are summarized in our online appendix\(^3\).

#### 3.2.1 Specificity

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, 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.

Specificity measures are usually based on the query terms’ distribution over the collection of documents. For our approach, we considered eight specificity measures from the text retrieval literature [5], namely: average inverse document frequency (AvgIDF); maximum inverse document frequency (MaxIDF); standard deviation of the inverse document frequency (DevIDF); average inverse collection term frequency (AvgICTF); maximum inverse collection term frequency (MaxICTF); standard deviation of inverse collection term frequency (DevICTF); query scope (QS); and simplified clarity score (SCS).

The first six measures are based on using the inverse document frequency metric (IDF), which is the inverse of the number of documents in the collection in which a term appears, and the inverse collection term frequency (ICTF), which is the inverse of the number of occurrences of a term in the entire document collection. The assumption is that the more documents a term appears in and the highest its frequency over the entire collection is, the more difficult it is to discriminate the relevant documents based on it. Thus, query terms should have high IDF and ICTF values and a high-performance query should have a high AvgIDF, and AvgICTF, which are the average IDF and ICTF among the query terms. MaxIDF and MaxICTF, which represent the maximum IDF and ICTF values across all query terms, respectively, are popular variations of the average, and are also expected to assume high values in the case of high-performance queries.

DevIDF and DevICTF are the standard deviations of the IDF and ICTF values over the query terms and assume that low variance reflects the lack of dominant, discriminative terms in the query, which may prevent the retrieval of relevant documents. In consequence, DevIDF and DevICTF are expected to have high values for high-performance, discriminative queries.

QS (query scope) measures the percentage of documents in the collection containing at least one of the query terms. A high QS value indicates that there are many candidates for retrieval thus separating relevant documents from irrelevant ones might be difficult. A query should, therefore, aim at having a low QS.

The last specificity measure we considered is SCS, which measures the divergence of the query language model from the collection language model, as an indicator of query specificity. The measure considers that a query is not specific if the language

\(^2\) [http://wordnet.princeton.edu/](http://wordnet.princeton.edu/)

\(^3\) [www.cs.wayne.edu/~severe/ASE12](www.cs.wayne.edu/~severe/ASE12)
used in it is similar to the language used in the entire collection of
documents, which indicates a large number of documents that
could potentially be retrieved. A high SCS, indicating a
significant divergence of the two language models, is thus
desirable.

In addition to the metrics existent in the field of TR, we
considered four new metrics based on using information entropy
in order to identify discriminative, high-performance queries. In a
preliminary study [16], we have shown that entropy is a better
indicator of query specificity for SE tasks than the leading
specificity measures from text retrieval. Therefore, we defined
four query specificity measures using entropy: \( \text{AvgEntropy} \), which
is the average entropy value among the query terms, \( \text{MedEntropy} \)
and \( \text{MaxEntropy} \), which represent the median and the maximum
entropy values across the terms in the query, and \( \text{DevEntropy} \),
which is the standard deviation of the entropy across all query
terms. As low entropy indicates high information content, the
desirable values for a high-performance query are low for the first
three entropy-based measures. For \( \text{DevEntropy} \), high values are
wanted.

### 3.2.2 Similarity

The similarity between the query and the entire document
collection is considered as being another indicator of query
performance. The argument behind this type of measures 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.

The existing similarity approaches for query performance in the
field of text retrieval make use of a metric called **collection query
similarity** (SCQ). This is computed for each query term, and is a
combination of the collection frequency of a term (CTF) and its
IDF in the corpus. Three measures of a query’s performance were
defined based on it, namely **SumSCQ**, which is the sum of the
SCQ values over all query terms, **AvgSCQ**, which is the average
SCQ across all query terms, and **MaxSCQ**, which represents the
maximum of the query term SCQ values. In the case of every
SCQ-based measure, a high value is expected for high-performance
queries.

### 3.2.3 Coherency

Another performance indicator 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. The coherence score (CS) of a term is one
of the measures used for this performance aspect and it reflects
the average pairwise similarity between all pairs of documents in
the collection that contain that particular term. The CS of the
query is then computed as the average CS over all its query terms,
and it is expected to be high in the case of high-performance
queries.

A second approach for measuring the query coherency is based on
measuring the variance (VAR) of the query term weights over the
documents containing the terms in the collection. The weight of a
term in a document indicates the importance, or relevance of the
term for that document and it can be computed in various ways.
One of the most frequent ways to compute it, which we also adopt
in our implementation, is **TF-IDF**, i.e., a combination between the
frequency of a term in the document (TF) and the term’s IDF
value over the document collection. The intuition behind
measuring the variance of the query term weights is that if the

### 3.2.4 Term Relatedness

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. We use two measures of term relatedness previously
used in text retrieval, both using the pointwise mutual information
(PMI) metric, which is based on the probability of two terms
appearing together in the corpus. The two PMI-based metrics are
\( \text{AvgPMI} \) and \( \text{MaxPMI} \), which compute the average and the
maximum PMI values across all query terms.

### 3.3 The Classifier

As mentioned before, we use a classification tree [4] in order to
determine rules that can predict if queries are high- or low-
performing. A classification tree is a prediction model that can be
represented as a decision tree [4]. Such a prediction model is
suitable to solve classification-type problems, where the goal is to
predict values of a categorical variable from one or more
continuous and/or categorical predictor variables. In our work,
the categorical dependent variable is represented by the query
performance (high or low), while the 21 query performance
measures represent the predictor variables.

Training data, with pre-assigned values for the dependent
variables are used to build the classification tree. This set of data
is used by the classification tree to automatically select the
predictor variables and their interactions that are most important
in determining the outcome variable to be explained. The
constructed classification tree is 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 based on two pre-retrieval measures can be
found in Figure 1.

Classification trees have some additional benefits. First, the
classification rules they produce are easy to understand by
humans, which is not true for other, more complex models.
Second, they offer automatic feature selection. This is a very
important property, as it allows using as input a large set of
measures that might capture a phenomenon, without having to
determine beforehand the ones that represent the phenomenon
the best. This allows us to use all 21 performance measures as input,
as our classification tree will determine automatically the
measures needed for the classification. One last advantage of
using classification trees is that the classification is performed
very fast, compared to using other, slower techniques.

In our study, presented in Section 4, we use the CART
(Classification and Regression Tree) implementation of
classification trees provided in the statistical platform R^4.

---

4 www.r-project.org/
4. EVALUATION

We evaluated our approach for concept (or feature) location in source code, as many existing concept location techniques use TR-based solutions [9, 23]. In the context of software change, concept location is concerned with identifying a point of change (e.g., a class or a method), given a change request.

4.1 Query Performance Assessment for Concept Location in Source Code

In order to collect the queries needed for the case study, we used an approach frequently adopted in concept location empirical studies. The approach is based on change reenactment [19] and user simulation, i.e., automatically extracting queries and the changed code from bug reports found in online bug tracking systems. We collected queries for five open source object-oriented (OO) systems from different problem domains, implemented in Java and C++, which are summarized in Table 1. Adempiere\(^5\) is a common-based peer-production of open source enterprise resource planning applications. A\(\text{Tunes}^6\) is a full-featured media player and manager. FileZilla\(^7\) is a graphical FTP, FTPS, and SFTP client. J\(\text{Edit}^8\) is a programming editor, and Win\(\text{Merge}^9\) is a document differencing and merging tool.

<table>
<thead>
<tr>
<th>System</th>
<th>Version</th>
<th>Language</th>
<th>KLOC</th>
<th>#Methods</th>
<th>#Queries</th>
</tr>
</thead>
<tbody>
<tr>
<td>Adempiere</td>
<td>3.1.0</td>
<td>Java</td>
<td>330</td>
<td>28,355</td>
<td>32</td>
</tr>
<tr>
<td>A(\text{Tunes})</td>
<td>1.10.0</td>
<td>Java</td>
<td>80</td>
<td>3,481</td>
<td>32</td>
</tr>
<tr>
<td>FileZilla</td>
<td>3.0.0</td>
<td>C++</td>
<td>240</td>
<td>3,240</td>
<td>36</td>
</tr>
<tr>
<td>J(\text{Edit})</td>
<td>4.2</td>
<td>Java</td>
<td>250</td>
<td>5,532</td>
<td>28</td>
</tr>
<tr>
<td>Win(\text{Merge})</td>
<td>2.12.2</td>
<td>C++</td>
<td>410</td>
<td>8,012</td>
<td>36</td>
</tr>
<tr>
<td>Total</td>
<td>-</td>
<td>-</td>
<td>1,310</td>
<td>48,620</td>
<td>164</td>
</tr>
</tbody>
</table>

For each system, we built the source code corpus used by the TR search by considering every 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). The corpus was indexed by Lucene\(^{10}\), a popular implementation of the Vector Space Model.

We then identified for each system a set bug reports that correspond to bugs that are present in the version of the software system used in our study, but fixed in a later version. We also determined the set of methods that were modified in order to fix each bug, based on the patches attached to the bug reports in the online issue tracking systems. This set of methods represents the oracle for concept location. We will refer to these methods as the target methods.

For each change request, we created two queries, extracted from the online issue tracking systems. The first query was obtained from the title of the bug report (i.e., the short description), while the second query was represented by the description of the bug (i.e., the long description). As usually done for concept location, any trace information or log files contained in these descriptions were eliminated prior to the extraction. Also, the normalization techniques applied on the corpus were applied on the extracted queries as well. Table 1 reports the number of queries we selected for each system. For example, from Bug #1605980 of Adempiere, we obtained the following two queries after extraction and normalization (in parenthesis is the original text extracted from the bug reports, before the normalization):

1. print invoic process draft select (Print Invoices process - draft & selection)
2. us garden world select date rang in todai all invoic select regardles document statu client bad print post custom us email option draft potenti cancel invoic sent (Using Garden World, if you select a date range from somewhere in 2001 to today then ALL invoices are selected regardless of document status OR client!!! Not so bad if you are printing them and posting them to customers but if you use the email option then drafted (and potentially cancelled) invoices are sent too!)

While fixing this bug, the target method changed by the developers was doIt(), found in the \texttt{process} package, InvoicePrint.java file, and InvoicePrint class. The document corresponding to this method is the one that the queries are supposed to retrieve.

During concept location, it is important that developers find their target method (i.e., the method where they have to start the change) as fast as possible. Other methods that will change are identified during impact analysis. When reenacting concept location, the success criterion is translated into the rankings of the target methods (as opposed to many other TR applications where recall and precision are considered). In other words, if any of the target methods ranks in among the top retrieved results, we consider it a successful retrieval. A rule used in concept location applications is that finding a target method among the top 20 ranked results is considered a good result, based on the assumption that most developers would look at no more than 20 methods before reformulating their query. Hence we define a query as \textit{high performing} if any of the target methods is retrieved in the top 20 results. Otherwise, we consider the query as \textit{low performing}. In the above example, if a query returns the target method doIt() in top 20, then it is considered high performing. The rank (i.e., the position) of the target method in the ranked list of results retrieved by the two queries in the previous example, as well as the classification of the two queries and the values of the 21 measures of query performance are presented in Table 2. For

\(^{10}\) http://lucene.apache.org

---

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
all measures besides AvgEntropy, MaxEntropy, MedEntropy, and QS, a higher value indicates a better query.

**Table 2. The 21 pre-retrieval measures of the short and long description queries for Bug #1605980 in Adempiere**

<table>
<thead>
<tr>
<th>Measure</th>
<th>Short</th>
<th>Long</th>
<th>Measure</th>
<th>Short</th>
<th>Long</th>
</tr>
</thead>
<tbody>
<tr>
<td>AvgIDF</td>
<td>3.69</td>
<td>4.83</td>
<td>SCS</td>
<td>1.93</td>
<td>0.75</td>
</tr>
<tr>
<td>MaxIDF</td>
<td>6.40</td>
<td>10.25</td>
<td>AvgVAR</td>
<td>0.04</td>
<td>0.04</td>
</tr>
<tr>
<td>DevIDF</td>
<td>0.62</td>
<td>0.83</td>
<td>MaxVAR</td>
<td>0.11</td>
<td>0.12</td>
</tr>
<tr>
<td>AvgICTF</td>
<td>2.72</td>
<td>4.04</td>
<td>SumVAR</td>
<td>0.17</td>
<td>1.18</td>
</tr>
<tr>
<td>MaxICTF</td>
<td>6.08</td>
<td>10.25</td>
<td>CS</td>
<td>0.13</td>
<td>0.39</td>
</tr>
<tr>
<td>DevICTF</td>
<td>0.77</td>
<td>0.86</td>
<td>AvgSCQ</td>
<td>28.45</td>
<td>28.78</td>
</tr>
<tr>
<td>AvgEntropy</td>
<td>0.60</td>
<td>0.53</td>
<td>MaxSCQ</td>
<td>33.03</td>
<td>37.53</td>
</tr>
<tr>
<td>MedEntropy</td>
<td>0.67</td>
<td>0.60</td>
<td>SumSCQ</td>
<td>113.81</td>
<td>777.03</td>
</tr>
<tr>
<td>MaxEntropy</td>
<td>0.70</td>
<td>1.00</td>
<td>AvgPMI</td>
<td>0.06</td>
<td>0.13</td>
</tr>
<tr>
<td>DevEntropy</td>
<td>0.06</td>
<td>0.08</td>
<td>MaxPMI</td>
<td>1.13</td>
<td>5.02</td>
</tr>
<tr>
<td>QS</td>
<td>0.16</td>
<td>0.40</td>
<td>Rank (class)</td>
<td>125 (low)</td>
<td>1 (high)</td>
</tr>
</tbody>
</table>

We classify in this way all the 164 queries used in our evaluation. Table 3 reports the number of high and low performing queries for each system.

**Table 3. High and low performing queries**

<table>
<thead>
<tr>
<th>System</th>
<th>High-performing queries</th>
<th>Low-performing queries</th>
</tr>
</thead>
<tbody>
<tr>
<td>Adempiere</td>
<td>15</td>
<td>17</td>
</tr>
<tr>
<td>aTunes</td>
<td>14</td>
<td>18</td>
</tr>
<tr>
<td>FileZilla</td>
<td>8</td>
<td>28</td>
</tr>
<tr>
<td>JEdit</td>
<td>13</td>
<td>15</td>
</tr>
<tr>
<td>WinMerge</td>
<td>18</td>
<td>18</td>
</tr>
<tr>
<td>Total</td>
<td>68</td>
<td>96</td>
</tr>
</tbody>
</table>

4.1.1 Validation Method

In order to evaluate the ability of the proposed methodology in predicting the query performance, we performed two types of validation: single-system and cross-system validation. For the single-system validation, the classification model was trained on each system individually and a 4-fold cross-validation was performed. Building the classification tree took less than 1 second per system, using a MacBook Pro with a 2Ghz dual-core Intel Core 2 Duo processor and 4GB RAM. The process for the single-system validation is composed of five steps: (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) classify the queries in the test set using the classification model built on the query training set and store the accuracy of the classification, (iv) repeat this process, setting aside each query subset in turn, (v) compute the overall average accuracy of the model. The misclassification rate of the model has been evaluated in terms of Type I and Type II classification errors. A Type I error occurs when the model wrongly classifies a high performing query as low performing, while a Type II error is when the model wrongly classifies a low performing query as high performing.

For the cross-system validation, we performed the same 4-fold cross-validation process, this time considering all the 164 queries from the different object systems as a single dataset. When dividing the dataset into 4 approximately equal subsets, we ensured that in both training and test sets there was the same percentage of queries belonging to the different object systems.

The two types of validation, i.e., single-system and cross-system, were both needed in order to derive guidelines on how to use historical data to build the classifier. In particular, we aimed at analyzing whether a specialized model is required for each system or it is possible to define a generic model that could be applied on several systems.

4.1.2 Baselines

In the context of our study we compared our approach based on classification trees with four baseline approaches: logistic regression, a random classifier, and two variants of a constant classifier (pessimistic and optimistic). The random classifier randomly selects a prediction from the possible values, i.e., high or low. The two values have the same probability to be selected. The constant classifier always predicts a specific value disregarding the instance. In particular, the pessimistic constant classifier always classifies a query as low, while the optimistic constant classifier works in the opposite way, always classifying a query as high performing. It is worth noting that a classifier is useful only if it outperforms a random or constant classifier.

Logistic regression is used for the prediction of the occurrence probability of an event by fitting data to a logistic function [1]. It is one of the most commonly used classification techniques, and it has been applied to software engineering problems as well as other experimental fields. For this reason we decided to use it as an additional baseline for comparison in our study. Given the novelty of our work, there is no prior state-of-the-art technique to compare our results with.

Formally, the multivariate logistic regression model is based on the formula:

$$\pi(X_1, X_2, \ldots, X_n) = \frac{e^{c_0+c_1X_1+c_2X_2+\cdots+c_nX_n}}{1 + e^{c_0+c_1X_1+c_2X_2+\cdots+c_nX_n}}$$

where $$X_i$$ are the independent variables (i.e., the 21 pre-retrieval measures) and $$0 \leq \pi \leq 1$$ is a value on the logistic regression curve. In a logistic regression model, the dependent variable $$\pi$$ is commonly a dichotomous variable, and thus, assumes only two values, i.e., it states whether a query is high (1) or low (0) performing. In our study we used the WEKA tool for the definition of a logistic model. Before applying logistic regression to a dataset, it is a common approach to perform feature selection in order to determine which features should be considered when building the logistic model. We performed feature selection among the 21 pre-retrieval measures using the gain ratio technique implemented in WEKA.

4.2 Quantitative Results

Figure 2 and Table 4 report the results achieved in the single-system evaluation. In particular, Table 4 shows the number of Type I and Type II misclassifications performed by the experimented classifiers. The total number of errors performed by the classification tree (CART) is 34 (11 Type I + 23 Type II), compared to 82 for the logistic regression (49 Type I + 33 Type II), 96 for the optimistic constant model (all of Type II), 68 for the pessimistic constant model (all of Type I) and 72 for the random predictor (25 of Type I + 47 of Type II). This indicates that the model built using the classification tree significantly outperforms all the baseline classifiers, by correctly classifying 79% of the examined queries, i.e., 130 out of 164 (see Figure 2). In comparison, the model built using the logistic regression correctly classified only 82 queries (50%), the optimistic and pessimistic

11 http://www.cs.waikato.ac.nz/ml/weka/
constant predictors classified correctly 68 (41%) and 96 (59%), respectively, and the random classifier was able to correctly assess 92 queries (56%).

In addition to the significantly better results obtained by the classification tree-based predictor over the baseline approaches, our results are better than even state-of-the-art results from the NL document retrieval field. By comparison, the best approaches in NL document retrieval correctly classify, on average, between 62% [15] and 74% [30] of the queries.

The classification tree built on the Adempiere software system is reported in Figure 3. Based on the rule of this classification tree, we can see that the short description query for Bug #1605980 in Adempiere is correctly classified as a low—performing query as its SumSCQ is 113.81, which is smaller than 160.3 (see Table 2). At the same time, the long-description query of the same bug is also correctly classified, but as a high—performing query, having the SumSCQ equal to 777.03, thus greater than 160.3.

In our single-system evaluation the decision tree always selected one measure to discriminate between high-performing and low-performing queries (in the example reported in Figure 3, the measure SumSCQ was selected). However, the measure used to build the classification tree was often different among the software systems and sometimes even among the different training samples used in the 4-fold validation on the same system.

Table 5 shows the measures selected in each run of the single-system evaluation for each system.

The fact that the measure selected for building the classification tree is generally different among the object systems highlights the fact that different software corpora, having different characteristics (e.g., verbosity, vocabulary dimension, etc.) may require different classifiers to estimate the performance of a query. This is confirmed also by the cross-system evaluation, whose results are presented in Table 6.

The results illustrate that the cross-system classification tree performs poorly, as it correctly classifies only 51% of the queries. Its results are comparable to the results of the baseline techniques, which never achieve a correct classification rate higher than 53%. This indicates that the assessment of query performance is
strongly dependent on the system. In consequence, training needs to be performed on each system independently in order obtain a correct classification of the performance of incoming queries (for the same system).

Table 6. Type I and Type II errors in the cross-system evaluation

<table>
<thead>
<tr>
<th>Method</th>
<th>Type I</th>
<th>Type II</th>
<th>Type I &amp; II</th>
<th>Correct</th>
</tr>
</thead>
<tbody>
<tr>
<td>CART</td>
<td>44(27%)</td>
<td>37(22%)</td>
<td>81(49%)</td>
<td>83(51%)</td>
</tr>
<tr>
<td>Logistic Regression</td>
<td>49(30%)</td>
<td>31(19%)</td>
<td>80(49%)</td>
<td>84(51%)</td>
</tr>
<tr>
<td>Optimistic constant</td>
<td>0(0%)</td>
<td>91(55%)</td>
<td>91(55%)</td>
<td>73(45%)</td>
</tr>
<tr>
<td>Pessimistic constant</td>
<td>77(47%)</td>
<td>0(0%)</td>
<td>77(47%)</td>
<td>87(53%)</td>
</tr>
<tr>
<td>Random</td>
<td>49(30%)</td>
<td>43(26%)</td>
<td>92(56%)</td>
<td>72(44%)</td>
</tr>
</tbody>
</table>

4.3 Qualitative Analysis of Misclassifications
Among the errors performed by our classifier, 32% (11 out of 34) are Type I errors, i.e., a high performing query is classified as a low performing one, while 68% (23 out of 34) are Type II errors, occurring when a low performing query is classified as a high performing one.

We manually analyzed all Type I errors and we were not able to identify any pattern that could help understand why the proposed approach misclassifies the queries. One of the challenges caused by using bug descriptions and change data for such studies is that, in some cases, the bug descriptions capture the observed behavior of the system, whereas the bug is fixed in a part of a code more with different vocabulary characteristics. We believe that if we use more data (both in terms of queries and systems), we may get a richer set of Type I errors, which will reveal some patterns.

No confirm for delete in folder view

Reported by: trellmor
Priority: normal
Component: FileZilla client

Description
If you try to delete a folder by right click -> delete in the remote folder window (e.g. the upper right) it won’t ask for confirmation. If you do the same in the normal windows (e.g lower right) FileZilla asks if you really want to delete the file/folder(s).
This happens in FileZilla 3.0.0 and 3.0.1 on Ubuntu GNU/Linux. I haven’t tested it on other versions/platforms.

Figure 4. Bug report #3272 in FileZilla

On the other hand, we were able to make interesting observations from the analysis of the Type II errors. We analyzed the two different aspects involved in it, i.e., why the queries performed poorly and why our classifier considers these queries as high-performing. In order to explain our findings, we use an example of Type II error from the FileZilla software system. In FileZilla, the bug reports #3272 (Figure 4) and #3344 (Figure 5), both have the onMenuDelete (wxCommandEvent) method from the cRemoteTree class as the target method. Both bug reports describe problems occurring during the deletion of a folder. As described in our experimental design, we extracted four different queries from these two bug reports, i.e., two from their titles and two from their descriptions. The two queries for bug report #3272 retrieve the target method on positions 424 and 1,109, respectively, while those for bug report #3344 recover the target method on positions 684 and 372. Thus, all four queries are low-performing. However, two of these queries (those retrieving the target method on positions 372 and 1,109) are considered high-performing by our classifier, resulting in two Type II errors.

When investigating the origin of these errors, we first examined the reasons for which the two queries were classified as high-performing. We found that the queries contain some highly discriminating terms (i.e., “rmd”, “cwd”), which appear in few documents in the corpus, thus justifying the decision of the classifier to consider them as high-performing.

Directory is not removed from list after deletion

Reported by: larser255
Priority: normal
Component: FileZilla client

Description
I delete a directory from the remote site view and Filezilla tries to change to that directory after deletion: Command: RMD /1
Response: 260 RMD command successful Status: Retrieving directory listing...
Command: CWD /1
Response: 550 /1: No such file or directory
Error: Failed to retrieve directory listing

Figure 5. Bug report #3344 in FileZilla

Since we considered that the classifier performed properly in this case, we further focused on understanding why none of the queries composed for the two bug reports were able to retrieve the target method in the top 20 results. We first investigated whether it is possible to retrieve the target method among the top results by using a query based on the terms in the bug report. We thus investigated the performance of each individual term in the bug reports when run as a query, as well as that of all the possible combinations of terms from the two bug reports. In other words, for each bug report, we first executed all the queries composed of one of the terms present in it, then all the queries composed of pairs of terms and so on. We formulated and ran a total of 8,190 queries for bug report #3272 and 32,767 for bug report #3344. The highest performing query was delete remote, which was able to retrieve the target document on position 93. Therefore, no query based on terms from the bug reports was able to retrieve the target method in the top 20 results.

We then investigated the target method in order to understand what the queries based on bug report terms were missing. The target method contained the following terms (in parenthesis the number of times they appear in the method):

- path(5) menu(4) item(4) context(3) event(2) remote(2) state(2) tree(1) view(1) rename(1) wx(1) command(1) idl(1) server(1) empty(1) delete(1)
The first thing we noticed was that the terms contained in the method were mostly generic terms, which were found in many other methods in the system:


Thus, retrieving this method in the first 20 results would be difficult even for a query containing terms from the method, as there are many other candidates for retrieval, i.e., the other methods containing these terms.

This is also an example where the vocabulary mismatch between the bug description (i.e., system behavior) and the target method implementation is quite strong. We conjecture that in such cases Type II misclassifications are more likely to occur. However, we need more data to confirm this hypothesis, which we plan to do in the future.

In conclusion, in cases like the above example, it is most likely that misclassification would occur due to the fact that the target method is generally hard to retrieve, resulting in a low-performing query. In these situations, other approaches may be needed in order to complement TR during concept location, such as, static or dynamic analysis.

4.4 Threats to Validity

This section discusses the main threats to validity [33] that could affect our results.

Construct validity threats concern the relationship between theory and observation. We evaluated the proposed predictor through two metrics (i.e., Type I and Type II errors) that are widely used to evaluate predictor models [1]. In addition, we analyze and compare the overall classification accuracy of the proposed approach taking into account the number of queries correctly and wrongly classified, and also perform a qualitative analysis of the errors.

Concerning the internal validity, in our experimentation we automatically extracted the set of queries from the online bug tracking system of the object systems. In particular, we extracted two different queries, one derived from the title of the bug report and one from the description of the bug. This choice could affect the results of our study since such queries are approximations of actual user queries. However, developers are often faced with unfamiliar systems, in which cases they may rely on outside sources of information, such as issue reports, in order to formulate queries during TR-based concept location. Therefore, we believe that the approach used in our experimentation resembles real usage scenarios. Nevertheless, empirical studies conducted with users are required to evaluate our predictor in a real usage scenario and we plan to perform such studies in the near future.

The external validity refers to the generalization of our findings. In order to address this threat, we selected a set of 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 clearly strengthen the results from this perspective.

One threat to the external validity of our results is the fact that we used the results of only one TR engine in order to classify the queries as high-performing or low-performing. More precisely, we used the rank of the first target method retrieved by a query submitted to the Lucene TR-engine, which is an implementation of the VSM technique. Since several other TR methods have been previously used to support concept location [3, 22], further experimentation is needed to analyze whether the proposed predictor works well also with other TR methods.

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. We plan to evaluate the proposed predictor in other contexts, such as, traceability link recovery.

Finally, conclusion validity refers to the degree to which conclusions reached about relationships between variables are justified. In our case study, we only draw conclusions referring to the use of different classifiers, which we support with evidence in the form of classification correctness and Type I and II errors.

5. CONCLUSIONS AND FUTURE WORK

In this paper we proposed an approach that estimates the performance of a text query before it is executed, in the context of TR-based concept location. The proposed approach can be used for other SE tasks and it allows the identification of queries that are likely to perform poorly immediately after they are written. This can save the developer time and effort, as she can be notified right away when a query is unlikely to lead to satisfactory results and would likely need reformulation. The proposed approach is based on using classification trees and 21 pre-retrieval query performance measures selected from the field of text retrieval.

Our empirical evaluation showed that the classification trees built using very small training samples, are able to correctly classify 79% of queries in average, strongly outperforming several baseline approaches, as well as state-of-the-art results from the field of TR.

In our future work, we plan to perform a more extensive experimentation by evaluating several different classifiers (e.g., Bayesian, neural networks, random forests, etc.), using more TR techniques to classify the queries as high- and low-performing (e.g., LSI, LDA, etc.), and applying our approach to other TR-based SE tasks (e.g., traceability recovery, code reuse, etc.).

Another direction we plan to pursue is the tool-supported reformulation of low-performing queries. In particular, once a low-performing query is identified, we plan to provide support to the developer to reformulate the query, suggesting terms that can improve its performance. Such an approach can help developers find helpful software artifacts faster and finish their tasks sooner.

6. ACKNOWLEDGEMENTS

This work was supported in part through grants from the National Science Foundation: CCF-0845706 and CCF-1017263.

7. REFERENCES
