Automatic Query Performance Assessment during the Retrieval of Software Artifacts

Sonia Haiduc
Wayne State Univ.
Detroit, MI, USA

Gabriele Bavota
Univ. of Salerno
Fisciano (SA), Italy

Rocco Oliveto
Univ. of Molise
Pesche (IS), Italy

Andrea De Lucia
Univ. of Salerno
Fisciano (SA), Italy

Andrian Marcus
Wayne State Univ.
Detroit, MI, USA

sonja@wayne.edu, gbavota@unisa.it, rocco.oliveto@unimol.it, adelucia@unisa.it, amarcus@wayne.edu

ABSTRACT

Text-based search is done by developers in the context of many software engineering 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 the 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 this 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 have essential differences. One important difference is the fact that for many SE tasks (e.g., concept location) 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 predictions 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 use them on software data. Our approach uses 21 selected measures, which

ASE'12, September 3–7, 2012, Essen, Germany.

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 the 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[1] (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 predictions of the query performance 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 predict 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 predicting the performance of queries is a challenging problem, and sprouted the research on this topic. Since then, numerous approaches for predicting 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 prediction 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 this problem. While we take inspiration from this work in using classifiers, and in particular decision trees for predicting if a query exhibits low- of 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 prediction 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 or no bearing on the query performance [18]. Second, the other pre-retrieval measures used were all based on linguistic properties of the query, based on word relationships in NL. 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 predicting the performance of queries in the context of SE tasks bares clear resemblances to the NL task, there are some aspects that make SE tasks unique. First of all, 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 identify the other relevant artifacts, starting from the first one identified [25]. An approach for query performance assessment that is applicable to certain SE tasks would, therefore, benefit more from focusing on predicting if the query leads to identifying the first relevant document in a reasonable amount of time rather than predicting the AP (i.e., average precision). On the other hand, other SE tasks addressed using TR, such as, traceability link recovery between software artifacts, may be more interested in reducing the average precision of the retrieval.

We chose to focus on the former category of tasks, as the solution requires rethink exiting work from NL document retrieval. At the same time, an approach that is to be used in the context of solving SE tasks needs to offer a clear and pragmatic answer to the developer, indicating if a query is worth pursuing or requires reformulation. Our 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.

[1] http://trec.nist.gov/

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 predicting 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. We selected 21 such measures from the set of pre-retrieval measures, which refer to four aspects that 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 used.

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 predicting 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 set of software system. We evaluate both approaches for the task of concept location in our study, presented in Section 4.

[2] http://wordnet.princeton.edu/

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 prediction measures used by our approach. As this is a new problem in SE, in order to make the paper self-contained, we included an Appendix with the definitions and the formulas used to compute each of the 21 measures.

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 flowing 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 text retrieval specificity measure we considered is *SCS* (the simplified clarity score), 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 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: *AvgEntropy*, which is the average entropy value among the query terms, *MedEntropy* and *MaxEntropy*, which represent the median and the maximum entropy values, respectively, among the entropy values of the terms in the query, and *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 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 approach 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 usually 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 for 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 the 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 them 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 variance is low, then the retrieval system will be less able to differentiate between highly relevant documents and less relevant ones, making the query harder to answer.

Three measures based on VAR have been defined, i.e., *SumVAR*, which is the sum of the variances for all query terms, *AvgVAR*, computed as the average VAR value across all query terms, and *MaxVAR*, which is the maximum VAR value among the query terms. As in the case of CS, high values are expected for high-performance queries.

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 *AvgPMI* and *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.

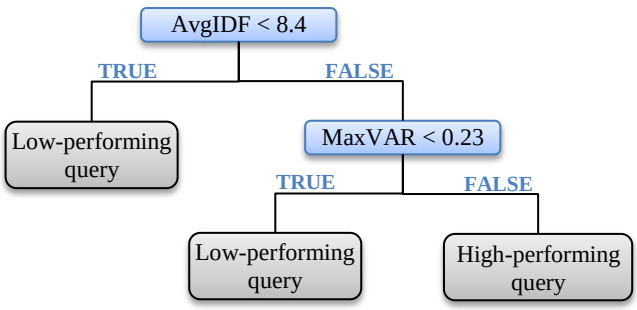

Figure 1. Example of classification tree

Classification trees have some additional benefits. First, the classification rules it produces are easy to understand by humans, which is not true for other complex models, not based on decision trees. 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 worrying about determining beforehand which ones represent the phenomenon the best. This allows us to give as input all 21 performance measures, as our classification tree will determine automatically the measures needed for the classification. Last, classification is performed very fast when using classification trees, which is an added advantage.

In our study, presented in Section 4, we use the CART (Classification and Regression Tree) implementation provided in R[3].

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 order to collect the queries needed for the case study, we used an approach frequently adopted in concept location empirical studies 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[4] is a common-based peer-production of open source enterprise resource planning applications. ATunes[5] is a full-featured media player and manager. FileZilla[6] is a graphical FTP, FTPS, and SFTP client. JEdit[7] is a programming editor and WinMerge[8] is a document differencing and merging tool.

Table 1. The systems used in the study and their properties

System	Version	Language	KLOC	#Methods	#Queries
Adempiere	3.1.0	Java	330	28,355	32
ATunes	1.10.0	Java	80	3,481	32
FileZilla	3.0.0	C++	240	3,240	36
JEdit	4.2	Java	250	5,532	28
WinMerge	2.12.2	C++	410	8,012	36
Total	**-**	**-**	**1,310**	**48,620**	**164**

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[9], 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 previously applied for 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):

- *print invoic process draft select*
  (Print Invoices process - draft & selection)
- *us garden world select date rang in todai all invoic select regardless 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 following target method was changed by the developers: doIt(), found in the process package, file InvoicePrint.java, and class InvoicePrint. 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 application 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 *high performing* if any of the target methods is retrieved in the top 20 results. Otherwise, we consider the query as *low performing*. In the above example, if a query returns the target method in top 20, then it is considered high performing. The rank

[3] www.r-project.org/

[4] http://www.adempiere.org

[5] http://www.atunes.org

[6] http://www.filezilla-project.org

[7] http://www.jedit.org

[8] http://www.winmerge.org

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

of the target method in the result list 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.

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

Measure	Short	Long	Measure	Short	Long
AvgIDF	3.69	4.83	*SCS*	1.93	0.75
MaxIDF	6.40	10.25	*AvgVAR*	0.04	0.04
DevIDF	3.14	11.60	*MaxVAR*	0.11	0.12
AvgICTF	2.72	4.04	*SumVAR*	0.17	1.18
MaxICTF	6.08	10.25	*CS*	0.13	0.39
DevICTF	3.88	13.03	*AvgSCQ*	28.45	28.78
AvgEntropy	0.60	0.53	*MaxSCQ*	33.03	37.53
MedEntropy	0.67	0.60	*SumSCQ*	113.81	777.03
MaxEntropy	0.70	1.00	*AvgPMI*	0.06	0.13
DevEntropy	0.28	1.08	*MaxPMI*	1.13	5.02
QS	0.16	0.40	**Rank (class)**	**125 (low)**	**1 (high)**

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

System	High-performing queries	Low-performing queries
Adempiere	15	17
aTunes	14	18
FileZilla	8	28
JEdit	13	15
WinMerge	18	18
Total	**68**	**96**

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. 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 accuracy of the model. The misclassification of the model has been evaluated in terms of Type I and Type II classification errors. A Type I misclassification is when the model wrongly classifies a high performing query as low performing, while a Type II misclassification is when the model wrongly classifies a low performing query as high performing.

As for the *cross-system* validation, we perform the same 4-fold cross-validation process considering all the 164 queries from the different object systems as a single dataset. When dividing the datasets 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. In this way we have a uniform distribution of queries belonging to the different systems.

These two types of validation, i.e., single-project and cross-project, were needed to derive guidelines on how to use historical data to build the classifier. In particular, we aim at analyzing whether a specialized model is required for each system or it is possible to define a generic model that can 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, i.e., it always classifies a query as high. It is worth noting that a classifier is useful only if it outperforms a random or constant classifier.

Logistic regression is used for prediction of the probability of occurrence 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:

$$\pi(X_1, X_2, \dots, X_n) = \frac{e^{C_0 + C_1 \cdot X_1 + \cdots + C_n \cdot X_n}}{1 + e^{C_0 + C_1 \cdot X_1 + \cdots + C_n \cdot X_n}}$$

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 π is commonly a dichotomous variable, and thus, assumes only two values, i.e., it states whether a query is high (1) or low (0). In our study we used the WEKA[10] 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 Experimental 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

[10] http://www.cs.waikato.ac.nz/ml/weka/

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 constant predictors classified correctly 68 (41%) and 96 (59%), respectively, and the random classifier was able to correctly assess 92 queries (56%).

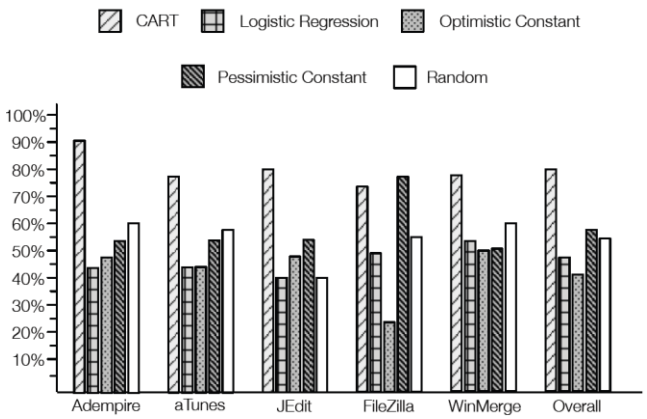

Figure 2. Accuracy achieved in the single-system evaluation

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.

It is worth noting that the accuracy achieved by the proposed classifier (79%) was obtained using very small training samples; the average dimension of the employed training samples is 24 queries. Thus, the proposed approach is able to provide excellent results with relatively little training. Such results emphasize the applicability of the proposed approach, as it does not require a large training set that might be not available for some software projects.

The classification tree is the most accurate predictor on all the systems except FileZilla. On this system, CART is able to correctly classify only 72% of the queries, whereas the pessimistic constant predictor achieves a correct classification on 77% of the queries. These results are explained by the fact that in FileZilla most of the queries are low-performing queries (28 out of 36), as shown in Table 3. This has the following consequences, which affect our results: (i) the classification tree faces an increased difficulty in identifying the characteristics of the high-performing queries, given that only a small number of such queries are available in the training set and (ii) the pessimistic constant predictor obtains a very good performance, as it always classifies queries as low-performance, and is, thus, correct in classifying all the low-performing queries, which represent the majority of the data.

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.

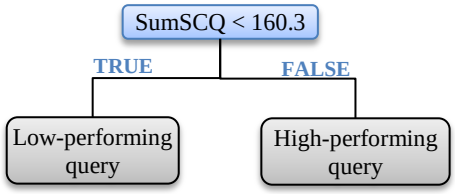

Figure 3. A classification tree on Adempiere

Table 4. Type I and Type II errors achieved in the single-system evaluation.

System	CART		Logistic Regression		Optimistic Constant		Pessimistic Constant		Random	
	Type I	Type II	Type I	Type II	Type I	Type II	Type I	Type II	Type I	Type II
Adempire	0 (0%)	3 (9%)	10 (31%)	8 (25%)	0 (0%)	17 (53%)	15 (47%)	0 (0%)	5 (16%)	8 (25%)
aTunes	3 (9%)	4 (13%)	7 (22%)	5 (16%)	0 (0%)	18 (56%)	14 (44%)	0 (0%)	3 (9%)	9 (28%)
FileZilla	2 (6%)	8 (22%)	14 (39%)	4 (11%)	0 (0%)	28 (78%)	8 (22%)	0 (0%)	3 (8%)	13 (36%)
JEdit	3 (11%)	3 (11%)	11 (39%)	6 (21%)	0 (0%)	15 (54%)	13 (46%)	0 (0%)	8 (29%)	9 (32%)
WinMerge	3 (8%)	5 (14%)	7 (19%)	10 (28%)	0 (0%)	18 (50%)	18 (50%)	0 (0%)	6 (17%)	8 (22%)

Table 5. Predictor selected by the classification tree in the single-system evaluation

System	1st fold	2nd fold	3rd fold	4th fold
Adempiere	*SumSCQ*	*SumSCQ*	*SumSCQ*	*SumSCQ*
aTunes	*DevIdf*	*MedEntropy*	*DevIdf*	*DevIdf*
JEdit	*DevIdf*	*MedEntropy*	*DevIdf*	*MedEntropy*
FileZilla	*MaxSCQ*	*DevIdf*	*AvgIdf*	*AvgIdf*
WinMerge	*AvgEntropy*	*AvgEntropy*	*AvgEntropy*	*AvgEntropy*

The results illustrate that the cross-system classification tree performs poorly, as it correctly classifies only 51% of the queries. Its results are also 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 achieved in the cross-system evaluation

Method	Type I	Type II	Type I & II	Correct
CART	44 (27%)	37 (22%)	81 (49%)	83 (51%)
Logistic Regression	49 (30%)	31 (19%)	80 (49%)	84 (51%)
Optimistic constant	0 (0%)	91 (55%)	91 (55%)	73 (45%)
Pessimistic constant	77 (47%)	0 (0%)	77 (47%)	87 (53%)
Random	49 (30%)	43 (26%)	92 (56%)	72 (44%)

4.3 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.

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 must 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 to 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* and *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 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 in 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.

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. REFERENCES

[1] Agresti, A., Categorical Data Analysis, Wiley-Interscience, 2002.

[2] Antoniol, G., Canfora, G., Casazza, G., and De Lucia, A., "Information Retrieval Models for Recovering Traceability Links between Code and Documentation", in Proc. of Int. Conf. on Software Maintenance, 2000, pp. 40-51.

[3] Asuncion, H. U., Asuncion, A., and Taylor, R. N., "Software Traceability with Topic Modeling", in Proceedings of 32nd ACM/IEEE International Conference on Software Engineering, 2010, pp. 95-104.

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

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

[6] Castro-Herrera, C., Cleland-Huang, J., and Mobasher, B., "Enhancing Stakeholder Profiles to Improve Recommendations in Online Requirements Elicitation", Int. Requirements Engineering Conf., 2009, pp. 37-46.

[7] Cleland-Huang, J., Czauderna, A., Gibiec, M., and Emenecker, J., "A machine learning approach for tracing regulatory codes to product specific requirements", in Proc. of International Conf. on Software Engineering, 2010, pp. 155-164.

[8] De Lucia, A., Oliveto, R., and Sgueglia, P., "Incremental Approach and User Feedbacks: a Silver Bullet for Traceability Recovery", in Proc. of IEEE International Conf. on Software Maintenance, 2006, pp. 299-309.

[9] Dit, B., Revelle, M., Gethers, M., and Poshyvanyk, D., "Feature location in source code: a taxonomy and survey", Journal of Software Maintenance and Evolution: Research and Practice 2011, pp. to appear.

[10] Furnas, G. W., Landauer, T. K., Gomez, L. M., and Dumais, S. T., "The Vocabulary Problem in Human-System Communication", Communications of the ACM, vol. 30, no. 11, 1987, pp. 964-971.

[11] Gay, G., Haiduc, S., Marcus, A., and Menzies, T., "On the Use of Relevance Feedback in IR-Based Concept Location", in Proceedings of IEEE International Conference on Software Maintenance, 2009, pp. 351-360.

[12] Gegick, M., Rotella, P., and Xie, T., "Identifying security bug reports via text mining: An industrial case study", in Proc. of Int. Working Conf. on Mining Software Repositories, 2010, pp. 11 - 20.

[13] Gethers, M., Kagdi, H., Dit, B., and Poshyvanyk, D., "An Adaptive Approach to Impact Analysis from Change Requests to Source Code", in Proceedings of International Conference on Automated Software Engineering, 2011.

[14] Gibiec, M., Czauderna, A., and Cleland-Huang, J., "Towards mining replacement queries for hard-to-retrieve traces", in Proceedings of IEEE/ACM International Conference On Automated Software Engineering, 2010, pp. 245-254.

[15] Grivolla, J., Jourlin, P., and De Mori, R., "Automatic Classification of Queries by Expected Retrieval Performance", in Proceedings of ACM Special interest Group on Information Retrieval, 2005.

[16] Haiduc, S., Bavota, G., Oliveto, R., Marcus, A., and De Lucia, A., "Evaluating the Specificity of Text Retrieval Queries to Support Software Engineering Tasks", in Proceedings of 34th International Conference on Software Engineering - NIER Track, 2012, pp. to appear.

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

[18] He, B. and Ounis, I., "Inferring Query Performance Using Pre-retrieval Predictors", in Proc. of International Conference String Processing and Information Retrieval, 2004, pp. 43–54.

[19] Jensen, C. and Scacchi, W., "Discovering, Modeling, and Reenacting Open Source Software Development Processes", New Trends in Software Process Modeling Series in Software Eng. and Knowledge Eng., vol. 18, 2006, pp. 1-20.

[20] Lessmann, S., Baesens, B., Mues, C., and Pietsch, S., "Benchmarking Classification Models for Software Defect Prediction: A Proposed Framework and Novel Findings.", IEEE Transactions On Software Engineering, vol. 34, no. 4, 2008, pp. 485-496.

[21] Liu, D., Marcus, A., Poshyvanyk, D., and Rajlich, V., "Feature Location via Information Retrieval based Filtering of a Single Scenario Execution Trace", in Proc. of Int. Conf. on Automated Software Engineering, 2007, pp. 234-243.

[22] Marcus, A. and Maletic, J., "Recovering Documentation-to-Source-Code Traceability Links using Latent Semantic Indexing", in Proceedings of 25th IEEE/ACM International Conference on Software Engineering, 2003, pp. 125-137.

[23] Marcus, A., Sergeyev, A., Rajlich, V., and Maletic, J., "An Information Retrieval Approach to Concept Location in Source Code", in Proceedings of 11th IEEE Working Conference on Reverse Engineering, 2004, pp. 214-223.

[24] McMillan, C., Grechanik, M., Poshyvanyk, D., Xie, Q., and Fu, C., "Portfolio: Finding Relevant Functions And Their Usages", in Proceedings of 33rd IEEE/ACM International Conference on Software Engineering, 2011, pp. 111-120.

[25] Petrenko, M. and Rajlich, V., "Variable Granularity for Improving Precision of Impact Analysis", in Proc. of Int. Conf. on Program Comprehension, 2009, pp. 10-19

[26] Petrenko, M., Rajlich, V., and Vanciu, R., "Partial Domain Comprehension in Software Evolution and Maintenance", in Proc. of Int. Conf. on Program Comprehension, 2008, pp. 13-22.

[27] Revelle, M. and Poshyvanyk, D., "An Exploratory Study on Assessing Feature Location Techniques", in Proceedings of 17th IEEE International Conference on Program Comprehension, 2009, pp. 218-222.

[28] Sridhara, G., Hill, E., Pollock, L., and Shanker, V., "Identifying Word Relations in Software: A Comparative Study of Semantic Similarity Tools", in Proceedings of 16th Int. Conf. on Program Comprehension, 2008, pp. 123-132.

[29] Starke, J., Luce, C., and Sillito, J., "Searching and Skimming: An Exploratory Study", in Proceedings of International Conference on Software Maintenance, 2009, pp. 157-166.

[30] Vercoustre, A.-M., Pehcevski, J., and Naumovski, V., "Topic Diffculty Prediction in Entity Ranking", in Proceedings of 7th International Workshop of the Initiative for the Evaluation of XML Retrieval, 2008, pp. 280-291.

[31] Voorhees, E., "The TREC robust retrieval track", ACM SIGIR Forum, vol. 39, no. 1, 2005, pp. 11-20.

[32] Wang, X., Zhang, L., Xie, T., Anvik, J., and Sun, J., "An Approach to Detecting Duplicate Bug Reports using Natural Language and Execution Information", in Proc. of 30th Int. Conf. on Software Engineering, 2008, pp. 461-470.

[33] Yin, R. K., Case Study Design: Design and Methods, 3rd ed., SAGE Publications, 2003.

[34] Yom-Tov, E., Fine, S., and Darlow, D. C. A., "Learning to Estimate Query Difficulty", in Proc. of ACM Special Interest Group on Information Retrieval, 2005, pp. 512-519.

Appendix. The 21 pre-retrieval measures of query performance

Property	Measure	Description	Formula
Specificity	*AvgIDF*	Average of the Inverse Document Frequency (*idf*) values over all query terms	$\frac{1}{\|Q\|}\sum_{q\in Q} idf(q)$
	MaxIDF	Maximum of the Inverse Document Frequency (*idf*) values over all query terms	$max_{q\in Q}(idf(q))$
	DevIDF	The standard deviation of the Inverse Document Frequency (*idf*) values over all query terms	$\sqrt{\frac{1}{\|Q\|}\sum_{q\in Q}(idf(q)-avgIDF)}$
	AvgICTF	Average Inverse Collection Term Frequency (*ictf*) values over all query terms	$\frac{1}{\|Q\|}\sum_{q\in Q} ictf(q)$
	MaxICTF	Maximum Inverse Collection Term Frequency (*ictf*) values over all query terms	$max_{q\in Q}(ictf(q))$
	DevICTF	The standard deviation of the Inverse Collection Term Frequency (*ictf*) values over all query terms	$\sqrt{\frac{1}{\|Q\|}\sum_{q\in Q}(ictf(q)-avgICTF)}$
	AvgEntropy	Average *entropy* values over all query terms	$\frac{1}{\|Q\|}\sum_{q\in Q} entropy(q)$
	MedEntropy	Median *entropy* values over all query terms	$median_{q\in Q}\ (entropy(q))$
	MaxEntropy	Maximum *entropy* values over all query terms	$max_{q\in Q}\ (entropy(q))$
	DevEntropy	The standard deviation of the *entropy* values over all query terms	$\sqrt{\frac{1}{\|Q\|}\sum_{q\in Q}(entropy(q)-avgEntropy)}$
	QS	Query Scope – the percentage of documents in the collection containing at least one of the query terms	$\frac{\|\cup_{q\in Q} D_q\|}{\|D\|}$
	SCS	Simplified Clarity Score – the Kullback-Leiber divergence of the query language model from the collection language model	$\sum_{q\in Q} p_q(Q)\cdot\log(\frac{p_q(Q)}{p_q(D)})$
Coherency	*AvgVAR*	Average of the variances of the query term weights over the documents containing the query term (*VAR*), over all query terms	$\frac{1}{\|Q\|}\sum_{q\in Q} VAR(q)$
	MaxVAR	Maximum of the variances of the query term weights over the documents containing the query term (*VAR*), over all query terms	$max_{q\in Q}(VAR(q))$
	SumVAR	Sum of the variances of the query term weights over the documents containing the query term (*VAR*), over all query terms	$\sum_{q\in Q} VAR(q)$
	CS	Coherence Score – the average of the pairwise similarity between all pairs of documents containing one of the query terms (*cs*) among all	$\frac{1}{\|Q\|}\sum_{q\in Q} cs(q)$
Similarity	*AvgSCQ*	The average of the collection-query similarity (*SCQ*) over all query terms	$\frac{1}{\|Q\|}\sum_{q\in Q} SCQ(q)$
	MaxSCQ	The maximum of the collection-query similarity (*SCQ*) over all query terms	$max_{q\in Q}(SCQ(q))$
	SumSCQ	The sum of the collection-query similarity (*SCQ*) over all query terms	$\sum_{q\in Q} SCQ(q)$
Term relatedness	*AvgPMI*	Average Pointwise Mutual Information (*PMI*) over all pairs of terms in the query	$\frac{2\,(\|Q\|-2)!}{(\|Q\|)!}\sum_{q_1,q_2\in Q} PMI(q_1,q_2)$
	MaxPMI	Maximum Pointwise Mutual Information (*PMI*) over all pairs of terms in the query	$max_{q_1,q_2\in Q}(PMI(q_1,q_2))$

$idf(t)=\log(\frac{\|D\|}{\|D_t\|})$	$p_t(d)=\frac{tf(t,d)}{\|d\|}$	$w(t,d)=\frac{1}{\|d\|}\log(1+tf(t,d))\cdot idf(t)$	$entropy(t)=\sum_{d\in D_t} p_t(d)\cdot\log_{\|D\|} p_t(d)$
$ictf(t)=\log(\frac{\|D\|}{tf(t,D)})$	$p_t(D)=\frac{tf(t,D)}{\|D\|}$	$SCQ(t)=(1+\log(ctf(t,D))\cdot idf(t)$	$PMI(t_1,t_2)=log\frac{p_{t_1,t_2}(D)}{p_{t_1}(D)\cdot p_{t_2}(D)}$
$\overline{w}_t=\frac{1}{\|D_t\|}\sum_{d\in D_t} w(t,d)$	$p_t(Q)=\frac{tf(t,Q)}{\|Q\|}$	$cs(t)=\frac{\sum_{(d_i,d_j)\in D_t} sim(d_i,d_j)}{\|D_t\|\cdot(\|D_t\|-1)}$	$VAR(t)=\sqrt{\frac{\sum_{d\in D_t}(w(t,d)-\overline{w}_t)^2}{df(t)}}$

Q –the set of query terms; q – a term in the query; D – the set of documents in the collection; D_t –the set of documents containing term t
d – a document in the document collection D; *tf(t,D)* – the frequency of term t in all docs; *tf(t,d)* – the frequency of term t in d
tf(t,Q) – the frequency of term t in the query; $sim(d_i,d_j)$ – the cosine similarity between the vector-space representations of d_i and d_j