1-1-2013

Supporting Text Retrieval Query Formulation In Software Engineering

Sonia Cristina Haiduc
Wayne State University,

SUPPORTING TEXT RETRIEVAL QUERY FORMULATION IN SOFTWARE ENGINEERING

by

SONIA HAIDUC

DISSERTATION

Submitted to the Graduate School

of Wayne State University,

Detroit, Michigan

in partial fulfillment of the requirements

for the degree of

DOCTOR OF PHILOSOPHY

2013

MAJOR: COMPUTER SCIENCE

Approved by:

Advisor Date

DEDICATION

To the loving memory of my mom, Maria Haiduc, the most amazing woman I have known. You have been my guiding light, my source of optimism, my strength. You taught me to enjoy every single day and to cherish the beauty of all things. Your never-ending support, belief, and unconditional love brought me here. I am forever grateful. Thank you for brightening my days for 29 years. I miss you every day.

ACKNOWLEDGEMENTS

I would first like to thank my academic advisor, Andrian Marcus for all he has given me. His guidance, knowledge, support, advice, openness, dedication, availability, work ethic, and example were priceless to me as a forming researcher and adult and have made me understand and truly appreciate what research and a job in academia are all about. My gratitude for all the time and energy he invested in me are beyond words.

I would like to thank Václav Rajlich, Marwan Abi-Antoun, Lori Pollock, and Denys Poshyvanyk for serving on my prospectus and dissertation committee and providing me with excellent and detailed feedback on my work and dissertation.

I am grateful to Lori Pollock, Jonathan Maletic, Andrea De Lucia, Tim Menzies, and Javad Abdollahi for patiently writing and sending reference letters for me during my job search.

I would also like to thank my collaborators Gabriele Bavota, Rocco Oliveto, Andrea De Lucia, Jairo Aponte, Massimiliano di Penta, Tim Menzies, Gregory Gay, Laura Moreno, Surafel Lemma Abebe, Paolo Tonella, Giuseppe di Rosa, Valerio Maggio, Anna Corazza, and Sergio di Martino. It has been always a pleasure to work alongside you. I have learned a lot due to our collaboration.

I thank my friends in the Department of Computer Science for their continued support over all these years, for patiently listening to my occasional complaints, and for being always available to do something fun when I needed it. In particular, I would like to thank Michele Donato, for always being there for me when I needed him, and for feeding me gourmet food over the past four years. I thank Laura Moreno for her support and help, and for gracefully playing the role of the little sister I never had. Also, I thank my dear friend Grace Metri for our coffee breaks that brightened my days and for all the encouragement she has given me. I also thank Calin Voichita and Radu Vanciu for their moral and logistic support over the past seven years. It has been great taking this journey together.

Last and foremost I thank my dad and my grandparents for their unconditional love and support through all these years, and for encouraging me to pursue what I wanted, even though it meant taking me away from them. Your love surpassed all the distance that physically separates us. I have always felt you close and I thank you for that.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

CHAPTER 1 INTRODUCTION

During software development and evolution a variety of software artifacts are created, such as, requirements, change requests, bug descriptions, user manuals, developer communication, use cases, design documents, source code, test cases, etc. These artifacts have different representations and contain different types of information, i.e., structural (e.g., control and data flow, the package organization in an OO system), dynamic (e.g., execution traces), process (change logs, time and activity logs, etc.), and textual (identifiers and comments in source code, developer communication, user manuals, requirements, etc.). The textual information found in software artifacts captures knowledge about the problem and solution domain, about developers' intentions, client demands, etc. and is the most common type of information found in software, often surpassing other types of information by an order of magnitude. Text is also the common form of information representation among various artifacts at different abstraction levels, making it easy for developers, among other things, to understand what the software is doing and to make decisions for their current task.

For very small software systems, developers could read all the text found in software artifacts and extract and use only the information that is needed for their task at hand. However, as the size and complexity of the systems increases, tools are required in order to extract, store, analyze, retrieve, and present this information to the developers. Text Retrieval (TR) techniques are one category of techniques that have been successfully used for this task over the past few decades. They are used to retrieve relevant information for a particular task from a large set of software artifacts based on an input query and present a series of advantages over other techniques.

First, they return a list of ranked results and therefore indicate the order in which the results should be examined. This is not true for other approaches, such as, regular expression search, which return results in no particular order. TR techniques are also lightweight and programming language independent. In addition, they provide complementary information to that provided by structural and dynamic techniques [85, 100] and scale well to large software systems.

For their versatility and good results, TR techniques have been applied in the context of more than 25 different software engineering tasks, including impact analysis [21, 49], concept and feature location [26, 87, 98], bug localization [77, 107], clone detection [82, 120], refactoring [11, 12], measuring the cohesion and coupling of software [85, 100], and traceability link recovery [6, 83].

1.1 Motivation

Despite the advantages TR techniques present for software engineering researchers, they also pose challenges which can hinder their further adoption in development environments and industrial settings. Approaches using TR require a query as input and the usefulness of the returned results depends strongly on this query and its relationship to the text in the software artifacts. The *quality of a query* indicates the relevance to the task at hand of the results returned by TR in response to the query. The higher the quality of a query, the better the results answer the information need expressed by the user. *High quality queries* retrieve the relevant documents in the top of the result list returned by TR techniques, while *low-quality queries* either retrieve the desired documents in the bottom part of the list of results, or they do not retrieve them at all.

Choosing a high quality query is a difficult task. One of the main challenges is the fact that in many cases the person writing the query is not the one that wrote the software artifacts, and may therefore be unfamiliar with the text the artifacts contain. This leads to the so-called “vocabulary mismatch problem” [46], due to the use of a different terminology in the software artifacts than the one used when formulating the query and in irrelevant results being retrieved. For example, when a developer wants to search the source code for the implementation of a feature, she may be unfamiliar with the identifiers used in the source code to refer to particular aspects of the feature and may use different words in the query. Another factor that makes query formulation difficult is the fact that TR techniques are often based on complicated mathematical models and text similarities and it is difficult for developers to understand how to write their queries in order to enable these models to match them to relevant results. All this can lead to poorly chosen queries, which in turn means time and effort wasted by developers by analyzing the irrelevant results retrieved in response to the queries.

The quality of a query can give an indication of whether the results returned by a TR technique are worth investigating or if rather a reformulation of the query should be sought instead. Knowing the quality of the query before the results are analyzed could lead to time saved in the cases when irrelevant results are returned. However, the only way currently available to determine if a query led to the wanted artifacts is by manually inspecting the list of results. Designing automatic ways to predict the quality of queries before the results are investigated could therefore benefit developers making use of TR techniques. However, no work has yet addressed automatic query quality detection in the software engineering domain.

When the results returned by a TR technique in response to a query are unsatisfactory (i.e., the query is of poor quality), the developer can *reformulate* the query with the purpose of improving it. However, this can be just as hard for developers as formulating the query in the first place and studies [119] have shown that some developers have a hard time writing a good query even after several reformulations. This is due to the fact that it is hard to understand when and why a query fails, thus it is difficult to determine what should be added or removed from the original query. More than that, different queries may require different approaches for reformulation (e.g., some may require removing terms, others may benefit from terms being added or replaced). Automating the reformulation process such that the automatically reformulated queries lead to better results than the original queries could therefore reduce developer effort and improve the software engineering tasks supported by TR techniques.

1.2 Thesis Statement

The thesis statement of this dissertation is the following:

When using text retrieval techniques to support of software engineering tasks, low quality queries lead to lost developer time and effort. Automatic approaches can accurately measure and predict the quality of text retrieval queries in the context of software engineering tasks, such as, concept location and can reformulate queries such that they lead to better results than the original queries.

The contributions of this dissertation supporting the above thesis statement are described in the following section.

1.3 Contributions

We introduce query quality measurement and prediction approaches in the context of software engineering tasks in order to automatically determine the quality of TR queries. We also propose query reformulation approaches meant to automate the query reformulation process, making it easier for developers to focus on the code and show that the queries reformulated using these approaches lead to better results than the original queries in most cases. We evaluate these techniques in the context of concept location in source code and show their benefits for this software engineering task. On the long run, we expect that the proposed approaches will contribute directly to the reduction of developer effort and implicitly the reduction of software evolution costs.

The research contributions of this dissertation are the following:

- *Determining the Specificity of TR Queries for software engineering Tasks*. We present the first measure for capturing a query quality attribute, namely specificity, for TR-based software engineering tasks. The specificity of a query measures how discriminative the terms in the query are for describing the current information need. The new measure, called Query Specificity Index (QSI), is able to capture the specificity of a query prior to running a TR engine and relies on information theory in order to determine the ability of a query to discriminate between relevant and irrelevant artifacts. We apply QSI in the context of concept location in source code and we show that it is able to better reflect the results of a query for this task than the leading specificity measure proposed in the field of natural language document retrieval, i.e., AvgIDF.

- *Automatic Query Quality Prediction for TR in software engineering.* We introduce a novel approach, called QualQ, able to automatically predict the quality of queries in the context of software engineering tasks. QualQ captures the properties of queries using a set of 28 query measures and based on them is able to learn, by using Classification and Regression Trees, the patterns that distinguish high quality queries from low quality ones. After it builds a model from a set of training queries, QualQ is able to automatically predict the quality of new queries, based on measuring only their properties. One important aspect of QualQ is the fact that it performs implicit feature selection and therefore is able to determine and use only the subset of query measures needed for capturing the quality of queries in a particular dataset. We evaluated QualQ in the context of concept location in source code and showed that it is able to correctly predict the quality of queries in 85% of the cases. This classification of the queries can be used as an indication by developers if the results of a search should be investigated or rather the query should be reformulated and the search rerun.
- *Semi-Automatic Query Reformulation for Concept Location using Rocchio.* We present a semi-automatic approach for reformulating TR queries in the context of concept location in source code. The approach makes use of developer feedback by incorporating the Rocchio relevance feedback mechanism. After each search, developers are asked to rate the first few results in the list as relevant or irrelevant and the query is then automatically reformulated based on this feedback. The process is iterative and can be repeated several times, until the developer finds the right source code artifacts. We evaluated the approach in

a study on concept location and we have shown that, while not a silver bullet, the Rocchio-based relevance feedback mechanism can generally improve the results if TR concept location.

- *Automatic Query Reformulation for TR in software engineering.* We introduce Refoqus, a novel approach for automatically reformulating TR queries in the context of software engineering tasks by automatically determining and applying the best reformulation approach for a query based on its properties. Refoqus makes use of the set of 28 query measures in order to determine the distinguishing characteristics of the queries in a dataset, and makes use of Classification and Regression trees in order to learn the best reformulation for a type of query, among four options. When a new query is issues, Refoqus can then determine the properties of the query and based on the model it built select the right reformulation approach for the given query from the ones available. Refoqus is flexible and can be adapted to include more reformulation approaches when needed. We evaluated Refoqus in the context of concept location in source code and the results of the study revealed that Refoqus is able to improve or preserve the results of TR queries for CL in 84% of the cases.

1.4 Dissertation Organization

The rest of the dissertation is organized as follows. Chapter 2 briefly describes the software engineering tasks that have been so far addressed using TR techniques and then presents background information on the process followed for applying TR approaches for software engineering applications. The overview of the steps in this process gives a perspective on where query formulation fits in. The chapter then offers a detailed description of the task of concept location (CL), which is the task used for the evaluation of the proposed approaches throughout the rest of the dissertation. The description of CL includes a detailed explanation of how each step in the TR process is applied for this task and offers also a brief overview of the related work that has addressed each step in this process in the context of CL.

Chapter 3 introduces novel approaches for measuring and predicting the quality of queries in the context of software engineering tasks. More specifically, section 3.1 presents a new measure for capturing the specificity of queries in software engineering, (i.e., the Query Specificity Index) and then presents an evaluation of this measure for concept location. Section 3.2 introduces QualQ, a novel approach able to automatically predict the quality of queries in the context of software engineering tasks. This approach is one of the main contributions of the dissertation. Section 3.2 presents also an evaluation of QualQ in the context of concept location. Section 3.3 concludes chapter 3 with an overview of the related work on query quality measurement and prediction in the field of software engineering, as well as in natural language document retrieval.

Chapter 4 addresses the issue of query reformulation and proposes approaches to help developers with this task. Section 4.1 introduces a semi-automatic approach which uses developer feedback about the relevance of the retrieved results in order to automatically reformulate the query. The section presents an evaluation of the approach for concept location. Section 4.2 proposes a novel approach which is able to automatically reformulate the query, and also to choose the best reformulation techniques among a series of options based on the properties of a query. The evaluation of the approach for concept location is also presented in the same section. Section 4.3 then presents a brief survey of the related work on query reformulation in the field of software engineering.

Chapter 5 concludes the dissertation, summarizing its main contributions, as well as directions for continuing this line of research beyond the scope of the dissertation.

1.5 Bibliographical Notes

Parts of this dissertation were previously published. This section also outlines some of the materials that were produced in collaboration with other researchers.

The new query measure QSI was previously published [55] and presented in the New Ideas and Emerging Results (NIER) track at the 34th IEEE/ACM International Conference on Software Engineering (ICSE'12).

The initial version of QualQ, as well as an initial evaluation were previously published [54] and presented in the main research track at the 27th IEEE/ACM International Conference on Automated Software Engineering (ASE'12). However, the version of QualQ already published made use only of pre-retrieval query measures, while the version of QualQ presented in this dissertation includes both pre- and post-retrieval measures and leads to better results than the initial version. The evaluation performed in our earlier work [54] uses the initial, pre-retrieval based version of the approach, and was performed on 5 software systems. The study presented in this dissertation evaluates the latest approach, making use also of post-retrieval information, and was performed on seven software systems. Both the work presented in Section 3.1 and 3.2 was performed in collaboration with Gabriele Bavota (University of Salerno, Italy), Rocco Oliveto (University of Molise, Italy), and Andrea de Lucia (University of

Salerno, Italy), who contributed mostly on the implementation of the approach and training the machine learning technique.

The materials presented in Section 4.1 were previously presented at the 25th IEEE International Conference on Software Maintenance (ICSM'09) [47]. This work was done in collaboration with Gregory Gay (West Virginia University) and his advisor Tim Menzies (West Virginia University). They contributed mostly on the implementation of the approach.

Section 4.2 is based on a recent publication [53], presented at the 35th IEEE/ACM International Conference on Software Engineering (ICSE'13). This publication was done in collaboration with Gabriele Bavota (University of Salerno, Italy), Rocco Oliveto (University of Molise, Italy), Andrea de Lucia (University of Salerno, Italy), and Tim Menzies (West Virginia University), who contributed mostly on implementation, and training the machine learning approach.

The work presented in this dissertation has not been included in any other dissertation nor thesis.

CHAPTER 2 BACKGROUND ON THE USE OF TEXT RETRIEVAL FOR SOFTWARE ENGINEERING TASKS

Text Retrieval (TR) approaches have been used in software engineering to leverage the textual information found in software artifacts, such as, requirements, source code, documentation, user manuals, etc. TR-based approaches for software engineering have become very popular since their introduction due to their capability to capture information complementary to that of static and dynamic approaches. The earliest work on using TR in software engineering began over two decades ago and focused on constructing software libraries [78, 79] and code reuse [45, 58, 90]. Since then, TR techniques have been used for more than 25 different software engineering tasks, including traceability link recovery between different types of software artifacts [1, 6, 8, 37, 40, 56, 74, 84, 88, 103, 115, 126], concept [7, 25, 26, 47, 72, 87, 89, 98, 114], concern [62, 117, 124], feature [43, 73, 99, 108, 109, 131], aspect [91], or bug [4, 15, 36, 77, 92, 93, 107, 132] location, impact analysis [5, 21, 22, 48, 60, 66, 102], defect prediction [16, 121], software restructuring and refactoring [13, 14, 30, 68, 80, 94] and so on. The last decade has witnessed a dramatic increase in the use of TA techniques to address software engineering tasks, with more than 300 papers published in the field.

Applying TR techniques to software engineering often implies seeing the software engineering tasks as classic TR problems: given a document collection and a query, determine those documents, i.e., software artifacts, from the collection that are relevant to the query. The idea is to treat software artifacts as a text corpus and use TR methods to index the corpus and build a search engine, which allows developers to search the software much like they search other sources of digital information (e.g., the internet).

2.1 The Process of Using Text Retrieval for Software Engineering

When the software engineering task is seen as a retrieval problem, TR-based approaches follow the same process, no matter the particular software engineering task for which they are being used. The steps of this process may be instantiated differently based on the TR method used and the software engineering task. They are described in the following subsections.

2.1.1 Corpus Creation

The first step in using TR techniques is to define a collection of text documents, also known as *corpus*, which are extracted from the software artifacts. Documents can be extracted at different granularities from an artifact. For example, in the case of source code, a document could be represented by structural elements of the code, such as, a source code file, a class, a function or method, a line of code, etc. In the case of verbose natural language artifacts, such as, requirements or user manuals, sentences, paragraphs, sections, or chapters can represent the documents. Thus, a software artifact may be represented by one or more documents in the corpus. The document granularity needs to be decided up front according to the needs of the task at hand.

Once the document granularity is determined, the documents are identified, extracted, and included in the corpus. Note that certain software artifacts may contain constructs which are not text, such as, images, attachments, etc. These are filtered out during this step. Also, what exactly is extracted depends on the type of software artifact. For example, the subject and the whole body of an email may be extracted, as all this information may be considered useful. On the other hand, in the case of source code, it is customary to extract only identifiers, comments, and strings from each source code document, as they usually represent the elements capturing the problem domain concepts and programmer intentions.

The granularity of the documents can influence greatly the results of text retrieval, as the frequency and term co-occurrence information change depending on how the documents are chosen.

2.1.2 Corpus Normalization

After the corpus is extracted, a few optional, corpus normalization steps can be performed before the documents are indexed by the text retrieval technique. These steps are usually performed to facilitate retrieval, and can impact the results retrieved. These normalization steps will be applied also to the query later in the process. This section describes the most common corpus normalization techniques used in software engineering applications.

Tokenization and Identifier Splitting.

Tokenization represents the process of converting a stream of characters into a sequence of tokens, or terms. Tokenization is done by removing punctuation, brackets, and extraneous separation characters, such as, spaces, tabs, and line breaks.

Identifier splitting follows tokenization and is usually employed when using text retrieval on source code. The success of retrieval depends on many factors, but one of the basic conditions that need to be satisfied is that the vocabulary of the corpus needs to correspond to the vocabulary used by developers when formulating queries. As developers usually express their information need using dictionary words, it is important that source code corpus contains also such words. Identifiers, however, are often composed of several concatenated dictionary words. It is important therefore that the corpus normalization includes a step where the identifiers are decomposed into their constituent words. This step is usually performed automatically, as manual identifier splitting is unfeasible due to the high number of identifiers in a software system.

The simplest approaches for identifier splitting are based on common conventions for separating words in identifiers, such as using camel case, underscore, numbers and symbols as separators. By splitting the identifiers where such separators are encountered, the constituent words are obtained. For example, "SETpointer", "set_pointer", "setPointer" would be all split to "set" and "pointer".

More advanced techniques make use of dictionaries and abbreviation lists in order to identify words in the cases where common naming conventions are not used. For example, the identifiers "setptr" would be split into "set" and "pointer" based on these techniques.

Splitting identifiers requires also a decision about keeping the original form of the identifiers or not. Keeping the original identifiers along with the words resulted after splitting can help when the developers included whole identifiers in their queries. On the other hand, when no identifiers are included in the queries, keeping the original form might negatively impact the results due to unnecessary increase of the vocabulary size.

Stop Words Removal

In this processing phase, terms that do not contribute to the semantics of the extracted documents, known as *stop words*, are removed from the corpus. Such terms are considered noise and include programming keywords (e.g., "for", "class", etc.) and common English terms, such as, conjunctions (e.g., “and”, “or”, etc.), pronouns (e.g., “he”, “they”, “it”, etc.), prepositions (e.g., “at”, “on”, “in”, etc.), common adverbs, etc. Some words, such as the name of the system, certain prefixes or abbreviations, can appear in many documents of the system without adding to the meaning of those particular documents. In this case, they can also be included in the stop words list. The terms remaining after filtering out the stop words are considered meaningful for the documents in the corpus.

Other than stop word list, a stop word function can be used. Such a function is used to prune out all the words having a length less than a fixed threshold (usually 3). Generally, good results are achieved using both the stop word function and the stop word list [9].

Stemming

Stemming is the process for reducing inflected (or sometimes derived) words to their stem, base or root form, generally a written word form. For example, “run”, “runs”, “ran” and “running” are all forms of the same root, “run”. The role of a stemmer is to attribute all the derived forms to their root form, “run”. Stemming is used to improve matching between similar words, in order to capture their underlying semantics, disregarding the specifics in the lexical form. Stemming can lead to retrieving more relevant documents, as the query gets matched to all documents containing similar terms to the ones in the query, instead of just those containing exact matches. Stemming can have a significant impact on the vocabulary size. Stemming can also significantly reduce the vocabulary size of a corpus, which can also influence the results of the retrieval.

Stemming programs are commonly referred to as stemming algorithms or stemmers. Designing stemmers has been a long-standing problem in computer science. The first paper on the subject was published in 1968. There are several types of stemmers, but the most used one in software engineering research is the Porter stemmer, introduced in 1980 by Martin Porter [97]. This stemmer works by removing well known word suffixes in several, iterative steps. Other stemmers used by researchers in software engineering are WordNet's morphstr[1] function, the Krovetz stemmer [67], and the Snowball[2] stemmer.

Stemmers can suffer from two types of errors. The first one is understemming, which appears when a stemmer does not remove enough suffixes in order to reduce all the forms of the same lexeme to the same stem. Understemming may reduce the number of relevant results since fewer results may match the words in the query. In contrast, overstemming happens when a stemmer reduces several words to the same lexeme, even when the words have different meanings. This may increase the number of irrelevant results returned by search, due to overmatching the query. Different stemmers have different strengths and weaknesses, and may affect the results of TR differently.

2.1.3 Corpus Indexing

In this step, a mathematical representation of the corpus is built, which is stored by the text retrieval approach in a quickly accessible format called *index*. Each document in the software corpus has a corresponding entry in the index. In order for the results of TR techniques to be accurate, the index needs to be remade when a significant number

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

[2] http://snowball.tartarus.org/

of documents in the corpus change or new documents are added. Indexing is different for every text retrieval technique and is often what sets the various TR techniques apart. Some of the most popular TR used in software engineering applications include the Vector Space Model (VSM) [112], Latent Semantic Indexing (LSI) [39] and Latent Dirichlet Allocation (LDA) [17]. VSM is one of the first and most widely used indexing techniques in software engineering applications. As we make use of the VSM indexing approach in the studies presented in this dissertation, we offer a brief description of the main concepts behind it below.

In the Vector Space Model the documents in a corpus, as well as the queries written by users are considered as bag of words and are represented as vectors in an n-dimensional space, where n is the set of unique words found in all the documents in that corpus. Each term in the corpus represents a distinct dimension in the n-dimensional space considered. The corpus can be represented as term-by-artifact matrix that captures all the artifact vectors and represents the distribution of terms in the artifacts. If a term in the collection appears in a document, then the value associated to the document for that dimension will be greater than zero. Otherwise, if the term appears in the collection, but not in a particular document, the value for that dimension in the document vector will be zero.

Once the query and the documents are represented as n-dimensional vectors, the similarity between the query and each of the software artifacts in the collection is measured. Several ways for computing this similarity between the vectors have been proposed, but the most commonly used one is the *cosine similarity*, which represents the cosine of the angle between the query vector and an artifact vector. A smaller angle between the query and an artifact leads to a larger cosine value and indicates a better match to the query and thus a higher rank in the retrieved set of results.

Independent of the text retrieval technique used, when building the index entry for a document the terms in that document can be assigned a higher or lower importance, or *weight*, based on two criteria: how well they describe the current document (*local weight*) and how they relate to the entire corpus (*global weight*). This is a major difference between text retrieval techniques and keyword matching approaches, which usually consider all terms equal.

The term weight can have a significant impact on the results returned by text retrieval techniques. Several term weighting schemes have been defined but the most common combination of local and global weights is known as *Term Frequency – Inverse Document Frequency* (*TF-IDF*). *Term Frequency* (*TF*) represents the frequency of a term in a document in the corpus, and a high TF indicates that the term is often used in that document and therefore likely representative for describing it. The *Inverse Document Frequency* (*IDF*) is a global weighting scheme and it represents the inverse of the number of documents in which a term appears in the corpus. A high IDF indicates that the term is found in few documents in the corpus and may therefore be more representative for the documents in which it appears than a term which appears in many documents in the corpus, thus having a low IDF. The value of TF-IDF increases proportionally to the number of times a word appears in the document, but is offset by the frequency of the word in the corpus, which helps to control for the fact that some words are generally more common than others.

2.1.4 Query Formulation

In this step, a text query composed of a series of words, is written. This query expresses a particular information need and is either formulated by a human (i.e., a user or a developer), or automatically extracted from a given software artifact. For example, when dealing with finding code relevant to a task, developers can use the information contained in the description of the task at hand (e.g., a bug report, a new feature request, etc.), as well as previous knowledge, the system documentation or any other sources of information as a starting point for formulating the query. After the query is formulated, it is subjected to the same normalization steps applied to the corpus (i.e., identifier splitting, stop words removal, stemming, etc.). Once normalized, the query is run by the text retrieval technique. When reformulating the query, developers return to this step in the process and issue a new query.

2.1.5 Retrieval of Relevant Results

Once the query is formulated and run, the text retrieval technique computes semantic similarities between the query and every document present in the corpus. It then returns as a result an ordered list of documents, starting with the ones that match the query best at the top of the list. Returning a ranked list of results is one of the key aspects that differentiate TR from keyword-based approaches, which return results in no particular order, and all results are considered equal matches to the query.

There are several similarity measures that can be used when matching the query to documents in the corpus. The similarity measures that can be used in a particular case depend on the type of text retrieval technique used. The choice of similarity must be done with care, as it can have an impact on the results. The *cosine similarity* is the most popular choice when using VSM or LSI. It is a measure of similarity between two vectors in an n-dimensional space and represents the cosine of the angle between them. The cosine values range from -1 to 1. The cosine of the angle between two vectors thus determines whether two vectors are pointing in roughly the same direction. Its formula is based on the Euclidean dot product formula.

2.1.6 Results Examination

After the list of documents has been retrieved, the ranked list can be examined. The higher a document is situated in the list of results, the higher it is ranked by the system as containing the wanted information. Thus, usually the order of examination is from the top of the list to the bottom. The examination of the results is performed by the developer and for every document examined, she decides if it is relevant or not to the task at hand. If the developer is able to find the documents that satisfy the information need among the top results in the list, the search succeeded and the process ends. Else, if new knowledge obtained from the investigated documents helps formulate a better query (e.g., narrow down the search criteria), then the query should be reformulated. Otherwise, the next document in the list should be examined.

2.2 Text Retrieval-Based Concept Location

Concept location in source code is one of the tasks that have been often addressed using TR techniques [42] and where the classic TR problem is being employed. We present here concept location as an example application for TR techniques in software engineering and we use throughout the rest of the dissertation as an application instance to evaluate the approaches we propose.

Concept location is most often defined in the context of software change [106] (see Figure 2-1), which occurs in the presence of a source code modification request. The software change process [29] starts with the modification request and ends with a set of changes to the existing code and addition of new code. The software maintainer undertakes a set of activities [104] to determine the parts of the software that need to be changed: concept location, impact analysis, change propagation, and refactoring. Concept location starts with the change request and ends when the developer finds the first location in the source code where a change must be implemented (e.g., a method). The next activity in the software change process, i.e., impact analysis [18] starts from the result of concept location and identifies the rest of the source code locations that are affected by the change. Some researchers have adopted a different definition of concept location, considering it the task of determining all the locations in the code

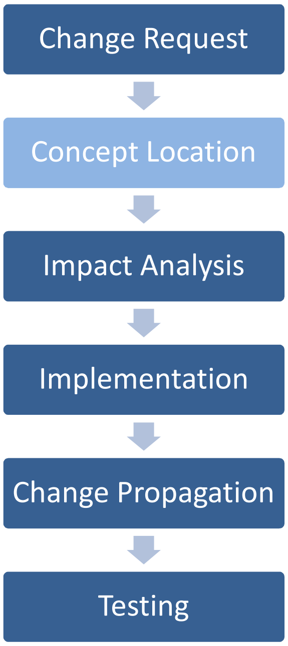


Figure 2-1. Simplified view of the software change process (adapted from [105]). Concept location starts with the change requests and produces the input for impact analysis

where changes need to be implemented. However, in the context of software change, we consider these as two different tasks, i.e., concept location, responsible for identifying the first change location, and impact analysis, which is responsible for finding the rest of the change locations starting from the one determined by concept location.

One particular instance of concept location is *feature location* [42], which deals with identifying the source code corresponding to a specific functionality of the software system that is accessible and observable by the user (i.e., a feature). In other words, the difference between concept and feature location is that feature location is focused on special concepts (i.e., features). All features are concepts, but not all concepts are features. For example, a linked list is a concept from the solution domain which may be implemented in the source code, yet it is not a specific feature of the system. *Bug localization* [107] is a specific instance of feature location which deals with identifying unwanted features, (i.e., bugs) in the code of a software system.

Concern location [124] is another variant of concept location and is concerned with locating anything that stakeholders of the software consider to be a conceptual unit, such as features, requirements, design idioms, or implementation mechanisms. The difference between concept and concern location is in the context and the scope. Concept location is performed during software change (hence it has a specific input and output), whereas concern location is a context agnostic view of the activity. Also, concern location usually involves finding all the code elements participating in the implementation of a concern, rather than locating just one of them. *Aspect mining* [91] is an instance of concern location dealing with cross-cutting concerns.

During concept location, the information need of the developers is to find a place in the code that needs to change in response to the modification request. In order to find the entity to change, developers search and navigate the source code. In this process, developers can use as a starting point the textual description of the change they need to perform, which often provides information that helps formulate a query for a search, choose a starting point for the navigation of the code, or choose a scenario for executing the program. In any case, the task of the developers is identifying the right fragment of code from the large amount of possibilities available in the source code of a system.

Approaches for concept location help developers with this task by suggesting a list of source code artifacts (i.e., classes, methods, etc.) in the system where such a change may take place. TR-based approaches for concept location start from the change request or user query and recommend an ordered list of code artifacts, where the ones found at the top of the list are the most likely to change in response to the modification request. The steps in the process described in section 2.1 are instantiated by concept location approaches as described below.

Corpus Creation

The task of concept location is focused on finding locations in code. Therefore, the documents in the corpus represent code entities. The granularity of the documents can influence greatly the results of concept location, as the frequency and term co-occurrence information change depending on how the documents are chosen. Researchers have used various document granularities for concept location and its variants. However, there seems to be a preference for smaller granularity levels, such as, methods [25, 73, 77, 98] in the case of object oriented systems and functions [3, 87, 130, 131] for other programming paradigms. Even though less widespread, file level granularity [34, 35] and class level granularity [86] were used in some work in the field making use of text retrieval. In the evaluations presented in this dissertation, method level granularity is considered.

Corpus Normalization

In concept and feature location recommendation systems using text retrieval, it is common to apply identifier splitting approaches, since the documents in the corpus are source code elements. The effect of different identifier splitting approaches has been studied in several papers in recent years and new techniques have been developed to ensure a better splitting [20, 29, 41, 44, 52, 69]. Abbreviation expansion of identifier terms has also been proposed as a way to increase the quality of identifiers [59, 69, 70]

Stop words removal is also a common normalization step applied in concept location approaches based on TR. A list of English stop words and programming keywords is commonly used. Some approaches also make use of stemming, which can have a significant impact on the results of text retrieval. Recent works have investigated the amplitude of this impact in concept location recommendation systems by analyzing the variation in results when different stemming algorithms were used [63].

Corpus Indexing

The internal representation of documents during text retrieval-based concept location depends on the particular TR model used. Different text retrieval models perform in different ways and researchers have employed various such models in the attempt to find the one that performs the best.

The Vector Space Model (VSM) [112] was among the first retrieval models being used in the context of concept location and its related activities (i.e., feature location, bug localization, etc.) and over time it has remained a common choice for researchers in the field [3, 33, 47, 108, 113, 114, 131]. Latent Semantic Analysis [39] was first introduced by Marcus et al. [87] for concept location and it has been used since then as one of the standard TR techniques for the task [35, 73, 86, 93, 99, 101, 116]. Language Models [96] and Latent Dirichlet Allocation [17] have also been used in the context of concept location in the past few years [25, 26, 76, 77]. Some works in the field have compared the performance of several text retrieval techniques in the context of concept location, feature location or bug localization [107], indicating that simple retrieval models such as VSM may work better than sophisticated models such as LSI or LDA. In our evaluations in the context of concept location presented in this dissertation we make use of the implementation of VSM available in the Lucene[3] library.

Query Formulation

For concept location, the queries are often formulated by developers, using the text of a change modification (i.e., bug report or new feature request) as a starting point for writing the query [73, 87]. Over the past few years, a common approach adopted in recent years for concept location evaluation is to automatically formulate queries by considering the textual description of the change requests as an initial query [43, 47, 77, 107]. This practice is described in more detail in section 2.2.1.

Retrieval of Relevant Artifacts

The similarity between the query and each of the documents in the corpus is computed and the artifacts most similar to the query are returned and ordered according

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

to their relevance to the query. The similarity used depends on the TR, but cosine similarity is the most common choice for concept location in source code.

Results Examination

Once the results are returned, they are examined either by developers, in order to find the wanted code, either by tools, which use this information for different purposes, such as, filtering the results or query reformulation.

2.2.1 Evaluation for TR-based Concept Location Approaches

To evaluate concept location approaches, the *effectiveness measure* is most commonly used. It was first proposed by Poshyvanyk et al. [99]. The measure was introduced in order to allow the comparison between concept location approaches using text retrieval and those using other methods, like dynamic analysis. Before the introduction of the effectiveness measure, the metrics used for evaluating text retrieval-based concept location techniques were precision and recall, which may not be adequate for evaluating other approaches and present some issues when applied in the context of concept location.

To deal with these issues, effectiveness was defined as the rank of the first changed method related to the concept or feature of interest. This allows also for a measurement of the effort of a developer during the location process, which can be defined as the number of methods which appear in the final ranked list that the developer needs to investigate. A lower value effectiveness value indicates less effort, hence a more effective technique. In our evaluations of the proposed approaches, we make use of the effectiveness measure.

In order to obtain evaluation data for concept location, researchers have often used an approach based on change reenactment [65] and user simulation, i.e., automatically extracting TR queries and the changed code from bug reports found in online bug tracking systems. In an ideal situation, the complete change should be implemented and tested to verify that this location is correct. However, this requires an enormous effort to perform post-hoc. Reenactment based on historical data allows to assess the correctness of concept location without complete implementation and testing. Many papers in the field in the past few years [43, 47, 77, 107] have adopted this approach to evaluate text retrieval approaches. We adopt the same evaluation procedure in the evaluation studies on concept location presented in this dissertation.

CHAPTER 3 MEASURING AND PREDICTING THE QUALITY OF QUERIES FOR TEXT RETRIEVAL APPLICATIONS IN SOFTWARE ENGINEERING

The results of TR techniques strongly depend on the textual query formulated by a human or extracted automatically from an existing textual artifact. The *quality of a query* refers 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). Queries are considered of *high quality* if they retrieve the relevant artifacts among the top results returned by TR techniques. Conversely, *low-quality queries* 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. This definition of high- and low- quality queries can be fine-tuned considering the needs of a developer in the context of a particular software engineering task. For example, in some applications, a high-quality query may be considered one that retrieves one of the relevant artifacts in the top 10 list of ranked results. In other applications, however, a high quality query could be one that retrieves all the relevant documents in the top 25 results.

Formulating a high quality query is not an easy task, and the only way for developers to know if a query led to the wanted results or not is to actually analyze the returned results. The problem with this approach is the fact that by the time developers realize that a query is of low quality, they already spent significant time analyzing irrelevant search results. It would be desirable, therefore, to have an approach that is able to automatically predict and communicate the quality of queries to developers before the analysis of the search results begins. This information could be used by developers as a recommendation indicating if the results are worth investigating or, instead, the query should be reformulated and the search executed again. Determining the quality of a query is a first step towards writing better TR queries and speeding up the software engineering tasks making use of such techniques.

In order to predict the quality of a query for TR applications in software engineering, however, it is first important to design appropriate measures for capturing the quality of queries in a software engineering context. This dissertation presents in Section 3.1 the first approach for capturing a quality property, namely specificity, of TR queries in the context of software engineering tasks. Section 3.1.2.4 then presents the first approach in software engineering for predicting the quality of TR queries, which makes use of the specificity measure defined in Section 3.1 along with other measures to predict the quality of the results returned by a query.

3.1 Determining the Specificity of Text Retrieval Queries to Support Software Engineering Tasks

The problem of capturing the quality of TR queries in the context of software engineering tasks has not yet been addressed. Predicting the quality of queries in software engineering bares clear resemblances to the analogous problem in the field of Text Retrieval for natural language (NL) documents, where a series of approaches to capture the quality of queries have been proposed [23]. However, the techniques used for NL documents do not always apply to software artifacts. For example, many approaches for determining the quality of queries in NL rely on the rules governing the English language. However, these rules often do not apply in software artifacts, which contain much more than just natural language. For example, Sridhara et al. [118]

showed that the semantic relationships between words (i.e., synonymy, hyponymy, etc.) are different in source code than in English. In consequence, one must carefully consider the existing query quality prediction approaches in the context of software engineering, select only those that are applicable to software artifacts, and develop new measures specific for software where needed.

We studied measures that capture the query *specificity*, which is among the most investigated query properties in natural language document retrieval [23]. *Specificity* measures how discriminative the terms in the query are for describing the current information need and is usually reflected by the query terms' distribution over the collection of documents. For example, a query that contains words which are found in half of the software artifacts in the system is less specific than a query having words that appear in only a small percentage of the artifacts. TR techniques have a harder time answering non-specific queries, as it is difficult to discriminate among the multitude of artifacts containing query terms and retrieve only the ones that are truly relevant to the task at hand. Therefore, a TR search based on a non-specific query is likely to retrieve many irrelevant documents.

Among the measures for query specificity proposed in natural language document retrieval, average inverse document frequency, or avgIDF [32] performs the best on natural language corpora. AvgIDF is computed as the average of the inverse document frequency (IDF) values of the terms in the query. The Inverse Document Frequency (IDF) of a term is the inverse of the number of documents in the collection in which a term appears and is a measure of the term's importance for any particular document it appears in. If a term's IDF is low, it means the term appears in many documents in the collection, so it is not specific for any document in particular. If, on the other hand, the term appears in few documents, its IDF will be high, and the term is specific and representative for those documents it appears in. A query term with a high IDF makes it easier to retrieve only relevant documents to the query, thus query terms should have high IDF.

AvgIDF was found to have a relatively high correlation with the retrieval performance of TR techniques. The retrieval performance indicates the quality of the results retrieved by the TR techniques and therefore reflects the quality of a query. In order to assess if avgIDF could be used to indicate the quality of queries also on software data, we measured the correlation between avgIDF and the retrieval effort on existing concept location data. Concept location is the process of identifying where a code change is to start, in response to a change request and many concept location techniques use TR as the underlying mechanism for tool support. The weak correlation between AvgIDF and the effectiveness measure for concept location (see Section 3.1.2) indicates that the metric does not work well in the case of concept location in source code. This indicates that there is the need to propose new ways to capture query specificity for queries in the context of software engineering tasks.

3.1.1 The Query Specificity Index

We introduce a novel metric, called *Query Specificity Index (QSI)*, to automatically detect the specificity of queries for TR approaches in the context of software engineering tasks. The metric is able to measure the expected specificity of a query prior to the retrieval and relies on information theory in order to determine the ability of a query to discriminate between relevant and irrelevant artifacts. QSI can be used in several ways in software engineering tasks. For example, when dealing with user formulated queries, QSI can be used to recommend the developer to reformulate the query before spending time on investigating the retrieval results. Some software engineering tasks rely on queries extracted from existing artifacts, such as, traceability recovery. During this process a lot of effort is spent on the manual validation of the retrieved links and on providing feedback to the retrieval system. QSI can be used to prioritize the links that should be investigated first by the users.

QSI is based on concepts from information theory. More specifically, it uses the concept of information entropy [31] to measure the specificity of a term in the query. Information entropy measures the amount of uncertainty of a discrete random variable [31]. In our case, the random variable is represented by a term in the query, while the documents in the corpus (i.e., software artifacts in our context) are the possible states that the variable can assume (i.e., the term does or does not occur in an artifact). This means that the more scattered the term is in the corpus the higher its entropy will be. Our conjecture is that a specific query should contain terms that are not very scattered through the corpus, but that are found in a high concentration in few documents (i.e., the relevant ones). We call such terms specific terms. While avgIDF is also based on the principle that terms with a low scattering are more specific, it overlooks one important aspect: the concentration (i.e., the frequency of terms in the documents where they appear). In consequence, it does not make a distinction between terms that are found many times in a few documents in the corpus and terms that are found only once in few documents in the corpus. QSI addresses this limitation by considering also the concentration of terms in documents, along with their dispersion.

Formally, the entropy of a term *t* is computed as:

$$entropy(t) = \sum_{d \in D_t} p(d) \times \, log_{\mu} p(d)$$

Where:

- D_t is the set of documents containing the term *t*
- *μ* is the number of documents in the corpus
- *p(d)* represents the probability that the random variable (term) *t* is in the state (document) *d*. Such a probability is computed as the ratio between the number of occurrences of the term *t* in the document *d* over the total number of occurrences of the term *t* in all the documents in the corpus.

The entropy has a value in the interval of [0, 1]. The higher the value, the less the term is discriminating. We thus compute the QSI based on the entropy of its terms as:

$$QSI_q = 1 - median\{entropy_t \mid t \in q\}$$

We chose to use the median over the average because the median is less impacted by skewed distributions of values, caused by a few non-specific terms that may occur in otherwise highly specific queries. Hence, we avoid situations where a few terms have a strong impact on the QSI.

In the rest of this chapter we use and discuss QSI in the context of concept location (CL) in source code. Figure 3-1 shows an example of computing the QSI of two different queries formulated for locating the code related to a change request. In this example the software is composed of six classes (C_1– C_6) and each is converted to a document in the corpus. The change request is a bug fix request: *"The window containing the user interface does not scale to full screen when pushing the F11 button on the keyboard."* The goal of CL in this example is to identify the class containing the bug (in this particular example is C_2). Figure 3-1 shows the two queries (i.e., Q_1 and QualQ) as well as the document corpus. The figure also lists the terms used in the two queries that occur in each of the classes (the number of occurrences appears in parenthesis).

As we can see, query Q_1 contains terms having very high entropy. All the terms from Q_1 appear in several classes of the system and thus they do not help much in

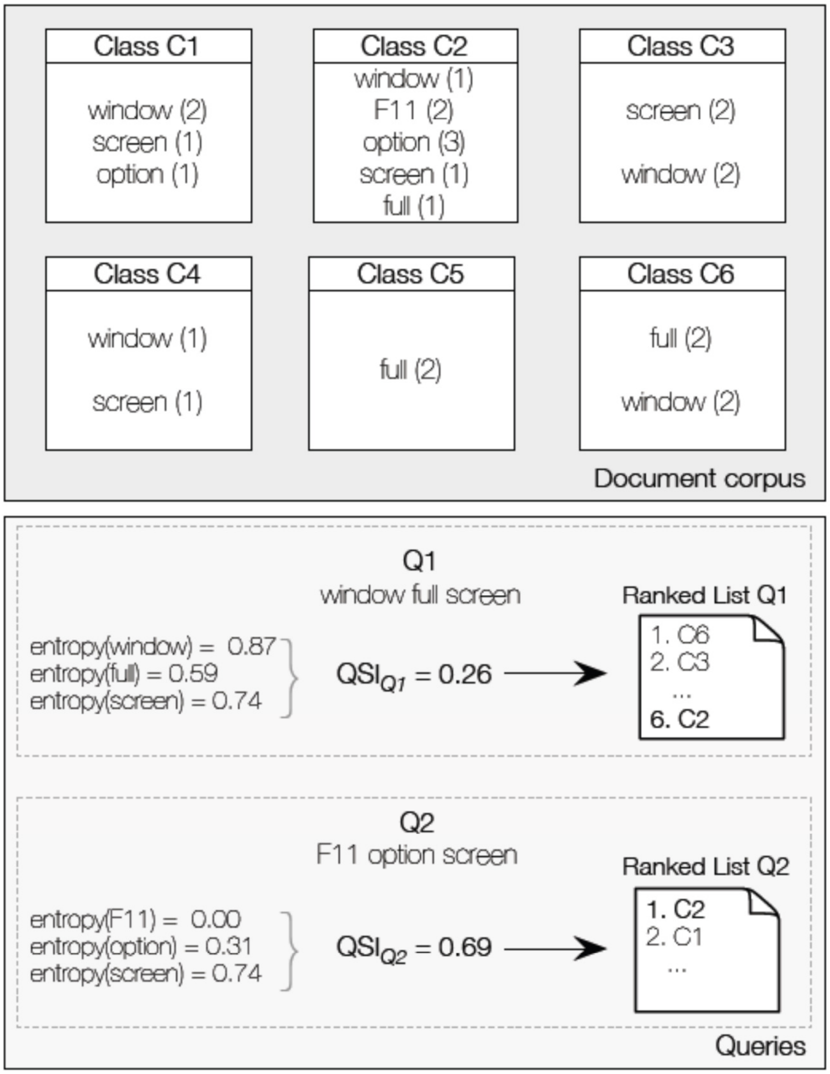


Figure 3-1. QSI for two queries for the same bug report

discriminating between the documents. For this reason the query specificity of Q_1 is not high (QSI_{Q1} = 0.26). As we can see, when executing Q_1, the relevant class (C_2) is the sixth class in the ranked list. Thus, the developer would need to investigate six classes in order to locate the faulty one. Conversely, the terms in QualQ are well focused on a particular document in the corpus (C_2), thus exhibit a low entropy and consequently a high QSI (QSI_{Q2} = 0.69). With such a query the relevant class is the first class in the ranked list, indicating a minimum effort spent on CL. Note that low entropy does not necessarily imply that the documents that the query is focused on are the relevant ones.

3.1.2 Evaluation on Concept Location in Source Code

We performed a study in the context of concept location (CL) in source code, where the goal was to assess if query specificity metrics can be used as indicators of the effort spent on TR - based CL by developers (which we consider as a measure of query quality). The rest of this section is organized as follows. Section 3.1.2.1 offers information about the design of the concept location study we performed, Section 3.1.2.2 describes the dataset we used, Section 3.1.2.3 presents the results of the study and Section 3.1.2.4 concludes by describing the threats to the validity of the study results.

3.1.2.1 Study Design

In this study we aim to determine how well the two specificity measures mentioned in the previous section, i.e., AvgIDF and QSI, reflect the quality of a query in the context of concept location in source code, as reflected by the effort spent on the task. In particular, we are interested in answering the following research questions:

RQ1: Is AvgIDF a good indicator of query quality in the context of concept location in source code?

RQ2: Is QSI a good indicator of query quality in the context of concept location in source code?

In the context of CL, we associate the quality of queries with the *effectiveness measure,* which approximates the CL effort as the number of retrieved results (i.e., source code documents) that a developer needs to examine before finding the first relevant document to the change (we assume the developer is examining the results in the order provided by the TR engine). The effectiveness measure is commonly used in the existing research that empirically evaluates CL techniques [42]. To answer the research questions above we measure the correlation between the AvgIDF and QSI, on one hand, and the CL effectiveness measure, on the other hand.

We used TR in a standard way in this study. We built the source code document corpus considering every method in the system as a separate document. For each method we extracted the text found in its identifiers and comments in the source code. We then normalized the text using identifier splitting, stop words removal (i.e., we removed common English words and programming keywords) and stemming. The same normalization techniques were applied on the extracted queries. The corpus was indexed by with Lucene[4], a popular implementation of the Vector Space Model TR technique.

The CL effort for a given query is the highest rank of any of the target methods in the ranked list of search results. As mentioned before, this is a standard measure used when evaluating CL techniques [42]. We computed the Pearson product-Moment

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

Correlation Coefficient (PMCC) [28] between the values obtained for avgIDF and QSI, respectively, and the CL effort measure for each of the queries. PMCC is a measure of correlation between two variables X and Y defined in [−1, 1], where 1 represents a perfect positive linear relationship, −1 represents a perfect negative linear relationship, and some value between −1 and 1 indicates the degree of linear dependence between X and Y. Cohen et al. [28] provided a set of guidelines for the interpretation of the correlation coefficient. It is assumed that there is no correlation when $0 < \rho < 0.1$, small correlation when $0.1 < \rho < 0.3$, medium correlation when $0.3 < \rho < 0.5$, and strong correlation when $0.5 < \rho < 1$. Similar intervals also apply for negative correlations.

3.1.2.2 Data

We use change data from three open source systems, namely Adempiere[5] 3.1.0, ATunes[6] 1.10.0, and JEdit[7] 4.2. Adempiere is a common-based peer-production of open source enterprise resource planning applications, ATunes is a full-featured media player and manager, and JEdit is a programming editor. For each object system, we selected a set of change requests from its issue tracking system corresponding to bugs present in the investigated version of the software, but fixed in a later version. The bug reports were selected such that they contained patches, from which we could identify the methods changed in order to fix the bugs. We determined the set of methods that were modified in order to fix each bug, which we used then as the golden set for CL. 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 composed from the title of the change request (i.e., short description) and the

[5] http://www.adempiere.com/
[6] http://www.atunes.org/
[7] http://jedit.org/

second query composed from the long description of the change request. Table 3-1 reports the number of queries used for each system.

Table 3-1. The systems used in the study

System	Version	#Methods	#Queries
Adempiere	3.1.0	28,354	34
ATunes	1.10.0	3,480	30
JEdit	4.2	5,532	30
Overall	-	37,366	94

3.1.2.3 Results and Discussion

Table 3-2 reports the correlation values between AvgIDF and QSI on one hand, respectively, and the effectiveness measure on the other hand for each of the three object systems. Since high values for AvgIDF and QSI are in general associated with high quality queries, and low values of the effectiveness measure are associated with low developer effort in the context of TR-based concept location, AvgIDF and QSI are considered good indicators of query quality in the context of concept location if there is a negative correlation between them and the effectiveness measure.

Research Question 1

As it can be observed from Table 3-2, AvgIDF obtains a negative correlation with the effectiveness measure only for ATunes, where a medium correlation is observed (-0.35) and JEdit, where a strong correlation is observed (-0.51). Over all systems, the average correlation between AvgIDF and the effectiveness measure is small (-0.13). The results indicate that AvgIDF is able to partially capture the quality of a query for concept location only in some systems, and that is definitely not a silver bullet.

Research Question 2

When observing the correlation of QSI with the effectiveness measure, we note that QSI obtains a negative correlation for all systems and that the correlation is medium or high in all cases. Also, the results indicate that QSI obtains a high average correlation with the effectiveness measure over all systems in the study (-0.53). One interesting observation is that the correlation obtained by QSI with the effectiveness measure is stronger than the correlation obtained in natural language document retrieval between the performance of a query and any individual query quality measure tested [23]. The highest average Pearson correlation obtained by any quality measure in natural language document retrieval is 0.47 compared to 0.53 for QSI.

Compared with AvgIDF, QSI achieves a higher correlation than AvgIDF on two of the three object systems. Only for JEdit the correlation obtained by AvgIDF is slightly higher (-0.51 vs. -0.47). However, the overall average correlation for QSI is significantly higher than that of AvgIDF.

We conclude that, overall, QSI is a better indicator of query quality than AvgIDF in the context of concept location in source code, achieving a high average correlation with the effectiveness measure.

Table 3-2. Linear Correlation between CL Effort and the Specificity Measures

System	**Correlation**	
	AvgIDF	**QSI**
Adempiere	0.48	**-0.72**
ATunes	-0.35	**-0.43**
JEdit	**-0.51**	-0.47
Overall	-0.13	**-0.53**

3.1.2.4 Threats to Validity

This section presents the threats to the validity of the study and of the results obtained, organized by threat category [128].

Threats to *construct validity* concern the relationship between theory and observation. To evaluate the CL task, we used the effectiveness measure, which is widely used in concept/feature location studies for this purpose since it provides a good estimation of the effort that a developer needs to spend in a TR-based concept location task. Also, for determining if AvgIDF and QSI are good indicators of query quality in the context of TR-based concept location, we used the Pearson Product-Moment Correlation Coefficient , which has been used in the field of natural language document retrieval for the same purposes [23].

Threats to *internal validity* concern co-factors that can influence the results. In our study we automatically extracted the set of queries from online bug tracking systems. Such queries are approximations of actual user queries. However, developers are often faced with unfamiliar systems, in which cases they must rely on outside sources of information, such as bug 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.

Threats to *conclusion validity* concern the relationship between treatment and outcome. We used standard statistical methods, i.e., the Pearson Product-Moment Correlation Coefficient to capture the correlation between AvgIDF and QSI and the effectiveness measure. These statistical methods are the de-facto standard used in the field of natural language document retrieval to capture the relationship between query performance and query quality measures [23].

Threats to *external validity* refer to the generalization of the results we obtained. Regarding the systems used for the case study, we tried to mitigate this threat, by selecting three software systems from diverse domains. A larger set of queries and more systems would clearly strengthen the results from this perspective. While we used data from several systems, we only used a single TR engine (i.e., Lucene). The results may differ when using other TR engines. The last threat to external validity is related to the fact that we only performed a study for the task of TR-based concept location. Thus, we cannot (and do not) generalize the results to other software engineering tasks or the obtained results.

3.2 Automatic Query Quality Prediction for Retrieval of Software Artifacts

The *quality of a query* captures how well the query retrieves the desired documents when executed by a TR approach. A *high-quality query* retrieves the relevant documents on top of the results list. Conversely, a *low-quality query* either retrieves the desired documents in the bottom part of the list of the results, or it does not retrieve them at all. We consider here a binary view of query quality, as we associate it with a decision developers have to make: to reformulate or not the query. If we predict the query to be of low quality, this can be used as an indication by developers that the query should be reformulated. If it is of high quality, the query should be kept and the results investigated. We plan to investigate ranges of query quality in our future work, beyond this dissertation.

When low-quality queries are executed, the software engineer will spend time and effort analyzing irrelevant search results, and can lose the confidence in the TR-based tools, to no fault of their own. Therefore, there is the need to provide software engineers using TR tools with feedback related to the quality of the query being run. Such tools will warn developers when the query is likely to lead to poor results or, on the contrary, indicate that the results are likely to contain useful information, in the case when the query is of high quality. The software developer can then take the right action without frustration or time wasted.

We aim at predicting the quality of TR queries and presenting this information to the software engineer before her analysis of the results begins. To this end, we propose a solution to the problem of query quality prediction in software engineering by adapting solutions from the field of natural language (NL) document retrieval [23] to their use on software data. While similar, the problem of query quality prediction in software engineering has essential differences with respect to NL document retrieval, due to the properties of software artifacts, which contain different information than NL documents (e.g., the text extracted from the source code is not always correct English). Therefore, we carefully analyzed, selected, and adapted those NL techniques which are applicable to software data. First, we performed an analysis of all the techniques existing for NL and we eliminated those that relied on English semantic and syntactic rules. At the same time, since we want to generate and present information about the quality of the query in real time to the developer that is searching source code, we did not consider those approaches which required a long processing time (i.e., higher than one minute). We identified a set of 28 measures of query quality that meet these criteria. These measures have been previously used mostly in isolation as predictors of query quality, with very few approaches making use of two or more measures.

Our approach, called *QualQ* (***Qual****ity of* ***Q****ueries*) makes use of machine learning techniques and the selected set of 28 pre-retrieval (collected before the query is run by the TR engine) and post-retrieval (collected after the query is executed by the TR engine) measures of query quality from the field of NL document retrieval. Based on the 28 query quality measures and a set of training data, it uses a classifier to learn the characteristics of high- and low-quality queries used in the context of software engineering tasks and is able to predict the quality of new queries with high accuracy. Based on these rules, the employed classifier is able to predict the quality of new queries. Therefore, our approach offers a clear and pragmatic indication to developers if a query is worth pursuing (high quality queries) or requires reformulation (low quality queries).

Even though the current implementation of QualQ starts from the 28 measures presented in the next section, when predicting the quality of queries in a system, our approach performs as a first, internal step, a feature selection step (the type of classifier we use, i.e., classification and regression trees, does this automatically when constructing the learned model). This means that the classifier we use will determine which measures among the 28 are truly representative to capture the quality of queries for a particular dataset and then it will use only that reduced set of measures to make the prediction. Therefore, for evaluating any of the data sets in our study, only two to three measures (these can change between systems or between different evaluation rounds in the same system) are used at a time by the classifier. Also, it is worth mentioning that the setup of our approach is generic, allowing the replacement of the 28 measures with any other set of measures deemed to capture query quality.

3.2.1 Query Quality Properties and Measures

Existing query quality measures are categorized into *pre-retrieval* and *post-retrieval* [23], depending on the moment when they are employed and the type of information about the query they capture. *Pre-retrieval measures* are computed before the query is run, i.e., before the results to the query are retrieved. They capture various linguistic and statistical properties of the query and of the document collection. In contrast, *post-retrieval measures* make use of the list of results returned by the query, and are, thus, employed after the query is run and the results are retrieved. The two types of measures capture complementary properties of the query and our approach makes use of both pre-retrieval and post-retrieval measures for a better prediction power. The following subsections explain in detail the pre- and post-retrieval measures we use and the quality properties of the query they measure.

3.2.1.1 Pre-Retrieval Measures

Our approach makes use of a set of 21 pre-retrieval measures, which assess four different aspects of query quality: *specificity*, *similarity*, *coherency*, and *term relatedness*. The measures were selected among all those proposed in the field of natural language document retrieval such that they can be applied to any type of software artifacts. We present the 21 measures used by approach below, categorized according to the query quality property they capture.

Specificity Measures

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, but the way this information is captured differs from measure to measure. For our approach, we considered eight specificity measures from the text retrieval literature [23], along with four new measures. The specificity measures used are described below, grouped according to the core metric they are based on. A summary description of each specificity measure as well as the formulas used to compute these measures can be found in Table 3-3 and Table 3-4.

Measures Based on the Inverse Document Frequency of a Term

The *Inverse Document Frequency (IDF)* of a term is the inverse of the number of documents in the collection in which a term appears and is a measure of the term's importance for any particular document it appears in. If a term's inverse document frequency is low, it means the term appears in many documents in the collection, so it is not specific for any document in particular.

Table 3-3. The eight query specificity measures from natural language document retrieval used by QualQ.

Measure	Description	Formula
AvgIDF	Average of the Inverse Document Frequency (*idf*) values over all query terms	$\frac{1}{\lvert Q\rvert}\sum_{q\in Q} idf(q)$
MaxIDF	Maximum of the Inverse Document Frequency (*idf*) values over all query terms	$\underset{q\in Q}{max}(idf(q))$
DevIDF	The standard deviation of the Inverse Document Frequency (*idf*) values over all query terms	$\sqrt{\frac{1}{\lvert Q\rvert}\sum_{q\in Q}(idf(q) - avgIDF)}$
AvgICTF	Average Inverse Collection Term Frequency (*ictf*) values over all query terms	$\frac{1}{\lvert Q\rvert}\sum_{q\in Q} ictf(q)$
MaxICTF	Maximum Inverse Collection Term Frequency (*ictf*) values over all query terms	$\underset{q\in Q}{max}(ictf(q))$
DevICTF	The standard deviation of the Inverse Collection Term Frequency (*ictf*) values over all query terms	$\sqrt{\frac{1}{\lvert Q\rvert}\sum_{q\in Q}(ictf(q) - avgICTF)}$
QS	Query Scope – the percentage of documents in the collection containing at least one of the query terms	$\frac{\lvert \cup_{q\in Q} D_q \rvert}{\lvert D\rvert}$
SCS	Simplified Clarity Score – the Kullback-Leiber divergence of the query language model from the collection language model	$\sum_{q\in Q} p_q(Q)\cdot\log(\frac{p_q(Q)}{p_q(D)})$
$idf(t) = \log(\frac{\lvert D\rvert}{\lvert D_t\rvert})$	$ictf(t) = \log(\frac{\lvert D\rvert}{tf(t,D)})$	***Q*** –the set of query terms; ***D*** – the set of documents in the collection; ***q*** – a term in the query;
D		

Table 3-4. The four new, entropy-based measures of specificity used by QualQ

Measure	Description	Formula
AvgEntropy	Average *entropy* values over all query terms	$\frac{1}{\lvert Q\rvert}\sum_{q\in Q} entropy(q)$
MedEntropy	Median *entropy* values over all query terms	$\underset{q\in Q}{median}\ (entropy(q))$
MaxEntropy	Maximum *entropy* values over all query terms	$\underset{q\in Q}{max}\ (entropy(q))$
DevEntropy	The standard deviation of the *entropy* values over all query terms	$\sqrt{\frac{1}{\lvert Q\rvert}\sum_{q\in Q}(entropy(q) - avgEntropy)}$
$p_t(d) = \frac{tf(t,d)}{\lvert d\rvert}$	$p_t(D) = \frac{tf(t,D)}{\lvert D\rvert}$	$entropy(t) = \sum_{d\in D_t} p_t(d)\cdot\log_{\lvert D\rvert} p_t(d)$
Q –the set of query terms; ***D*** – the set of documents in the collection; ***q*** – a term in Q; ***D_t*** –the set of documents containing term t; ***d*** – a document in D ***tf(t,D)*** – the frequency of term *t* in all docs; ***tf(t,d)*** – the frequency of term *t* in *d;*		

If, on the other hand, the term appears in few documents, its IDF will be high, and the term is specific and representative for those documents it appears in. A query term with a high IDF makes it easier to retrieve only relevant documents to the query, thus query terms should have high IDF. We use three measures based on IDF for capturing the specificity of a query. The *Average Inverse Document Frequency* (*AvgIDF*) captures the average value of IDF among all query terms, and a high-quality query should have a high AvgIDF. The *Maximum Inverse Document Frequency* (*MaxIDF*), which represents the maximum IDF value across all query terms, is a popular variation of the average, and is also expected to assume high values in the case of high-quality queries. The *Standard Deviation of the Inverse Document Frequency (DevIDF)* captures how much the values of IDF vary among all the query terms. The assumption is that a low variance reflects the lack of dominant, discriminative terms in the query, which may prevent the retrieval of relevant documents. In consequence, DevIDF is expected to be high for high-quality, specific queries.

Measures Based on the Inverse Collection Term Frequency of a Term

The *Inverse Collection Term Frequency (ICTF)* is another way to capture the specificity of a term. ICTF is the inverse of the number of occurrences of a term in the entire document collection. A specific term has a low ICTF, and the assumption is the similar to that used in the case of IDF: the more a term is used in the documents in the collection, the less specific it is, leading to a difficulty in discriminating the relevant documents based on it.

Three specificity measures we use are based on ICTF: the *Average Inverse Collection Term Frequency (AvgICTF)*, the *Maximum Inverse Collection Term Frequency (MaxICTF)*, and the *Standard Deviation of Inverse Collection Term Frequency (DevICTF)*, which represent the average, maximum and standard deviation of the ICTF values among all the terms in the query. Highly specific terms have a high ICTF values and a highly specific query should have a high AvgICTF, MaxICTF, and DevICTF.

Measures of Scope

Query Scope (QS) is another specificity measure, independent of IDF and ICTF. It 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.

Measures of Clarity

The last specificity measure we considered is the *Simplified Clarify Score(SCS)*, which measures the divergence of the query language model from the collection language model, as an indicator of query specificity. More specifically, the measure considers that a query is not specific if the language used in it (i.e., terms and their frequency) 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.

Measures Based on Entropy

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-quality queries. In a study, described in Chapter 3.1, we have shown that entropy is a better indicator of query specificity for software engineering 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 across 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-quality query are low for the first three entropy-based measures. For DevEntropy, high values are wanted.

Similarity Measures

The *similarity* between the query and the entire document collection is considered as being another indicator of query quality. 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 quality 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 quality 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 quality queries. The formulae used to compute each of these measures can be found in Table 3-5.

Coherency Measures

Another quality 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 considered to be. The measures used to capture the coherency of queries, along with the formulae used to compute them are described in Table 3-6.

Table 3-5. The query similarity measures used by QualQ.

Measure	**Description**	**Formula**
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	$\sum_{q \in Q} SCQ(q)$
	$SCQ(t) = (1 + \log(tf(t, D)) \cdot idf(t)$	$idf(t) = \log(\frac{\|D\|}{\|D_t\|})$
Q –the set of query terms; ***D*** – the set of documents in the collection ***q*** – a term in the query; ***tf(t,D)*** – the frequency of term *t* in *D*		

The *coherence score* (*CS*) of a term is one of the measures used for this quality 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-quality 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 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 quality queries.

Table 3-6. The query coherency measures used by QualQ.

Measure	**Description**	**Formula**
AvgVAR	Average of the variances of the query term weights over the documents containing the query term (*VAR*), over all query terms	$\frac{1}{\lvert Q\rvert}\sum_{q\in Q} VAR(q)$
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	The average of the pairwise similarity between all pairs of documents containing one of the query terms (*cs*) among all	$\frac{1}{\lvert Q\rvert}\sum_{q\in Q} cs(q)$
	$VAR(t) = \sqrt{\frac{\sum_{d\in D_t}(w(t,d) - \bar{w}_t)^2}{df(t)}}$	$cs(t) = \frac{\sum_{(d_i,d_j)\in D_t} sim(d_i,d_j)}{\lvert D_t\rvert \cdot (\lvert D_t\rvert - 1)}$
	$w(t,d) = \frac{1}{\lvert d\rvert}\log(1 + tf(t,d)) \cdot idf(t)$	$\bar{w}_t = \frac{1}{\lvert D_t\rvert}\sum_{d\in D_t} w(t,d)$
Q –the set of query terms; ***D*** – the set of documents in the collection; ***q*** – a term in Q ***sim(di,dj)*** – the cosine similarity between the vector-space representations of d_i and d_j		

Term Relatedness Measures

Term relatedness measures make use of term co-occurrence statistics in order to assess the quality 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. For example, the query “money order” would be a high-quality query if the terms "money" and "order" frequently co-occur in the corpus.

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. High average and maximum PMI values indicate a query with strongly related terms and indicate high query quality. The formulae used for the term relatedness measures can be found in Table 3-7.

Table 3-7. The term relatedness measures used by QualQ.

Measure	**Description**	**Formula**
AvgPMI	Average Pointwise Mutual Information (*PMI*) over all pairs of terms in the query	$\frac{2\,(\lvert Q\rvert-2)!}{(\lvert Q\rvert)!}\sum_{q_1,q_2\in Q} PMI(q_1,q_2)$
MaxPMI	Maximum Pointwise Mutual Information (*PMI*) over all pairs of terms in the query	$\max_{q_1,q_2\in Q}(PMI(q_1,q_2))$
	$PMI(t_1,t_2)=log\frac{p_{t_1,t_2}(D)}{p_{t_1}(D)\cdot\, p_{t_2}(D)}$	$p_t(D)=\frac{tf(t,D)}{\lvert D\rvert}$
Q –the set of query terms; ***D*** – the set of documents in the collection; ***q*** – a term in Q		

3.2.1.2 Post-Retrieval Measures

Post-retrieval query quality measures analyze the list of results retrieved in response to the query and make a prediction based on the language used in the top documents. The list of results provides a different type of information about the query than the pre-retrieval measures. For example, the coherence of the search results, i.e., how focused they are on aspects related to the query, is not captured by the query text and is hard to assess without an analysis of the results list. Post-retrieval measures are categorized into different paradigms, based on the properties of the query and of the results list they capture. We used seven measures that capture the *robustness* and *score distribution* of the results, described below.

Robustness Measures

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

We use five robustness measures: *Subquery Overlap*, *Robustness Score*, *First Rank Change*, *Clustering Tendency*, and *Spatial Autocorrelation*.

Subquery Overlap

Subquery Overlap perturbs the query and captures the extent (i.e., the standard deviation) of the overlap between the result set retrieved by the entire query and the result sets retrieved by individual query terms. This is based on the observation made in the field of natural language document retrieval that some query terms have little or no influence on the retrieved documents, especially in the case of low quality queries. A low standard deviation of the overlap values indicates that the list of results is robust to modifications of the query. Therefore, the lower the standard deviation is, the higher the quality of the query. In order to obtain the Subquery Overlap, we used the following algorithm:

a) Run the original query q, and obtain the result list R

b) Run each individual query term q_t in the original query as a separate query and obtain the result list R_t.

c) For each individual query term q_t, compute the overlap between the first k (k=10) documents in R and the first k documents in R_t (i.e., number of documents found in both result lists)

d) The overall score of the query is the standard deviation of the values of the overlap considered for each term in the query.

Robustness Score

To measure the *Robustness Score*, a document perturbation measure, the terms' weights in the top relevant documents are slightly perturbed and the resulting documents are re-ranked. The correlation between the initial rank and that after modification is considered. A higher robustness score indicates higher query quality. The algorithm followed to compute the Robustness Score is:

a) Run the original query q, and obtain the result list R

b) Take the top 50 documents in R and consider them as ranked list L

c) For each document d in L, get a perturbed document d' from d in the following way:

   - All terms t from the corpus that do not appear in document d, will not be included in d' neither.
   - All terms t from the corpus that appear in document d with frequency f, but do not appear in the query will appear in document d' with the same frequency f.
   - Each term t that appears in d with frequency f and appears also in the query q will appear also in d' , but with a frequency f', which is a random number obtained from a Poisson distribution P(λ) with λ = f

d) The new 50 documents obtained are ranked according to the query q, resulting in a second ranked list L', where each document corresponds to a document in L.

e) Compute the Spearman rank correlation between the positions of the 50 documents in L and the positions of their corresponding perturbed documents in L' and record the correlation obtained.

f) Repeat steps c) – e) 100 times, and the final robustness score is the average Spearman correlation between the 100 runs.

First Rank Change

First Rank Change captures the probability of a document found on the first position in the list of results to still remain on the first position after a perturbation is applied to it. A high quality query will have a high first rank change, corresponding to a high probability that the first ranked document will remain the same across perturbations. We computed the First Rank Change in the following way:

a) Run the original query q, and obtain the result list R

b) Take the top 50 documents in R and consider them as ranked list L

c) For each document d in L, get a perturbed document d' from d in the following way:

   - All terms t from the corpus that do not appear in document d, will not be included in d' neither
   - All terms t from the corpus that appear in document d with frequency f, but do not appear in the query will appear in document d' with the same frequency f.
   - Each term t that appears in d with frequency f and appears also in the query q will appear also in d' , but with a frequency f', which is a random number obtained from a Poisson distribution P(λ) with λ = f

d) The new 50 documents obtained are ranked according to the query q, resulting in a second ranked list L', where each document corresponds to a document in L.

e) Record a 1 if the top ranked document in L is also the top ranked document (after perturbation) in L', and record 0 otherwise.

f) Repeat steps c) – e) 100 times, and the final score is the sum of the values (0 or 1) obtained in step e) for all the 100 runs.

Clustering Tendency

Clustering Tendency (CT) is another document perturbation measure, capturing the cohesion of the top retrieved documents as the textual similarity between them. The higher the clustering tendency is, the better the query. We used the following formula to compute the Clustering Tendency:

$$CT = Mean\left(\frac{Sim_{query}\left(d_{mp}, d_{nn}\middle|q\right)}{Sim_{query}\left(p_{sp}, d_{mp}\middle|q\right)}\right) * \frac{1}{T}\sum_{i=1}^{T}(x_i - y_i)$$

Where:

- q is the query
- p_{sp} is the sampled point, i.e., a randomly chosen document from the corpus, which does not appear in the top 100 documents
- d_{mp} is the marked point, i.e., the document, inside the top 100 documents in the ranked list , with largest similarity with the sampled point
- d_{nn} is the nearest neighbor of the marked point within the top 100 documents in the ranked list
- x_i is the maximum weight for a term i across the top 100 retrieved documents
- y_i is the minimum weight for a term i across the top 100 retrieved documents
- The mean in the CT formula is computed for 100 randomly sampled points (i.e., the similarity formulas are computed 100 times, each time with a different randomly sampled point).

To compute the textual similarity between two documents we used the following formula:

$$Sim_{query}(d_i, d_j|q) = \frac{\sum_{K=1}^{T} d_{ik} d_{jk}}{\sqrt{\sum_{k=1}^{T} d_{ik}^2 \sum_{k=1}^{T} d_{jk}^2}} * \frac{\sum_{k=1}^{T} c_k q_k}{\sqrt{\sum_{k=1}^{T} c_k^2 \sum_{k=1}^{T} q_k^2}}$$

Where:

- d_i and d_j are the two documents
- T is the number of unique terms in the collection
- q is the query (with weight q_k for term k) - the weight of a term in the query or a document is its tf-idf
- c is the vector of terms common to both d_i and d_j with weights c_k being the average of d_{ik} and d_{jk}

Spatial Autocorrelation

The *Spatial Autocorrelation* measure replaces the retrieval-scores of each top relevant document with the average of the scores of its most similar documents. The linear correlation between the new scores and the original ones is the spatial autocorrelation of the query. A higher spatial autocorrelation indicates a higher quality of the query. We used the following process to obtain the Spatial Autocorrelation:

a) Run the original query q, and obtain the result list R

b) Take the top 50 documents in R and consider them as ranked list L

c) For each document d in L, compute the cosine similarity between d and the rest of the documents in L, using tf-idf as the weight of the terms in the document vectors.

d) Among the documents in L, select the 5 documents that are most similar to d according to the cosine similarity.

e) Let s be the score of document d in L. Assign a new score to d, which is the average score of the 5 most similar documents to it according to the cosine similarity.

f) Perform the above steps for each document d in L

g) The Pearson correlation between the original scores of the documents in L and the derived scores of those documents represents the index of spatial autocorrelation.

Score Distribution Measures

Score distribution-based methods analyze the similarity between the query and the results, which are used to rank the results of the retrieval. For example, the highest retrieval score (i.e., similarity) and the mean of top scores indicate query quality since, in general, low scores of the top-ranked documents indicate some difficulty in retrieval. We use two score distributions measures, namely *Weighted Information Gain* and *Normalized Query Commitment*.

Weighted Information Gain

The *Weighted Information Gain* (*WIG*) measures the divergence between the mean retrieval score of top-ranked documents and that of the entire corpus. The hypothesis is that the more similar these documents are to the query, with respect to the query similarity exhibited by a general non-relevant document (i.e., the corpus), the more effective the retrieval. The higher the weighted information gain, the better the query.

We used the following formula for computing the Weighted Information Gain:

$$WIG(q) = \frac{1}{k} \sum_{d \in D_q^k} \sum_{t \in q} \lambda(t) log \frac{\Pr(t|d)}{\Pr(t|D)}$$

Where:

- q is the query
- t is a term in the query q
- D is the set of all documents in corpus
- D_q is the set of documents in the result set to query q
- D_q^k is the top k documents in the result list
- k is the number of top documents to consider
- |q| is number of terms in the query q
- $\lambda(t) = \frac{1}{\sqrt{|q|}}$

Normalized Query Commitment

Normalized Query Commitment (*NCQ*), on the other hand, measures the standard deviation of retrieval scores in the top k documents returned in response to query, normalized by the score of the whole collection. The higher NCQ, the higher the quality of the query.

We used the following formula to compute the Normalized Query Commitment:

$$NCQ = \frac{\sqrt{\frac{1}{k}\sum_{d \epsilon D_q^k}(Score(d) - \mu)^2}}{Score(D_q)}$$

Where:

- k is the number of top documents to consider. Best performance was obtained with k=100
- D_q^k is the top k documents from the result list returned in response to query q.
- Score (d) is the score obtained by document d in D_q^k
- D_q is the set of all results returned in response to query q

- Score (D_q) is the sum of the scores of all the documents in the result list returned by the IR technique
- $\mu = \frac{1}{k}\sum_{d \epsilon D_q^k} Score(d)$

3.2.2 Query Quality Prediction for Text Retrieval in Software Engineering

Our approach, QualQ, uses the 28 query quality measures defined in the previous section in order to learn, using a classifier, the properties that indicate the quality of a query, i.e., the relevance of the returned results to the task at hand. QualQ consists of two steps. The first step is training the classifier, which constructs a model of query quality based on rules involving the query quality measures. For example, if queries having a high AvgIDF are often associated with good results, then a rule in the model will be designated to check the AvgIDF of the queries against a threshold; queries for which AvgIDF will be above the threshold will be considered of higher quality than those with an AvgIDF below the threshold. A series of such rules are produced based on the query quality measures. Once such a model is built, the second step can be applied, i.e., predicting the quality of new queries by just computing their quality measures and feeding them into the model. The two steps of QualQ are described in more details in the following subsections.

3.2.2.1 Training the classifier

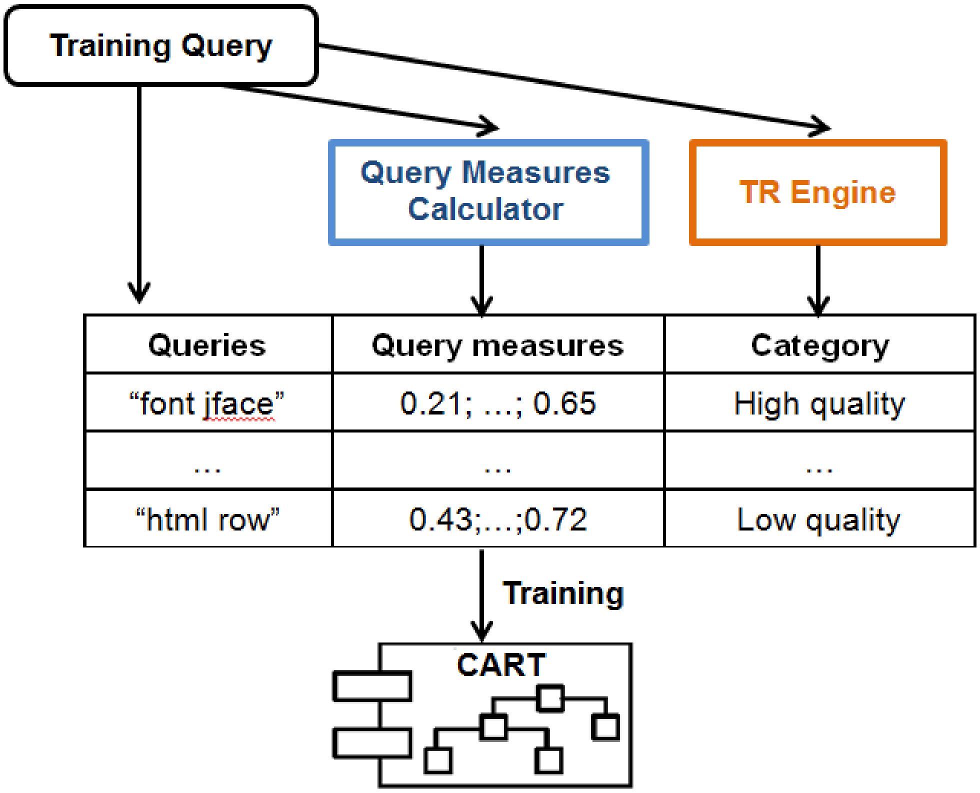
In order to predict the quality of queries, our approach makes use of a classifier, which is first trained (see Figure 3-2) on a set of queries to discriminate among *low-* and *high-quality* queries. To this aim, two things are needed:

a) a set of *training data* consisting of queries and their associated relevant documents.

b) a *classification criterion* defined by the user (i.e., the developer) on how to discriminate between low- and high-quality queries.

These two points strongly depend on the software engineering task addressed. For example, if the task to perform is concept location, the training queries could be automatically extracted from bug tracking systems (i.e., BugZilla) from the text present in bug reports associated with fixed bugs. Using fixed bugs, it is easy to identify the documents relevant to the queries by looking at the source code components (i.e., methods, classes, etc.) changed to solve the bugs. Concerning the criterion defined by the developer to discriminate between low- and high-quality queries, a criterion for the concept location task could be related to the effectiveness measure, setting a maximum acceptable value for it such that the query is considered of high quality. The criterion used in our evaluation study is described in the Section 3.2.3.

Our approach uses the training data and the chosen classification criterion to classify the training queries as low- or high-quality and then learn a model for query quality from the training data using the process described in Figure 3-2. Each training query is first executed using the TR engine and, analyzing the ranked list of retrieved documents, is classified as low- or high-quality according to the chosen classification criterion and the position of the relevant documents in the list of results retrieved by the TR engine. Then, the value of the 28 query measures for each training query is computed. Finally, the classifier is trained using the collected training data. One data point in the final training data used by the classifier corresponds to a query. Each data point has 28 attributes corresponding to the query quality measures and one corresponding to the query classification, i.e., low- or high-quality (see Figure 3-2).



Queries	Query measures	Category
“font jface”	0.21; …; 0.65	High quality
…	…	…
“html row”	0.43;…;0.72	Low quality

Figure 3-2. The training phase of QualQ. The Classification and Regression Tree (CART) is trained based on a set of training queries, the measures of their properties, and their category

As a classifier, QualQ currently uses a *classification and regression tree* (CART) [19] in order to determine the rules that can predict if queries are high- or low-quality. However, QualQ is based on a general model able to accommodate any other classifier instead of CART.

CART is a prediction model that can be represented as a decision tree [19]. Decision tree learning has been previously applied successfully to query performance prediction in NL [51, 122, 129], and also to analyzing software engineering data [71] (i.e., for defect prediction). We chose classification trees for the current implementation of QualQ, as they present several advantages. First, the rules produced by classification trees are easy to understand by humans, which is not true for other, more complex models. Second, classification trees perform implicitly feature selection. This is a very important property, as it allows our approach to be less sensitive to the choice of query property measures. In the current form, it allows us to give as input all 28 measures of a query, as our classification tree will determine automatically the subset of measures relevant for the classification, and will use only those for the classification. Note that CART selected no more than three measures for any data set.

Classification trees are suitable to solve problems where the goal is to determine the values of a categorical variable based on one or more continuous and/or categorical variables. In our approach, the categorical dependent variable is represented by the quality of a particular query (i.e., low- or high-quality), while the independent variables are the 28 query quality measures described in Section 3.2.1. The classifier uses the training data to automatically select the independent variables and their interactions that are most important in determining the dependent variable to be explained. As a reminder, even though QualQ currently makes use of the set of 28 measures presented in Section 3.2.1, it allows for the replacement of these measures with any other set of measures deemed to capture the quality of a query.

There are two possible approaches when training the classifier, namely *within-project* and *cross-project training,* each having advantages and disadvantages. In *within-project training*, the classifier is trained and tested on the same system, and the evaluation is done independently for each software system. In order to ensure the least bias in the evaluation, all data points should be used for training and testing at some point. For this purpose, cross-validation is used, where the evaluation is done in several rounds, such that all data points get to be evaluated exactly once, in one of the rounds. In each round a small part of the data is kept for testing, while the rest is used for training. When performing this kind of validation it is important to select balanced training sets, where there are enough data points to learn from for each possible class and that the number of data points belonging to each class is balanced in the training set. In the case of Refoqus this means that the training sets need to be chosen such they contain approximately equal numbers of data points assigned to each of the reformulation strategies.

The advantage of within-project training is the fact that it could potentially capture properties of the data specific to a software system. On the other hand, it requires additional overhead as the classifier requires retraining for new systems. This can be a problem when little or no training data is available for the new systems. The alternative is *cross-project training*, where, given a set of n systems, the classifier is trained using all data points from n-1 systems and then tested on the data from the n^{th} system, which was not included in the training. This evaluation is repeated n times, each time considering one of the systems for testing and the rest for training.

Cross-project training has the disadvantage, however, that it may miss some project-specific properties of the data, which the within-project training may be able to take advantage of for producing more accurate results. We investigate both approaches in our evaluation, described in Section 3.2.3.

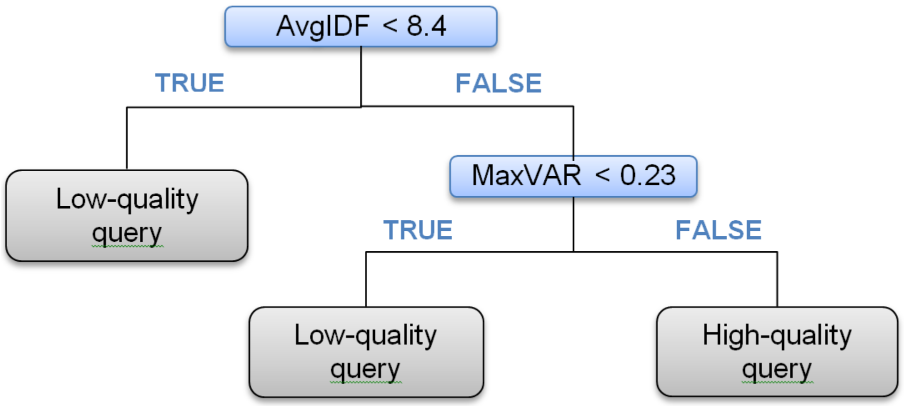


Figure 3-3. Example of Classification and Regression Tree built for a dataset. For this data set, only two measures are considered important for the classification, i.e., AvgIDF, and MaxVAR.

The output of the training stage is the classification tree, represented by a set of yes/no questions that splits the training sample into gradually smaller partitions that group together cohesive sets of data, i.e., those having the same value for the dependent variable. An example of classification tree can be found in Figure 3-3. Note that, when within-project training is performed and the evaluation is based on cross-validation, a different classification tree may be built for each evaluation round during the cross-validation, as the training data used is different in each round.

In the second step of the approach used by QualQ, the classification tree built during the training phase is used to predict the quality of new queries. This step is described in detail in the following section.

3.2.2.2 Using the Classification Tree to Assess the Quality of Queries

Once the classification tree is built, it can be used to automatically assess the quality of a new query. When the new query is issued (manually or automatically) to the TR engine, QualQ computes the 28 query measures for it. Based on the classification

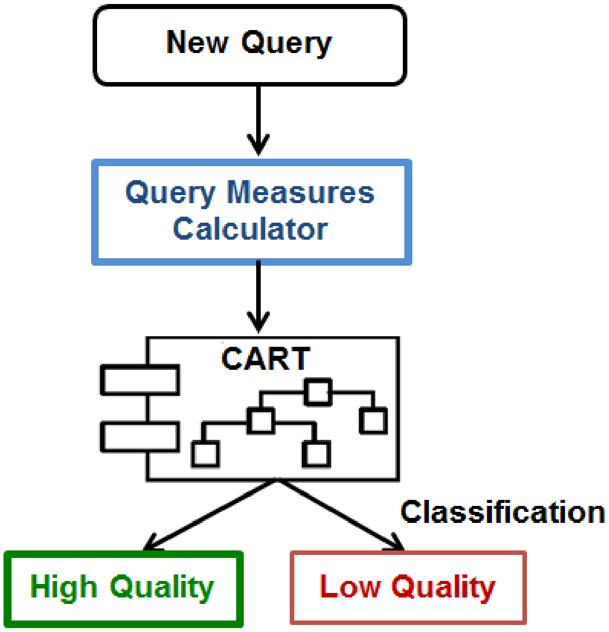


Figure 3-4. The classification phase of QualQ. Based on the query property measures, the Classification and Regression Tree (CART) is classifies a new query as high or low quality.

tree and the quality measures, QualQ automatically classifies the quality of the given query as being low or high. This phase of the approach is depicted in Figure 3-4.

Based on the task to be performed, this information can be useful in different ways. For example, if the task at hand is concept location and QualQ classifies a query written by the developer as a low-quality one, the developer could reformulate the query without spending time analyzing the likely irrelevant results retrieved by the TR engine. In the following section we present an evaluation of QualQ in the context of this task (i.e., concept location in source code).

3.2.3 Evaluation on Concept Location in Source Code

We evaluated QualQ in a study on concept location in source code, as many existing concept location techniques use TR-based solutions [42]. Concept location is an activity performed during software change, concerned with identifying a point in the source code (e.g., a class or a method) where a change needs to be made in order to implement a given change request. TR approaches are used in order to search the code for the point of change using a query formulated based on the change request. More details about concept location and about using TR approaches to address it can be found in Section 2.2.

3.2.3.1 Study Design

The goal of this study is to determine how well QualQ performs when predicting the quality of TR queries in the context of concept location. There are several aspects we want to evaluate. First, we want to establish how well QualQ performs in predicting the quality of queries for concept location. We investigate both types of training for the classifier in QualQ, i.e., *cross*- and *within-project* training and in doing so, we want to learn which training strategy works better for QualQ. Therefore, we formulate the following research question:

RQ1: How accurate is QualQ in predicting the quality of queries for TR-based concept location when using within- and cross-project training?

For the *within-project training*, the classification model was trained on each system individually and a 4-fold cross-validation was performed. The process for the within-project validation is composed of five steps:

a) randomly divide the set of queries for a system into 4 approximately equal subsets

b) 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)

c) classify the queries in the test set using the classification model built on the query training set and store the accuracy of the classification

d) repeat this process, setting aside each query subset in turn

e) 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 quality query as low quality, while a *Type II error* is when the model wrongly classifies a low quality query as high quality.

For the *cross-project training*, given a set of n systems, we use the data from n-1 of the software systems to train the model and then tested it on the remaining, n^{th} system. This model mimics the realistic situation when the queries of a new system need to be evaluated, for which no training data is available.

Analyzing the two types of validation, i.e., within-project and cross-project, can give us an indication of 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, thus reducing the overhead of procuring training data and rebuilding the classification model for each new system.

The second research question refers to the performance of QualQ compared to baseline techniques:

RQ2: Does QualQ perform better than the baseline classifiers?

In the context of our study we compared QualQ based on classification trees with three baseline approaches: 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 quality. 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 quality, while the optimistic constant classifier works by always classifying a query as high quality. It is worth noting that a classifier is useful only if it outperforms a random or constant classifier.

3.2.3.2 Data

In order to collect the queries needed for the study, we used an approach frequently adopted in concept location empirical studies, based on change reenactment and user simulation [65], described also in Section 2.2.1. We collected queries for seven open source object-oriented (OO) systems from different problem domains, implemented in Java and C++, which are summarized in Table 3-8. Adempiere[8] is a common-based peer-production of open source enterprise resource planning applications. ATunes[9] is a full-featured media player and manager. FileZilla[10] is a graphical FTP, FTPS, and SFTP client, JEdit[11] is a programming editor, Mahout[12] is a machine learning and data mining library, Eclipse[13] is a popular integrated development environment for Java, and WinMerge[14] is a document differencing and merging tool.

The online bug tracking systems of each of the seven systems were consulted and a set of closed bug fix requests were identified. The selected bug reports correspond to bugs that are present in the version of the software system used in our study, but fixed

[8] http://www.adempiere.org
[9] http://www.atunes.org
[10] http://www.filezilla-project.org
[11] http://www.jedit.org
[12] http://mahout.apache.org/
[13] http://www.eclipse.org/
[14] http://www.winmerge.org

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 bug tracking systems. This set of methods represents the oracle for concept location. We will refer to these methods as the *target methods*.

Table 3-8. The Systems Used in the Study and their Properties

System	Version	Language	KLOC	#Methods	#Queries
Adempiere	3.1.0	Java	330	28,355	51
ATunes	1.10.0	Java	80	3,481	51
Eclipse	2.0	Java	2,500	76,335	51
FileZilla	3.0.0	C++	240	3,240	87
JEdit	4.2	Java	250	5,532	54
Mahout	0.4	Java	110	15,338	54
WinMerge	2.12.2	C++	410	8,012	69
Total	**-**	**-**	**3,920**	**140,293**	**417**

For each change request, we collected three queries, two automatically extracted from the online bug tracking systems and one manually formulated by one of the authors. While automatically extracting queries from bug tracking systems is a de-facto practice in software engineering, it still lacks resemblance to actual human queries. We included the third, manually formulated query to mitigate this aspect. The first extracted query was obtained from the title of the bug report (i.e., the short description), while the second extracted query represented 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. 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). Table 3-8 reports the number of queries we selected for each system.

For example, from Bug #1605980 of Adempiere, we obtained the following three queries after extraction and normalization (in parenthesis is the original text extracted from the bug reports, before the normalization):

1. From bug title: *invoic process draft select*

   (Original title: Print Invoices process - draft and selection)

2. From bug description: *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*

   (Original description: 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!)

3. From developer query: *print invoic select draft email*

   (Original developer query: print invoice selection draft email)

While fixing this bug, the target method changed by the developers was *doIt()*, found in the 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 TR-based concept location.

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 the same normalization approaches used for the queries: identifier splitting, stop words removal, and stemming. The chosen TR technique was Lucene[15], which a popular and improved implementation of the Vector Space Model. Lucene was used to index the source code corpus and to search the source code using the defined queries.

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. Therefore, if any of the target methods ranks in among the top retrieved results, we consider it a successful retrieval. Other methods that change are usually 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 often 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 *high quality* if any of the target methods is retrieved in the top 20 results. Otherwise, we consider the query as having *low quality*. We classify in this way all the queries used in our evaluation. In the above example, if a query returns the target method *doIt()* in top 20, then it is considered of high quality.

Knowing the target methods beforehand (from the submitted patches) allowed us to categorize all the queries used in the study following the same procedure. Table 3-9

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

shows the actual number of high and low quality queries for each system used in our study. Note that this reflects the actual quality of the queries, not the predicted one.

Table 3-9. The actual quality of the queries used in the study.

System	Total # queries	# High quality queries	# Low quality queries
Adempiere	51	23	28
ATunes	51	20	31
Eclipse	51	15	36
FileZilla	87	19	68
JEdit	54	24	30
Mahout	54	25	29
WinMerge	69	23	46
All	**417**	**149**	**268**

One interesting observation is that, while for some systems the numbers of high and low quality queries are more or less balanced, for others, such as, FileZilla, Eclipse, and WinMerge the number of low quality queries is two to four times higher than the number of high quality queries. This means that, when TR techniques are applied for concept location in these systems, it is likely that developers will have to investigate more than 20 irrelevant artifacts before finding a relevant one. This underlines the need for query quality prediction, which would prevent developers from investigating the results of such low quality queries and would prompt them to reformulate the query instead.

Having such a difference in the number of high and low quality queries can also represent a challenge for our approach, as QualQ would have significantly less examples to learn from for determining the properties that high quality queries share. However, QualQ is still able to obtain good results for these cases, as shown in the following section.

3.2.3.3 Results and Discussion

Research Question 1

We first compared the accuracy of QualQ using within- and cross-project training, as we were interested to know which training procedure is more appropriate, and also to know if cross-project training is feasible to use when training data is not available for new systems. Table 3-10 shows the results for the within- and cross-project training for each system and overall for all seven systems in the study, including the misclassification rates for each system and error type.

In the cross-project training scenario, QualQ obtains a much lower accuracy on average (60%) than the within-project training (85%). This difference in accuracy, favoring the within-project training is observable also for each individual system, where the difference in accuracy ranges between 15% in the case of FileZilla, and 33% in the case of Adempiere and jEdit, in favor of the within-project training.

Table 3-10. The Accuracy and Error Rates of QualQ for Within- and Cross-Project Training

System	**Within-project training**			**Cross-project training**		
	Correct	**Type I**	**Type II**	**Correct**	**Type I**	**Type II**
Adempiere	90%	10%	0%	57%	25%	18%
ATunes	88%	8%	4%	59%	29%	12%
Eclipse	90%	10%	0%	65%	25%	10%
FileZilla	89%	7%	4%	74%	22%	4%
JEdit	83%	9%	8%	50%	43%	7%
Mahout	82%	10%	7%	56%	37%	7%
WinMerge	75%	15%	10%	57%	32%	11%
Average	**85%**	**10%**	**5%**	**60%**	**30%**	**10%**

Regarding the misclassifications, within-system training obtains on average three times less Type I misclassification errors and half the Type II misclassifications compared to cross-system training. It is worth noting that the high accuracy (85%) achieved by QualQ using within-project training was obtained using very small training samples, as the average size of a training set is 45 queries per system. Such results emphasize the applicability of QualQ using within-project training. Based on these results, we can conclude that within-project training is superior compared to cross-project training and it should be sought whenever training data for a new system is available, even if in small amounts.

For both within- and cross-project training, however, the number of Type I errors is considerably higher than the number of Type II errors. When trying to explain this observation, we noticed that for each system, and overall for all systems there are less high quality queries in the training data than low quality queries (see Table 3-9) and in some systems this difference is considerable. For the within-system training, on average, there are 16 high quality queries to learn from per system in each evaluation round, compared to 29 low quality queries on average per system. This may mean that QualQ can learn better the patterns characterizing low quality queries than those for high quality queries. This may lead to more high quality queries being misclassified than the low quality queries, and may therefore explain the difference in Type I and Type II errors observed in each of the software systems and overall in the average, as well as the occurrence of this type of error in the first place.

In the remainder of the section, due to the superior results, we focus the analysis of the results on QualQ using the within-project training.

RQ1 answer. Within-project training is superior to cross-project training. Within-project training obtains very good results (85% accuracy on average) with little training data.

Research Question 2

Table 3-11 shows the accuracy (i.e., percentage of correctly classified queries) of QualQ and the baselines for the seven software systems. QualQ is the most accurate predictor on all the systems and overall, when considering the average across all systems. As mentioned before, QualQ obtains a correct classification rate of 85% on average. In comparison, the optimistic constant model obtains a correct classification rate of only 37%, the pessimistic constant model has a correct classification rate of 63%, and the random model correctly classifies the queries in only 52% of the cases. This indicates that QualQ outperforms all the baseline classifiers.

Table 3-11. The Accuracy (Correct Classifications) of QualQ and the Baseline Classifiers

System	QualQ	Optimistic Constant	Pessimistic Constant	Random
Adempiere	90%	45%	55%	41%
ATunes	88%	39%	61%	59%
Eclipse	90%	29%	71%	61%
FileZilla	89%	22%	78%	57%
JEdit	83%	44%	56%	50%
Mahout	82%	46%	54%	46%
WinMerge	75%	33%	67%	49%
Average	**85%**	**37%**	**63%**	**52%**

In addition to the improved results obtained by QualQ over the baseline approaches, these 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 queries as high or low quality between 62% [51] and 74% [122] of the times on average.

We further investigated the results by analyzing the number of misclassifications obtained by QualQ and the baselines. Table 3-12 shows the number of Type I and Type II misclassifications. The total percentage of errors obtained by QualQ is 15%, with (10% Type I + 5% Type II). This is much lower compared to 63% for the optimistic constant classifier, 36% of the pessimistic constant and 49% of the random classifier.

Table 3-12. The Percentage of Incorrect Classifications for Within-Project Training, by Error Type

System	**QualQ**		**Optimistic**		**Pessimistic**		**Random**	
	Type I	**Type II**	**Type I**	**Type II**	**Type I**	**Type II**	**Type I**	**Type II**
Adempiere	10%	0%	0%	55%	45%	0%	33%	26%
ATunes	8%	4%	0%	61%	39%	0%	31%	10%
Eclipse	10%	0%	0%	71%	20%	0%	25%	14%
FileZilla	7%	4%	0%	78%	22%	0%	7%	36%
JEdit	9%	8%	0%	56%	44%	0%	43%	7%
Mahout	10%	7%	0%	54%	46%	0%	6%	48%
WinMerge	15%	10%	0%	67%	33%	0%	20%	31%
Average	**10**%	**5**%	**0**%	**63**%	**36**%	**0**%	**24**%	**25**%

Among the baseline classifiers, the pessimistic classifier performs the best on average (63% accuracy), as well as for each system in particular. This is explained by the fact that there is a considerably higher number of low quality queries compared to high quality queries in the data sets used in the study (see Table 3-9). All these low quality queries are therefore classified correctly by the pessimistic classifier, which leads to its higher accuracy compared to the rest of the baseline approaches.

RQ2 answer. QualQ outperforms all baselines, and its accuracy also surpasses that of approaches proposed in natural language document retrieval.

Analysis of Misclassifications

The higher occurrence of Type I errors (i.e., a high quality query is classified as a low quality one) compared to Type II (low quality query classified as high quality) can be explained by the presence of a lower number of high quality queries. This smaller number of high quality queries may be insufficient to learn all the patterns high quality queries follow, leading to the misclassifications. There is, however, no apparent indication of the reasons behind Type II errors. Therefore, we analyzed some examples in order to understand the reasons behind the poor accuracy of QualQ in these cases.

QualQ obtained the most misclassifications of Type II in the case of WinMerge, i.e., 10% of classifications were Type II errors. We investigated some Type II misclassification examples from this software system. We observed that in most cases where a low quality query was misclassified as high quality, it contained very few or none of the terms found in the target methods. Therefore, these queries often do not retrieve any of the target methods in the list of results, or retrieve them on a very high position in the ranked list of results. This is one of the challenges caused by using bug descriptions and change data for such studies as, in some cases, the bug descriptions capture the observed behavior of the system, whereas the bug is fixed in a part of a code with different vocabulary characteristics. This leads to vocabulary mismatch between the bug reports and the target methods and leads to Type II misclassifications, where queries fail to retrieve relevant results, therefore being low quality, even though the quality metrics, based on statistics (and therefore with no knowledge of the meaning) may mark them as high quality.

In order to verify our assumption, we also checked the Type II misclassifications in another system, i.e., FileZilla. We observed the same phenomenon, i.e., most misclassified low quality queries contained no or few terms found in the target methods. In these cases, other approaches may be needed in order to complement TR during concept location, such as, static or dynamic analysis. In order to determine this type of cases, however, QualQ would need to be complemented with measures capturing the semantic information contained in queries. We leave this for future work, beyond the scope of this dissertation.

3.2.3.4 Threats to Validity

This section discusses the main threats to validity [128] that could affect our results.

Construct validity threats concern the relationship between theory and observation. We evaluated the accuracy of the proposed approach by observing the number of Type I and Type II errors. These measures are widely used in software engineering to evaluate predictor models [2]. 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.

With respect to the *internal validity*, in our experimentation for concept location we automatically extracted the set of queries from the online bug tracking system of the object systems. In particular, we extracted two different queries from the bug reports, one derived from the title of the bug report and one from the description of the bug. Such queries are approximations of actual user queries. However, we also included a manually formulated query to address the threat to validity introduced by the automatically constructed queries.

The *external validity* refers to the generalization of our findings. In order to address this threat, we selected a set of seven software systems from diverse domains, implemented in two programming languages for our concept location study. 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-quality or low-quality. More precisely, we used Lucene, which is an implementation of the VSM technique. Since several other TR methods have been previously used to support concept location, further experimentation is needed to analyze whether the proposed approach 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 software engineering tasks.

Finally, *conclusion validity* refers to the degree to which conclusions reached about relationships between variables are justified. In our 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.

3.3 Related Work

3.3.1 Query Quality Analysis in Software Engineering

There is currently no other work in software engineering analyzing the quality of text retrieval queries.

The closest work in the field includes studies that have looked at the results of formulating different queries for the same information need [73, 87, 95, 119]. The drastic differences between the results returned by the different queries highlight the strong dependence of the retrieval performance on the query and motivate our work.

3.3.2 Query Quality Analysis in Natural Language Document Retrieval

Query quality was first studied in the Text Retrieval community and emerged from the need to explain the high variance in performance across different queries observed during the Text REtrieval Conference (TREC) competitions. A special track was created for this purpose at the TREC conference between 2003-2005, i.e., the Robust track [123]. The participants were first encouraged to decrease the variance of their TR engines across the range of queries, by focusing on improving the performance on queries known to be hard to answer (i.e., having low quality). This was followed in 2004 and 2005 by an additional challenge of predicting the performance of the TR engines on each individual query, i.e., to predicting the quality of each query. The query quality predictions 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. The outcome of the Robust track indicated that predicting the quality of queries is a challenging problem, and sprouted the research on this topic. Since then, numerous approaches for assessing the quality of TR queries have been proposed in the NL document retrieval field [23], but the main goal has remained the same: predicting the AP of a query based on measures that correlate with it.

Our main goal is different than most of the work on query quality existent in the field of NL document retrieval. While having a correlation between the quality of a query and the AP is interesting, it has little practical application as the end goal. On the other hand, assigning a quality label to each query, i.e., high or low, enables the software engineer to use this information as a recommendation of whether the results returned by a TR approach are worth investigating or not. If the query is estimated to be of low quality, it is likely that the results returned in response to that query are not satisfactory, and thus, investigating them could lead to time and effort wasted. If the query is of high quality, on the other hand, the software engineer is likely to find useful information among the top retrieved results.

A few papers in the field of NL document retrieval have also investigated the query quality prediction problem from the perspective of classifying incoming queries into easy to answer (high-quality) and hard to answer (low-quality) queries [51, 122, 129]. 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 is of low- of high-quality, our work is different in several ways from the work in NL document retrieval.

First of all, we use a different set of quality measures for training the classifiers. Many of the measures used by the work in NL retrieval are relying on the fact that the query and the documents are written in natural language, specifically English, and make use of the rules that govern this language. On the other hand, software artifacts and the queries used to search them commonly contain terms and constructions that are not correct English and therefore do not adhere to those rules. We carefully selected a set of 28 pre-retrieval (collected before the query is executed) and post-retrieval (collected after the query is executed) query quality measures which do not rely on English rules and are suited to be applied to software engineering artifacts.

The second aspect of our work that is different is the context in which the query quality approaches are used. This is the first time query quality prediction has been addressed in the context of software engineering tasks. Accurately predicting the quality of text retrieval queries can have a significant impact on the multitude of tasks supported by such approaches.

CHAPTER 4 QUERY REFORMULATION SUPPORT FOR TEXT RETRIEVAL IN SOFTWARE ENGINEERING

When the results returned by TR in response to a query are not relevant (i.e., the query is of low quality), the query is usually reformulated by adding or removing words. Rewriting a query in order to improve its quality and retrieve the relevant documents closer to the top of the list of results is often as difficult as writing the query in the first place. This is due to several reasons. First, the developers writing the queries are often not the same as the ones that wrote the software artifacts being searched, which can lead to mismatch between the vocabulary used in the query and the one of the searched artifacts, leading to failed searches. Another problem is the fact that it is hard for developers to understand what was wrong with the initial query and how to improve it. This is often due to the fact that TR techniques use complicated mathematical models which are hard to grasp in enough detail to understand how queries should be written or improved. Also, studies [119] have shown that developers often have a hard time reformulating and improving queries even after several tries.

This problem has been recognized by software engineering researchers and two types of approaches have been proposed to assist developers with the query reformulation. The first category of approaches is based on user relevance feedback and it has been employed in the context of traceability link recovery [38]. In this chapter we investigate the application of relevance feedback in the context of a different task, namely concept location (Section 4.1).

The second category of approaches are completely automatic, but employ the same reformulation strategy for all queries [27, 50, 61, 87, 117, 127]. We argue that different queries may require different reformulation approaches and introduce a novel approach which differentiates between queries based on their lexical properties and selects the best reformulation approach for each query individually based on these properties (Section 4.2).

4.1 Semi-Automatic Query Reformulation for Text Retrieval in Software Engineering

Text Retrieval techniques have some limitations when applied to software engineering tasks:

1. all TR approaches are highly sensitive to the ability of the user to write good queries;
2. the knowledge gained by the user while using the TR approaches is not captured explicitly.

Developers start the process of using TR techniques from a description of the task or a software artifact and they either use the entire description or artifact as a query or they select a set of words from it and use this subset as a query. How the developers formulate the queries depends on their experience and on their knowledge of the system. Previous work showed that developers tend to write queries with significantly different performance starting from the same change request [73]. As the developer investigates the results of the first search, she learns more about the system and can eventually decide to improve the query by adding or removing words. However, there is a gap between the source code representation as classes and methods (or other decomposition units) and the words in a query and some developers can fill this gap easier than others.

Relevance feedback is a semi-automatic technique which aims at addressing these limitations. It captures the developer knowledge by utilizing user input in order to automatically reformulate TR queries. Relevance feedback has been one of the successes of information retrieval research for the past 30 years [81]. For example, the Text Retrieval Conference (co-sponsored by the National Institute of Standards and Technology - NIST and the U.S. Department of Defense) has a relevance feedback track. While the applications of relevance feedback and the type of user input to relevance feedback have changed over the years, the actual algorithms have not changed much. Most algorithms are either pure statistical word based, or are domain dependent. There is no general agreement of what the best RF approach is, or what the relative benefits and costs of the various approaches are. In part, that is because RF is hard to study, evaluate, and compare. It is difficult to separate out the effects of an initial retrieval run, the decision procedure to determine what documents will be looked at, the user dependent relevance judgment procedure, and the actual RF reformulation algorithm.

There are three types of feedback: explicit, implicit, and blind ("pseudo") feedback. In our approach, we chose to implement an explicit RF mechanism. Explicit feedback is obtained from users by having them indicate the relevance of a document retrieved for a query. Users may indicate relevance explicitly using a binary or graded relevance system. Binary relevance feedback indicates that a document is either relevant or irrelevant for a given query. Graded relevance feedback indicates the relevance of a document to a query on a scale using numbers, letters, or descriptions (such as "not relevant", "somewhat relevant", "relevant", or "very relevant").

Classic text retrieval applications of RF make several assumptions [81], which are not always true in the case of source code text and make the problem more challenging:

- *The user has sufficient knowledge to formulate the initial query*. This is not always the case when it comes to software, as developers might be unfamiliar with a software system or they might not have enough knowledge about a particular problem domain.
- *There are patterns of term distribution in the relevant vs. non-relevant documents*: (i) term distribution in relevant documents will be similar; (ii) term distribution in non-relevant documents will be different from that in relevant documents (i.e., similarities between relevant and non-relevant documents are small). There is no evidence so far if this is true for source code.

RF has also some known limitations, which approaches in software engineering also face:

- It is often harder to understand why a particular document was retrieved after applying relevance feedback.
- It is easy to decrease effectiveness (i.e., one irrelevant word can undo the good caused by many good words).
- Long queries are inefficient for a typical IR engine. In the case where the queries represent software artifacts, it is easy to end up with long queries.

In software engineering, RF has been previously used for traceability link recovery [38, 57]. In the context of software engineering tasks, the developers especially benefitting from TR use are those that are not familiar with the software, which means they may not be able to reformulate the query in a way that matches relevant documents. Relevance feedback relieves the burden of reformulating queries from developers' shoulders and allows them to focus on the software artifacts rather than on the query. Rather, relevance feedback requires developers to analyze the results of TR and judge if the artifacts are relevant or not to the task at hand. Using this information, relevance feedback mechanisms can then reformulate the queries without developer involvement.

4.1.1 Rocchio-based Relevance Feedback for Software Engineering

Using relevance feedback with TR-based approaches for software engineering changes the process described in Section 2 by modifying the last step, i.e., results examination. When using the relevance feedback mechanism, the developer examines the top N documents in the ranked list of results and for every software document (i.e., class, method, requirement, test case, etc.) examined, makes a decision on whether the document satisfies the information need for the current task or not. If the document satisfies the information needed, then the search succeeded and the process ends. Else, the user marks the document as *relevant* or *irrelevant* for the task at hand, based on the information it contains. A relevant document will contain useful information, on the topic of the current task, while it is still not the final target of the search. After the N documents are marked a new query is automatically formulated and the TR engine uses the new query to search the code. The process is iterative and can be repeated several times. If several rounds of feedback do not result in reaching the wanted documents, the query may still be reformulated manually by the user.

There are several options to implement a relevance feedback mechanism. One of the most popular approaches is the *Rocchio* relevance feedback method [111], used in conjunction with a Vector Space Model (VSM) [112] indexing technique. Rocchio has been previously used in the context of traceability link recovery [38, 57] in software engineering.

The Rocchio algorithm bases the reformulation of the query on the formula described below. Given a set of documents D_Q encompassed by query Q, let R_Q be the subset of relevant documents and I_Q the set of irrelevant documents to the query. The original query Q can be then transformed by adding terms from R_Q and removing terms from I_Q. This mechanism is meant to bring the query closer to the relevant documents and drive it away from the irrelevant documents in the vector space. The new query is formulated as follows:

$$Q' = \alpha Q + \frac{\beta}{|R_Q|}\sum_{d \in R_Q} d - \frac{\chi}{|I_Q|}\sum_{d \in I_Q} d$$

Where:

- *Q'* is the new, reformulated query
- *Q* is the initial query (the query before reformulation)
- *α* is the weighting parameter for the terms in the initial query. It represents the boost or importance (in the reformulated query Q') given to the terms in the original query Q
- *β* is the weighting parameter for the terms in the relevant documents. It represents the boost or importance (in the reformulated query Q') added to the terms which appear in the documents marked as relevant by the users
- *χ* is the weighting parameter for the terms in the irrelevant documents. It represents the penalization (in the reformulated query Q') of the terms which appear in the documents marked as irrelevant by the users

- *d* represents a document and its associated vector in VSM

The relevance feedback can be given by the user in several feedback rounds and the query is updated after each round based on the query generated in the previous round and the terms in the documents marked as relevant or irrelevant by the user. The three constants α, β, and χ are provided so that a level of importance can be specified by the user for the initial query, the relevant documents and the irrelevant documents.

4.1.2 Evaluation on Concept Location in Source Code

We performed an evaluation of Rocchio in an empirical study in the context of concept location in source code.

4.1.2.1 Study Design

The goal of the study was to address the following research question:

RQ: Does Rocchio improve the results of TR-based concept location?

To answer it, we compared the results of TR-based concept location with and without using Rocchio, given a set of change requests. The study consists of the reenactment of past changes in open source software (i.e., we know which methods were modified in response to the change request). The modified methods form the *change set*, and we call these methods *target methods*. This methodology has been used in previous work on evaluating concept location techniques [73, 76, 99].

We implemented our own version of Rocchio, which integrates with Apache Lucene[16], a commonly used and improved implementation of VSM. For setting the weighting parameter values, De Lucia et al. [38] advocate using α=1, β=0.75, and χ=0.25 (i.e., relevant documents are three times more important than irrelevant ones).

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

Our implementation follows a similar line of thought, setting values of α=1, β=0.5, and χ=0.15 for the three weighting parameters. We tried other sets of weights (α=1, β=0.75, and χ=0.25 and α=1, β=1, and χ=1), but the final choice of weights yielded the best results. In the rest of this section, we refer to the implementation using the chosen set of weights.

Query reformulation using relevance feedback is prone to noise, as many terms can be added to the query in just one round of feedback. To filter noise and prevent common but unimportant terms to be included in the query, the system used in this paper only allows terms to be added to the query if they appear in less than 25% of the documents in the corpus.

When analyzing the top ranked methods, a user is asked to judge the current method as relevant, irrelevant, or neutral to the current change task. For our study on concept location, one developer provided the feedback for Rocchio. He has seven years of programming experience (five in Java) and was not familiar with the source code used in the study. For each change request his task was to locate one of the methods from the change set, based on the following scenario:

a) He starts by running a query based on the change request, called the *initial query*.
b) If any one of the target methods is among the top 5 methods in the ranked list of results, then he stops and selects another change request, as Rocchio is not needed in this case (i.e., IR-based concept location will reach the method fast enough). We considered 5 as the threshold here as we selected this as the

maximum number of methods the developer needs to evaluate in a round of feedback before reformulating the queries.

c) Else he provides Rocchio in several rounds. In each round, the developer marks the N top ranked methods as being *relevant* or *irrelevant*. If he cannot judge the relevancy of a method, the he marks the document as *neutral* and proceeds to the next document, increasing the size of the set of marked methods set by one.

d) Rocchio automatically reformulates the query based on the feedback provided by the developer, runs the new query, and a new round of feedback begins. We keep track of the number of methods marked by the developer.

e) After each query is run, based on the positions of the target methods in the ranked list of search results and on the number of methods marked, the following decisions are made:

    - If any of the target methods is located in the top N documents, then STOP; consider Rocchio successful and a target method found.
    - If for two consecutive feedback rounds the positions of the target methods declined in the ranked list of results, then STOP; consider that Rocchio failed (i.e., the developer needs to reformulate the query manually).
    - If more than 50 methods were marked by the developer, then STOP; consider that Rocchio failed (i.e., the developer needs to reformulate the query manually).

The values used for *N* vary and the performance of Rocchio depends on it. The most commonly used values in literature range from 1 to 10. We investigated the results of Rocchio for three values of *N*: 1, 3, and 5, which are recommended values in recent studies for presenting lists of results to developers for investigation [110]. Each reenactment was done three times by the developer, the difference from case to case was the number *N* of marked methods in one round.

4.1.2.2 Data

We chose as the objects of the study three open source systems: Eclipse[17] 2.0, JEdit[18] 4.2, and Adempiere[19] 3.1.0. Eclipse is an integrated development environment developed in Java. For our study, we considered version 2.0 of the system, which has approximately 2.5 millions lines of code and 7,500 classes. JEdit is an editor developed for programmers and it comes with a series of plugins which add extra features to its core functionality. It is developed in Java and version 4.2 used in this study has approximately 300,000 lines of code and 750 classes. Adempiere is a commons-based peer-production of open source enterprise resource planning applications. It is developed in Java and it has approximately 330,000 lines of code and 1,900 classes in version 3.1.0, which was used in our study.

All three systems have an active community and a rich history of changes. They all have online bug tracking systems, where bugs are reported and patches are submitted for review. We used the history of a software system, i.e., approved patches of documented bugs, in order to extract real change requests and their corresponding change sets. The bug descriptions are considered to be the change requests. This approach, based on reenactment, has been used in previous work on evaluating

[17] http://www.eclipse.org/
[18] http://www.jedit.org/
[19] http://www.adempiere.org/

concept location techniques [73, 76, 99]. Some changes involve the addition of new methods. We do not, however, include these methods in the change sets, as they did not exist in the version that a developer would need to investigate in order to find the place to implement the change. For each of the systems, we analyzed their online defect tracking systems and manually selected a set of ten bugs to extract change sets for our study.

The Eclipse community uses the open-source bug tracking system BugZilla[20] to keep track of bugs in the system. Each bug has an associated bug report, which consists of several sections, one of which is the bug description. Sometimes the patches used to fix the bugs are also contained in the bug report, as attachments. They are usually in the form of *diff* files, containing the lines of code that changed between the version of the software where the bug was reported and the version where the bug was fixed. For our study, we chose an initial set of ten Eclipse bugs reported in version 2.0 of the system, for which the patches were available in their bug reports.

For jEdit[21] and Adempiere[22], we analyzed the bug tracking systems hosted on the projects' sourceforge.net website. Both projects have systems that keep track of the patches submitted for known bugs in the source code. In these trackers, each patch has an associated report where the changes implemented in the patch are described in a *diff* file attached to the report. We selected for each system ten initial patches for which a good description of the bug fixed by the patch was available, either in the description of the patch or in a separate bug report. All the patches we selected for

[20] https://bugs.eclipse.org/bugs/
[21] http://sourceforge.net/tracker/?group_id=588&atid=300588
[22] http://sourceforge.net/tracker/?atid=879334&group_id=176962

jEdit were submitted and their corresponding bugs reported after version 4.2 of the system was released. For Adempiere, the patches selected were after the release of version 3.1.0.

Based on the patches reported for the three systems, we constructed the 10 change sets for each system. All change sets contained between one and six target methods.

We extracted a corpus for each of the three systems. We used the version of the software in which the bugs chosen in the previous step were reported. We mapped each method in the source code to a document in our corpus. The Eclipse corpus has 74,996 documents, the JEdit corpus has 5,366 documents, and the Adempiere corpus has 28,622 documents. By comparison, the size of the corpora used in previous work on Rocchio in traceability [38, 57] is in the few hundreds of documents range.

The corpora were built in the following manner:

a) We extracted the methods using the Eclipse built in parser. Then, the comments and identifiers from each method implementation were extracted.

b) The identifiers were split according to common naming conventions. For example, “setValue”, “set_value”, “SETvalue”, etc. are all split to “set” and “value”. We kept the original identifiers in the corpus, which would favor any query containing an identifier already known by the user.

c) We filtered out programming language specific keywords, as well as common English stop words[23].

[23] www.dcs.gla.ac.uk/idom/ir_resources/linguistic_utils/stop_words

d) We used the Porter stemmer[24] in order to map different forms of the same lexeme to a common root.

As mentioned before, the goal of relevance feedback is to allow the developer not to write manually defined queries. Hence, in the study the developer used as the initial query the bug description and bug title contained in the bug or patch reports (i.e., he copied the bug description and title). However, prior to the study we eliminated any details referring to the implementation of the bug fix contained in these descriptions. The query was then automatically transformed following the same steps as the corpus (i.e., identifier splitting, stop word removal, stemming).

Tool support in concept location is geared towards reducing developers' effort in finding the starting point of a change. Previous work on concept location [73, 76, 99] defined and used as an assessment measure the number of source code documents that the user has to investigate before locating the point of change, called *effectiveness*. We use here the same measure with an added advantage. In previous work the cost for formulating and reformulating a query was never considered in evaluation (i.e., assumed to be zero). In our case, the cost of formulating and reformulating a query is indeed almost zero, as the initial query is copied from the bug description and title, whereas subsequent queries are formulated automatically. The number of methods investigated is automatically tracked as they are explicitly marked by the developer.

For each change request, the baseline is provided by the IR-based concept location without query reformulation. The initial query is run and the baseline effectiveness measure is the highest rank (*k*) of any of the target methods. This means the user

[24] http://tartarus.org/~martin/PorterStemmer/

would have to investigate *k* methods to reach the target. For the Rocchio case the effectiveness measure is the number of methods marked one way or another (i.e., relevant, irrelevant, or neutral) before the target was found or until Rocchio fails (see the methodology described above) plus the last rank of the target method in the results list. Rocchio is considered to improve the baseline if its effectiveness measure is lower than that of the baseline (i.e., fewer methods are investigated).

4.1.2.3 Results and Discussion

We selected 10 changes for each system (i.e., 30 in total). In 12 cases, at least one of the target methods was ranked in top 5 after the initial query, hence we did not use Rocchio in those cases. Therefore, our analysis focused on the remaining changes: 7 for Eclipse, 6 for jEdit, and 5 for Adempiere.

Quantitative Evaluation

Table 4-1 shows the quantitative results obtained by the Baseline and Rocchio with the N values of 1, 3 and 5. The Baseline column shows the positions of the target methods in the result list when the initial query was run. The best rank in each case where there is more than one target method is marked in bold. This represents the effectiveness measure in the baseline case (i.e., how many methods would the user need to investigate to find the best ranked target).

The Rocchio columns show the positions of the target methods at the end of the relevance feedback location process, whether it succeeded or not. If a target method was not ranked in the top 1,000 results at the end of Rocchio, we denote its position as 1K+. The N in the column header indicates the number of marked methods in each feedback round during Rocchio. Note that only the methods for which relevance was

Table 4-1. Concept location results for Eclipse, jEdit and Adempiere

Eclipse					
No.	**Defect #**	**Baseline**	**Rocchio with N=1**	**Rocchio with N=3**	**Rocchio with N=5**
1	13926	**54**	1 (**16**m/15r)	11(51m/16r)- *50m+*	36 (50m/10r) - *50m+*
2	23140	**17**,42,47	99, **1**, 2 (**9**m/8r)	4, **1**, 2 (**7**m/3r)	6, **4**, 14 (**9**m/2r)
3	19691	1K+,368,531, K+, **108**, 139	1K+, 1K+, 1K+, 1K+, 1K+, 1K+ (2m/2r) - *NI*	1K+, 1K+, 1K+, 1K+, 1K+, 1K+ (7m/2r) - *NI*	1K+, 1K+, 1K+, 1K+, 1K+, 1K+ (11m/2r)- *NI*
4	12118	**9**	1 (**5**m/5r)	1 (23m/8r)	4 (10m/2r)
5	17707	**8**	1 (**2**m/2r)	1 (**4**m/2r)	2 (**7**m/2r)
6	19686	**428**	448 (5m/5r) - *NI*	3 (**48**m/16r)	5 (**46**m/9r)
7	21062	583,**56**	1K+, 781 (2m/2r) - *NI*	604, **1** (**37**m/13r)	1K+, 1K+ (20m/4r) - *NI*
jEdit					
1	1649033	40,87,**22**	70,60,50(8m/7r) - *NI*	39,7,42 (22m/7r) - *NI*	30**,** **5**, 33 (**26**m/5r)
2	1469996	**296**	1 (**37**m/36r)	289 (12m/4r) - *NI*	5 (**41**m/9r)
3	1593900	**7**	1 (**6**m/4r)	1 (**5**m/2r)	1 (7m/2r)*
4	1601830	**47**	216 (2m/2r) - *NI*	242 (9m/3r) - *NI*	146 (10m/2r) - *NI*
5	1607211	**354**	98 (5m/5r) - *NI*	3 (**36**m/12r)	3 (**28**m/6r)
6	1275607	**151**	238 (4m/4r) - *NI*	38 (48m/16r) - *NI*	35 (50m/10r) - *50m+*
Adempiere					
1	1605419	**15**,550	**1**, 11 (**8**m/7r)	**3**, 109 (**17**m/5r)	**1**, 81 (**12**m/3r)
2	1599107	**122**	613 (6m/3r) - *NI*	1K+ (8m/2r) - *NI*	1K+ (12m/2r) - *NI*
3	1599116	**7**	1 (**3**m/2r)	1 (**5**m/2r)	1 (7m/2r)*
4	1612136	**58**	141 (4m/3r) - *NI*	1 (**13**m/5r)	1 (**16**m/4r)
5	1628050	**52**	1 (**3**m/3r)	2 (**5**m/2r)	2 (**7**m/2r)

Rocchio retrieves results more efficiently	Rocchio retrieves a better cumulative ranking of the target methods.

* Rocchio performs as efficiently as the baseline
NI: no improvement for 2 consecutive rounds;
50m+: 50+ methods need to be analyzed to reach a target method

given are counted as one of the N methods ranked in a feedback round (i.e., methods marked neutral are not counted towards the N, yet they count towards the effectiveness measure).

The number of methods analyzed by the developer before he stopped (i.e., the effectiveness measure), either because a target method was found or because Rocchio failed is reported in parenthesis (marked with m). This number includes all the methods marked by the developer in all the rounds of feedback, including also the methods marked as neutral, plus the rank of the target method in the final round. To complete the picture, the number of feedback rounds is also reported in parenthesis (denoted with r), including the (incomplete) round when the target is found.

For example, row #2 in Eclipse, reads as follows. Baseline (**17**, 42, 47) means there are three target methods and the best ranked is on position 17. Rocchio with *N*=3 (4, **1**, 2 (**7m**/3r)) means that one of the three target methods (i.e., the second) was ranked #1 on the 3rd round and the user marked a total of 7 methods to reach it (including the target method in the 3rd round). The two numbers to compare here are: 17 in the baseline vs. 7 in the Rocchio case. We consider that Rocchio improves here and highlight the table cell with dark grey. Cells marked with light grey show no improvement of Rocchio, but they are interesting as the cumulative ranking of methods is better than in the baseline (the number of investigated methods needs to be added here to the ranks of the target methods for a proper comparison). White cells indicate cases where Rocchio does not improve the baseline. NI marks the cases where there was no improvement for 2 consecutive rounds and 50m+ the cases where more than 50 methods were analyzed by the developer without reaching a target method. The stars in the white cells indicate the cases when Rocchio performed as good as the baseline.

The data reveals that Rocchio brings improvement over the baseline in 13 of the 18 change requests. In 3 cases, the improvement is observed for all values of N (i.e., all three Rocchio cells are dark grey in these rows). Rocchio with N=1 improved in 9 cases, Rocchio with N=3 improved in 9 cases, and Rocchio with N=5 improved in 8 cases, not all the same. More specifically, in Eclipse for 6 out of the 7 change sets reported, Rocchio retrieved one of the target methods more efficiently than the baseline. In jEdit, the ratio was 3 to 3, and in Adempiere Rocchio performed better in 4 out of 5 cases. We did not observe a pattern of when one of the values of N performs better than the other ones, nor about the magnitude of the Rocchio improvement over the baseline. So, we can not formulate at this time rules such as "*N*=5 is a better choice than *N*=3 or *N*=1", nor we can state that there is a correlation between the initial query and Rocchio improvements.

Qualitative Evaluation

One interesting phenomenon that we observed is that for one change set in jEdit and for one in Adempiere Rocchio did not improve the effectiveness of the baseline (based on our working definition), however it achieved a better cumulative ranking of the target methods. These two cases are marked with light grey in the table. We highlight these cases as we believe is still an indication that Rocchio brings some added benefit in these situations.

Another interesting and rather unexpected phenomenon is that in some cases where the there are more target methods the baseline favors one of them, whereas Rocchio helps retrieve another one faster. See Bug #23140 in Eclipse and Patch #1649033 in JEdit (the light grey cell).

We identified cases when neither the ranking of the first target method, nor the cumulative ranking of Rocchio was better than in the case of the baseline (i.e., all white rows in the table). Our initial assumption was "if the initial query is really poor, Rocchio does not help much". However, this is not true as there were several cases where the initial query led to poor results, yet Rocchio improved them drastically (i.e., by one order of magnitude). For example, see Bug #19686 in Eclipse, Patch # 1469996, and Patch #1607211 in JEdit.

We then investigated the cases with poor Rocchio performance in more detail. For example, in the case of Bug #19691 in Eclipse, we found that the methods the developer would consider as being relevant based on the bug description would in fact not be relevant, even if they contained related terms from the bug description. The bug description is about exporting preferences for the team, whereas the target methods just contained "ignore" settings in the team preferences. This case highlights the difficulty of concept location in practice. Change requests are often formulated in terms different that the source code, both linguistically and logically. We can safely conclude that Rocchio brings improvements over IR based concept location in many cases, but it is far from being a silver bullet.

4.1.2.4 Threats to Validity

This section presents the threats to the validity of the study and of the results obtained, organized by threat category [128].

Threats to *construct validity* concern the relationship between theory and observation. To evaluate the CL task, we used the effectiveness measure and the ranks of the relevant methods in the list of results, which are widely used measures in concept/feature location studies since they provide a good estimation of the effort that a developer needs to spend in a TR-based concept location task.

Threats to *internal validity* concern co-factors that can influence the results. In our study we automatically extracted the set of queries from online bug tracking systems. Such queries are approximations of actual user queries and in practice, the user may reformulate the query along the way and IR may retrieve better results with the user reformulated query. Simply put, the study approximates the situation when the developer is not good at writing queries. However, developers are often faced with unfamiliar systems, in which cases they must rely on outside sources of information, such as bug 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. We also argue that in the case when developers formulate good initial queries, Rocchio is not needed. In fact, as the results revealed, 12 of the 30 bug descriptions produced great initial queries. It is important to clearly establish the cases where explicit Rocchio helps.

Threats to *conclusion validity* concern the relationship between treatment and outcome. Our results are based on the feedback provided by only one user. Different people might give different feedback to Rocchio. Also, the change requests were selected by a researcher from the pool of change requests available, which had also the patches available. Even though the researcher selecting the change requests was not the one providing the relevance feedback and there was no knowledge of the results for particular change requests at the time of the selection, we are aware of the fact that selecting other changes might lead to different results.

We used only three values of *N* (i.e., 1, 3, and 5) in the study and a single weighting scheme in the Rocchio implementation. We are aware of the fact that other values of *N* might retrieve different results. However, these values are within the range of values usually adopted in the implementation of explicit relevance feedback and represent a reasonable amount of information for a user to analyze in one round of feedback. The current set of weights used in our Rocchio implementation was chosen based on empirical evidence. Other weights could lead to slightly different results.

Threats to *external validity* refer to the generalization of the results we obtained. Regarding the systems used for the case study, we tried to mitigate this threat, by selecting three software systems from diverse domains. A larger set of queries and more systems would clearly strengthen the results from this perspective. While we used data from several systems, we only used a single TR engine (i.e., Lucene). The results may differ when using other TR engines.

4.2 Automatic Query Reformulation for Text Retrieval in Software Engineering

While the semi-automatic reformulation of queries led to promising results, one of the shortcomings of this approach is that the developers still need to put in effort in analyzing the list of results and marking them according to their relevancy. An automatic approach, able to reformulate queries without developer assistance would be desirable. Researchers have looked at this issue and proposed a series of approaches for automatic query reformulation in software engineering [27, 50, 61, 87, 117, 127]. However, these approaches also have a limitation: they apply the same reformulation approach to all queries.

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

In this section, we propose and evaluate an automated approach that, for a given query, recommends a reformulation strategy based on its properties in order to improve its results. We call this recommender *Refoqus* (*REF*ormulation *O*f *QU*erie*S*). Refoqus is based on the idea that the properties of a query dictate the best reformulation technique to be used with it. It relies on historical data of queries, their properties, and their performance when subjected to different reformulation approaches in order to learn the best pairings between query types and reformulation techniques. To determine the type of a query, it relies on a set of measures indicating several query quality attributes, i.e., *specificity*, *coherency*, *similarity*, *term relatedness*, *robustness*, and *score distribution*. These properties and the measures capturing them have been shown to correlate with the performance of queries in the field of natural language document retrieval [23]. Section 3.2 presents an application of these measures in software engineering for predicting the quality of TR queries and offers also a description of the 28 measures used. We use the same set of measures in Refoqus, as these measures have been carefully selected to be applicable to software data.

Refoqus determines among a set of possible reformulations the best one to use for each individual query. We selected four reformulation strategies proposed in the field of natural language document retrieval (see Section 4.2.1), which perform best in that field, yet they are appropriate for software engineering data. Refoqus automatically applies each reformulation strategy for the queries in the training set and learns which reformulation strategy works best for which type of query (based on the relevant properties). Given the model it builds based on training data, it is able to determine the best reformulation for incoming queries, based on measuring just their properties. The underlying algorithms of Refoqus are generic, so the measures and recommendation strategies can be replaced, if needed.

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

4.2.1 Background on Automatic Query Reformulation Approaches

In this section we introduce terminology and definitions necessary to understand the reformulation strategies used by Refoqus. The goal of query reformulation is to define a new query, starting from the initial one, which is able to lead to improved retrieval results. What exactly "good search results" means can differ according to context in which the search is used, but it usually refers to the relevant documents being as close as possible to the top of the search results list. This is the interpretation of quality we adopt in this work, and we instantiate it later in our evaluation on concept location.

Over time, researchers in the field of TR have proposed and investigated a large variety of approaches for producing candidate reformulations for an initial query. These approaches fall in two categories [75]: *query expansion* approaches and *query reduction* approaches. We introduce briefly each category with emphasis on the reformulation strategies used in our proposed approach.

Query Expansion

Query expansion is meant to offer a solution to the problem known as "the vocabulary mismatch problem" [46], where the terms in the query do not match the vocabulary of the relevant documents in the corpus. A variety of query expansion approaches have been proposed in the field of TR. We found, however, that not all these were applicable to our circumstances (i.e., source code based corpora). We selected three existing approaches in the following way. We did not consider approaches that relied on linguistic features or on sources of information external to the corpus, like the web, ontologies, Wikipedia, or Wordnet. Such approaches are designed to work for natural language documents as they rely on word relationships that exist in English. Since we target source code-based corpora and previous studies [118] have shown that words do not share the same relationships in source code as they do in natural language, we decided not to consider such strategies in our recommender.

Some approaches [23]are based on algorithms with high computational complexity to produce reformulations for a query. Since our end goal is to produce a recommender which can be used by developers during their daily tasks, we did not consider such approaches practical and thus, we did not select them.

Finally, from all other available strategies we selected seven expansion strategies that are reported to perform best in the TR literature [24]. We performed a preliminary evaluation with the seven expansion strategies and selected the best three approaches to be used by Refoqus. The results of the preliminary study can be found in the Appendix of this dissertation. This final selection of reformulation approaches was necessary in order to reduce the number of categories considered by the machine learning approach when performing the classification step in order to assign queries to a reformulation approach. This is needed in order to accommodate the situations where the training data available for a particular software system is limited, and is therefore not enough to learn accurate classification models based on many categories. This was the case also with the software systems we considered for our study on concept location, where the number of queries available was limited.

Even though Refoqus in its current implementation makes use of only four reformulation strategies overall (three expansion and one reduction technique), it is able to obtain very good results and improve the results of queries after reformulation. However, *Refoqus is designed to be flexible, such that any reformulation strategy can be replaced and additional ones can be added*.

All three query expansion strategies selected are based on some form of *pseudo-relevance feedback*, in that they consider the top K documents from the list of results as relevant documents to the query. Then they use different techniques to order the terms in these K documents and select the top N ones to use for the query expansion. Currently, we use K=5 and N=10 in the implementation of Refoqus. However, these parameters can be modified as needed and we plan to experiment with more values in the future.

The first strategy is similarity-based and orders the terms in the top K documents based on their *Dice* similarity (see below) with the individual query terms. The idea behind Dice similarity is that two terms are related if they appear in the same documents in the corpus, a common assumption in all TR engines. The formula of the Dice similarity is:

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

Where:

- *u* is a term from the query
- *v* is a term from the top K documents

- *df* denotes the number of documents in the corpus containing *u*, *v*, or both *u* and *v*, respectively.

The other two techniques do not rely on similarities with the terms in the query. The idea is to use the first K documents retrieved in response to the original query as a more detailed description of the underlying query topic. Therefore, descriptive terms for this topic can be used for expansion, and can be determined by identifying the most representative terms for the set of top retrieved documents. One of the approaches is based on *Rocchio*'s [111] method for relevance feedback and assigns a score to each term in the top K documents based on the sum of the tf-idf scores of the term in each of the K documents. Tf-idf is a score often used in the field of TR to determine the importance of a term for a particular document relative to the corpus. The formula of Rocchio is:

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

Where:

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

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

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

Where:

- *C* denotes the collection of documents in the corpus

- *R* is the set of top K relevant documents in the list of retrieved results
- *d* is a document in R
- *t* is a term in d
- *p(t|R)* is the number of times t appears in the top K documents in the list of results (R) divided by the number of terms in R
- *p(t|C)* is the number of times t appears in the whole document collection(C) divided by the number of terms in C.

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

Query Reduction

Query reduction is based on the idea that the query contains both important information as well as noise, i.e., words that do not contribute to the main intent of the query and may hinder the retrieval of relevant documents. Therefore, query reduction should help improve the results of a query. In the absence of user feedback and information about the semantics of the query, automated query reduction needs to be done with care, as intrusive reduction strategies may actually harm the results [75].

We adopt a conservative reduction strategy, which eliminates the terms that appear in more than 25% of the documents in the corpus, as they are considered non-discriminating. We previously used this strategy with Rocchio (see Section 4.1) for filtering the set of terms added to a query when reformulating it.

4.2.2 REFOQUS

Refoqus is based on the idea that the properties of a query are indicative of the best way to reformulate it. Therefore, it uses historical data capturing the measures of query quality and the performance of queries when reformulated using the different reformulation approaches in order to learn, using a machine learning algorithm, the best reformulation techniques for different types of queries.

When learning the best reformulation approaches, Refoqus uses a classifier and assigns a label to each reformulation approach (including a label for “none”, indicating that the query leads to the best results when left in its original form). When a new query comes in, it will be assigned one of these labels based on its properties and on the model learned by Refoqus from the historical data.

Refoqus has two main steps: (1) training the classifier; and (2) using the classifier to recommend the best reformulation technique for incoming queries.

Training the Classifier

Refoqus needs a training data set for its classifier. The training data consists of queries and their associated relevant documents. Refoqus communicates with the TR engine used by the developer in order to run a query and get its list of results. In the current implementation (which we used in the empirical evaluation from the next Section), we used Lucene[25]. Refoqus executes the following steps in order to train its classifier:

a) Refoqus uses the TR engine to rank all the relevant documents for each query in the training data set.

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

b) The values of the 28 query property measures are computed for each of the queries in the training data.

c) The four reformulation techniques are applied, one at a time, to each query in the training set and the resulting reformulated queries are run by the TR engine.

d) The results obtained by the four reformulation variants are compared and the best performing reformulation is determined for each query.

e) If there are queries that led to no relevant document being retrieved by the TR engine after they were run in their original form and in any of the reformulated forms, then these queries are removed from the training set. This is a necessary step, as for such queries Refoqus will not be able to make any recommendation, given that it cannot decide which is the best reformulation strategy.

f) The classifier is trained using the collected training data. One data point in the final training data used by the classifier corresponds to a query. Each data point has 29 attributes, 28 attributes corresponding to the query property measures and one corresponding to the best reformulation strategy.

We chose classification trees [19] as the machine learning techniques, due to their advantages and our previous good results in using them for predicting the quality of queries (Section 3.1.2.4). The rules produced by classification trees are easy to understand by humans, which is not true for other, more complex models. Hence, a developer could interpret easily the recommendation made by Refoqus, before allowing it to automatically reformulate the query, if she chooses to do so. Second, classification trees perform implicitly feature selection. This is a very important property, as it allows Refoqus to be less sensitive to the choice of query property measures. In the current form, it allows us to give as input all 28 measures of a query, as the classification tree will determine automatically the subset of measures relevant for the classification, with little overhead. The subset of measures used by Refoqus always contained only two measures, selected among the 28 given as input. Note that this set of measures can change between systems and between different evaluation rounds for cross- validation within the same system.

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

There are two possible approaches when training the classifier, namely *within-project* and *cross-project training,* each having advantages and disadvantages. In *within-project training*, the classifier is trained and tested on the same system, and the evaluation is done independently for each software system. In order to ensure the least bias in the evaluation, all data points should be used for training and testing at some point. For this purpose, cross-validation is used, where the evaluation is done in several rounds, such that all data points get to be evaluated exactly once, in one of the rounds. In each round a small part of the data is kept for testing, while the rest is used for training. When performing this kind of validation it is important to select balanced training sets, where there are enough data points to learn from for each possible class and that the number of data points belonging to each class is balanced in the training set. In the case of Refoqus this means that the training sets need to be chosen such they contain approximately equal numbers of data points assigned to each of the reformulation strategies.

In *cross-project training*, given a set of n systems, the classifier is trained using all data points from n-1 systems and then tested on the data from the n^{th} system, which was not included in the training. This evaluation is repeated n times, each time considering one of the systems for testing and the rest for training.

Cross-project training has the advantage that it does not require training data for a new system, thus simulating a plausible scenario when such data is not available. However, it may miss some project-specific properties of the data, which the within-project training may be able to take advantage of for producing more accurate results. We investigate both approaches in our evaluation, described in Section 4.2.3.

The output of the training step, no matter the type of training used (i.e., within- or cross- project), is the classification tree, represented by a set of yes/no questions that splits the training sample into gradually smaller partitions that group together cohesive sets of data, i.e., those having the same value for the dependent variable. An example of classification tree built in our study is reported in Figure 4-1.

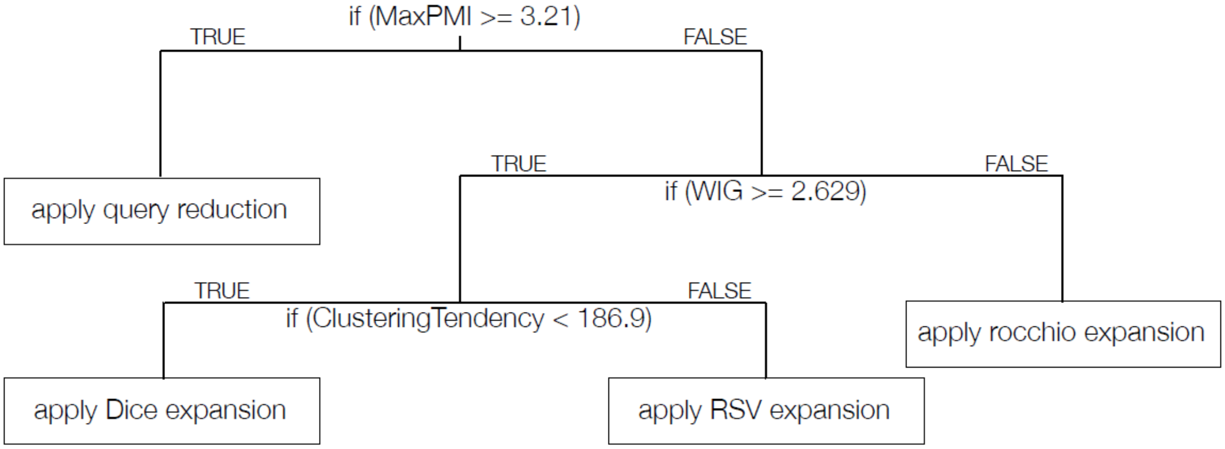


Figure 4-1. An example of classification tree

Using the Classifier for New Queries

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

4.2.3 Evaluation on Concept Location in Source Code

4.2.3.1 Study Design

We conducted an empirical study to investigate the performance of Refoqus in the context of TR-based concept location. There are several aspects of Refoqus that we want to evaluate. First, we want to establish which training strategy (i.e., within- or cross-project training) works better. Second, we want to establish whether the reformulations recommended by Refoqus improve the queries and if so by how much. Our conjecture is that the strength of Refoqus comes from the fact that it selects the best reformulation strategy for each query. Hence, third, we compare Refoqus with baseline approaches, based on the individual reformulation strategies used by Refoqus.

In order to address these issues, we formulated three research questions and conducted three experiments to answer them:

RQ1: Which training approach leads to better predictions for Refoqus?

RQ2: Does Refoqus improve the performance of the queries?

RQ3: Does Refoqus perform better than the baseline reformulation techniques?

Answering RQ1 allows us to determine and inform future users what is the best way to construct the training data. A positive answer for RQ2 implies that Refoqus can be used to improve TR-based concept location approaches (and hopefully TR approaches for other software engineering tasks). A positive answer for RQ3 confirms our conjecture that selecting the best reformulation strategy for each query is better than applying the same strategy to all queries.

4.2.3.2 Data

Our choice of empirical evaluation uses reenactment of concept location based on past changes. This is a very common evaluation technique used in feature/concept location research. Past changes in software provide us with a change request (or bug description in this case) and the actual changes in the code done in response to the request, named the change set. During concept location a user or a tool starts with the change request and finds a place in the code where a change should be made. To verify that this location is correct, the complete change should be implemented and tested. Reenactment based on historical data allows us to assess the correctness of concept location without complete implementation and testing. If concept location results in a place in the code that is in the original change set, then we can conclude that concept location succeeded. If the result of the concept location leads to a place that is not in the change set, then we consider that concept location failed.

Reenactment also allows us to automatically formulate queries for TR-based concept location. The bug reports contain both the title of the bugs and their description. In this study we automatically created queries considering two different options: (i) the title of the bug; (ii) the description of the bug. In addition, to have a better simulation of a usage scenario of the proposed approach, we also asked a Ph.D. student to manually formulate a query after analyzing only the bug report content. In the end, we obtained three queries for each bug report. For each query formulated for a bug report, the set of relevant documents to be retrieved is defined by the *change set.*

The same data set is used when answering each research question. We collected an initial set of 309 queries, corresponding to 103 bugs extracted from the bug tracking systems of five open source systems implemented in Java and C++: Adempiere[26] 3.1.0, ATunes[27] 1.10.0, FileZilla[28] 3.0.0, JEdit[29] 4.2, and WinMerge[30] 2.12.2. Adempiere is a common-based peer-production of open source enterprise resource planning applications. ATunes is a full-featured media player and manager. FileZilla is a

[26] http://www.adempiere.org/
[27] http://www.atunes.org/
[28] https://filezilla-project.org/
[29] http://www.jedit.org/
[30] http://winmerge.org/

graphical FTP, FTPS, and SFTP client, while JEdit is a text editor for programmers. Finally, WinMerge is a document differencing and merging tool.

We removed the queries for which no target method was retrieved when running the original query and all of its four reformulated forms. The data set was reduced to 94 bugs and their corresponding 282 queries. From this point on, we will refer only to these remaining 282 queries. The number of queries extracted from each project are reported, together with some size attributes of the object systems, in Table 4-2.

Table 4-2. Characteristics of the Five Software Systems

System	Version	Language	KLOC	#Methods	#Queries	#Bugs
Adempiere	3.1.0	Java	330	28,355	51	17
ATunes	1.10.0	Java	80	3,481	51	17
FileZilla	3.0.0	C++	240	3,240	72	24
JEdit	4.2	Java	250	5,532	54	18
WinMerge	2.12.2	C++	410	8,012	54	18
Total	**-**	**-**	**1310**	**48,620**	**282**	**94**

4.2.3.3 Planning and Execution

In order to generate term suggestions for query expansion, we used the top five documents in the ranked list of results. Also, when expanding the query, we considered the first 10 term suggestions. These decisions were made based on recommendations found in the domain literature [24].

After the collection of the data, we performed the following steps:

a) *Document corpus creation.* We built the source code corpus by considering each method in the system as a separate document. For each method, we extracted the terms found in its 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).

b) *Query execution and effectiveness measurement.* We performed the same text normalization process adopted for the methods on all the 282 queries and their reformulations. Then, we executed each query on their respective document corpus by using Lucene and measured the query effectiveness by identifying the position of the first relevant document (i.e., changed method) in the ranked list of search results. The higher the method appears in the result list (i.e., the lower its rank), the better the query performance.

c) *Answering RQ1.* To find out which training strategy works better, Refoqus was trained using the within- and cross-project strategy, respectively. For the within-project case, the classification model is trained on each system individually and a 4-fold cross-validation was performed: (i) randomly divide the set of queries for a system into 4 approximately equal subsets, (ii) set aside one query subset as a test set, and build the classification model with the queries in the remaining subsets (i.e., the training set), (iii) use the classification model built on the training set to identify the best reformulation technique for the queries in the evaluation set, (iv) repeat this process, setting aside each query subset in turn. The key element here is that each query is used only once in the test set. For the cross-project training, the queries from four of the five projects are used for training and the queries from the fifth project is used for evaluation. This is repeated such that the queries in each project are tested. The 282 queries were reformulated and the performance (i.e., the best rank among the methods in the change set) of the reformulated queries was recorded for each type of training. The two sets of performances were then compared.

d) *Answering RQ2*. To find out whether Refoqus improves the results compared to the original queries, the performance of the reformulated queries based on the Refoqus' recommendation were compared with the performance of the original queries.

e) *Answering RQ3*. We defined four baselines using the reformulation strategies employed by Refoqus: query reduction, Rocchio expansion, RSV expansion, and Dice expansion. Each baseline approach applies a single reformulation strategy to all 282 queries, respectively. For example, the reduction baseline applies query reduction to all queries. In order to analyze the comparisons, when comparing Refoqus with any of the baselines (or when comparing the two training strategies), we report the number of times the query reformulated by Refoqus and by the compared baseline has a better performance (i.e., lower rank of the top changed method) than the original query, the number of times the performances are the same, and the number of times the original query achieves better query performance. We also report the minimum, maximum, median, mean, and the 25% and 75% percentiles values of the differences in performance (i.e., difference in ranking of the top changes method).

The sets of results were also analyzed through statistical analysis using the Mann-Whitney test [18]. We chose this test as we cannot assume normality of data and the test does not make normality assumptions. The results are interpreted as statistically significant at $p < 0:05$. However, since we performed multiple tests, we adjusted our p-values using the Holm's correction procedure [64]. This procedure sorts the p-values resulting from n tests in ascending order, multiplying the smallest by n, the next by n-1, and so on.

4.2.3.4 Results and Discussion

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

Research Question 1

Table 4-3 and Table 4-5 report the improvement in results achieved by Refoqus compared to the initial query, for within- and cross- system training, respectively. Also, Table 4-4 and

Table 4-6 report the results for the queries whose performance was maintained or got worse after the reformulation with Refoqus. The *within-project* strategy achieves a mean query performance improvement of 262 positions (for 146 queries) and a maximum of 5,286, compared to the mean of 229 (for 113 queries) and the maximum of 5,197 obtained by the cross-project training strategy. At the same time, the number of queries that were improved using the within-project approach is higher by 33 queries compared to the cross-project approach, while the number of worsened queries is higher by only 6 for the within-project.

Table 4-3. Improvement results of Refoqus for within-project training

System	#Queries	#Improved	Improvement					
			Mean	Q1	QualQ	Q3	Min	Max
Adempiere	51	30	418	3	12	97	1	5,286
ATunes	51	29	85	5	9	86	1	667
FileZilla	72	42	383	7	163	611	1	1,409
JEdit	54	19	64	5	29	56	1	434
WinMerge	54	26	230	4	18	36	2	4,909
All	**282**	**146**	**262**	**4**	**23**	**166**	**1**	**5,286**

Table 4-4. Results that were worsened or preserved using Refoqus for within-project training

System	#Preserved	#Worsened	Worsening					
			Mean	Q1	QualQ	Q3	Min	Max
Adempiere	11	10	261	18	26	381	3	970
ATunes	12	10	54	5	40	100	1	324
FileZilla	23	7	90	10	21	106	1	371
JEdit	26	9	25	2	12	52	1	83
WinMerge	17	11	43	6	11	53	2	151
All	**89**	**47**	**100**	**5**	**19**	**86**	**1**	**970**

Table 4-5. Improvement results of Refoqus for cross-project training

System	#Queries	#Improved	Improvement					
			Mean	Q1	QualQ	Q3	Min	Max
Adempiere	51	15	585	7	11	109	1	5,197
ATunes	51	25	62	3	9	51	1	413
FileZilla	72	32	275	11	157	425	1	1,403
JEdit	54	18	68	7	29	61	1	434
WinMerge	54	23	242	2	8	46	1	4,603
All	**282**	**113**	**229**	**4**	**15**	**157**	**1**	**5,197**

Table 4-6. Results that were worsened or preserved using Refoqus for cross-project training

System	#Preserved	#Worsened	Worsening					
			Mean	Q1	QualQ	Q3	Min	Max
Adempiere	33	3	71	25	49	107	1	165
ATunes	20	6	107	41	51	148	4	319
FileZilla	28	12	105	22	27	158	2	437
JEdit	26	10	112	8	52	71	1	781
WinMerge	21	10	34	2	5	16	1	164
All	**128**	**41**	**87**	**5**	**28**	**96**	**1**	**781**

We therefore hypothesize that the within-project approach leads to better results than the cross-project one and we use the Mann-Whitney test to test this hypothesis. The Mann-Whitney test is a non-parametric test (therefore does not assume normality of the data) applied to observe if a particular treatment leads to significantly different results compared to another treatment or the original state. In our case, the test reports statistically significant differences between the performance values of the reformulated queries using the two approaches, in favor of the within-project training (p-value=0.002, mean=-40). A mean value of -40 indicates the within-project training returns the first relevant method 40 positions on average higher in the results list than the cross-project training, therefore leading to an average of 40 less methods that need to be analyzed before finding the first relevant method in the list of results.

RQ1 answer. We conclude that the *within-project* training is superior to the cross-project training. Nonetheless, *cross-project* training for Refoqus still manages to improve or preserve the performance of a large number of the original queries. This indicates that cross-project training could be still used, when within-project data is not available.

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

Research Question 2

When compared to the performance obtained by original queries, Refoqus is able to improve or maintain the performance of 235 out of the 282 queries (Table 4-3) on which it has been applied (i.e., 84% of the queries). This improvement is in several cases by hundreds or thousands of positions. When analyzing the results, it is important to focus on the performance in the worse cases, as these are the situations where Refoqus is most useful (i.e., when the original query is really bad). When the original query is already good (for example, the best ranked method is in top 10), reformulation strategies in general led to small improvements or no improvement. The rather large difference between the median and mean improvements indicates that many "bad" queries had large performance improvements.

We therefore hypothesize that Refoqus (using the within-project approach) leads to better results than the original query. We make use again of the Mann-Whitney test to verify this hypothesis. The results indicate that the difference between the effectiveness measure as returned by Refoqus and that of the original query is statistically significant (p-value<0.0001, mean = -119). On average, Refoqus is able to obtain an improvement (i.e., a lower effectiveness measure) of 119 positions in the list of ranked results and this improvement is statistically significant.

RQ2 answer. We conclude that the query reformulation recommendations formulated by Refoqus led to the improvement or preservation of the query performances in most cases (52% of the queries improved their performance and 32% preserved it).

We discuss some examples and observations from the data, in order to get a better understanding of the cases when Refoqus works the best or does not work. An example of large improvement in query performance was observed on a query in the FileZilla system. The original query was automatically extracted from the title of the bug report: *set use medium large icon*. Using this query the first target document retrieved was the method `LoadPage` from the `COptionsPageThemes` class, on position 175.

Refoqus suggested to apply the Rocchio expansion, and was reformulated as: *set use medium large icon theme panel scroll preview wx ptheme*. In other words, the terms *theme, panel, scroll, preview, wx, ptheme* were added to the query. The reformulated query retrieved the same target method (i.e., `COptionsPageThemes.LoadPage`) on position 6 of the ranked list. When analyzing the content of this method we observed that all the terms added by the Rocchio expansion were present in the body of the method: *wx* (25 occurrences), *theme* (24), *panel* (12), *ptheme* (9), *scroll* (6), and *preview* (2); which explains the improvement.

Further analysis of the queries that preserved their performance after reformulation revealed that, for all of them, Refocus recommended query reduction. One observation is that, when applying this technique the query is not always modified (only if it contains "non-discriminatory" terms that appear in more than 25% of the methods in the system, which is not always the case).

We also noticed that 20 of the queries achieving stable performances were not improvable, that is, they already retrieved the first relevant method on the first position. The fact that Refoqus does not decrease the performances of these queries is certainly a notable result. Another 22 original queries retrieved the relevant method in the top ten positions of the ranked list.

There were 47 (17%) cases when the performances of the reformulated queries using Refoqus decreased. The decrease was, on average, of 100 positions in the ranked list, which is, less than half of the average improvement obtained by Refoqus on the improved queries. In other words, the potential negative effect of the reformulations may be outweighed by the significant improvements.

It is also worth noting that we did not observe significant differences between the percentage of manually formulated queries that were improved by Refoqus (51%) and automatically extracted queries that were improved (52%). We also did not observe significant differences between the C++ systems and the Java systems, which indicates that Refoqus is robust with respect to this aspect.

Research Question 3

Table 4-7 compares Refoqus and the four baseline reformulation techniques.

Table 4-7. Comparison between Refoqus and the baseline reformulation techniques on the 282 queries of the study

Reformulation	#Preserved	Improvement							Worsening						
		#	Mean	Q1	QualQ	Q3	Min	Max	#	Mean	Q1	QualQ	Q3	Min	Max
Reduction	242	47	78	4	15	33	1	530	13	15	2	4	20	1	59
Rocchio	28	124	166	3	14	148	1	5286	130	100	6	28	127	1	1280
RSV	22	146	233	4	21	178	1	4843	114	148	5	29	103	1	4529
Dice	18	127	266	4	32	237	1	5197	137	314	5	52	204	1	12829
Refoqus	**89**	**146**	**262**	**4**	**23**	**166**	**1**	**5286**	**47**	**100**	**5**	**19**	**86**	**1**	**970**

The obvious observations are: the number of queries improved by Refoqus is matched by RSV Expansion (i.e., 146), the mean improvement is slightly better for the Dice Expansion (i.e., 266 vs. 262), and the number of queries with reduced performance after reformulation is better for Query Reduction (i.e., 13 vs. 47). However, the RSV Expansion, along with the other expansion techniques led to the worsening of the results for two to three times as many queries than Refoqus. We conclude that there is a higher risk to use them over Refoqus.

We can see that the number of queries with preserved results when applying the Query Reduction is very large (86%). As explained before, these can be explained by the fact that this technique is rather conservative and it only eliminates words from the query in few cases, keeping the query unchanged in many cases.

Finally, Table 4-8 reports the results of the Mann-Whitney Test performed between the results of Refoqus and each baseline, respectively. The tests indicate that Refoqus achieves statistically significant better results compared to each baseline. Indeed, the mean of differences is negative, showing that Refoqus achieves, on average, lower (and thus better) effectiveness measures for the queries.

Table 4-8. The Mann-Whitney Test for the comparison between Refoqus and the baselines

Test	p-value	Mean
Refoqus vs. *Reduction*	<0.0001	-112
Refoqus vs. *Rocchio*	<0.0001	-92
Refoqus vs. *RSV*	<0.0001	-58
Refoqus vs. *Dice*	<0.0001	-152

RQ3 answer. We conclude that Refoqus outperforms the baseline approaches considered.

4.2.3.5 Threats to Validity

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

Threats to *internal validity* concern co-factors that can influence the results. In our study we automatically extracted the set of queries from the online bug tracking system of the object systems. Such queries are approximations of actual user queries.

However, developers are often faced with unfamiliar systems, in which cases they must rely on outside sources of information, such as bug 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. However, in order to mitigate such a threat we also asked a Ph.D. student to manually formulate queries as well.

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

Threats to *conclusion validity* concern the relationship between treatment and outcome. Where appropriate, we used non-parametric statistical tests (Mann-Whitney) to show statistical significance of threats *to external validity* concern generalization of the obtained results. In order to mitigate this threat, we selected five software systems from diverse domains, implemented in two programming languages, i.e., Java and C++. A larger set of queries and more systems would clearly strengthen the results from this perspective. While we used data from several systems, we only used a single TR engine (i.e., Lucene). The results may differ when using other TR engines.

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

4.3 Related Work on Query Reformulation

In the field of natural language document retrieval, query reformulation has long been established as a way to improve the results returned by an TR engine [111]. Various approaches have been proposed over time, which fall in two main categories: query reduction [10, 125] and query expansion [24] approaches.

In software engineering, a few works have also taken advantage of query reformulation strategies in order to improve software engineering tasks supported by TR. A few studies have investigated the manual reformulation of queries by developers. Query reformulation using ontology fragments has been investigated in the context of concept location by Petrenko et al. [95]. In this work, developers build and update ontology fragments which capture their knowledge of the system and then reformulate queries based on these fragments, leading to improved results. Starke et al. [119] have studied how developers search source code when performing corrective tasks on an unfamiliar system. Their findings indicate that even after several reformulations some developers are unable to locate the information they need. These studies provide motivation for our work as they support the need for automatic techniques for query reformulation.

The semi-automated (i.e., interactive) approach for reformulating the queries, which requires the intervention of a developer for relevance feedback [111] was previously used to improve TR-based traceability link recovery between various types of software artifacts [38, 57]. The results suggest that user relevance feedback generally benefits software engineering tasks. However, they also underline that it is not always the solution.

A few papers have investigated automated query reformulations. These approaches are usually based on reformulating the query using words that are either similar or related in some way to the query terms. Some of these approaches determine word relations based solely on their usage in source code. For example, Marcus et al. [87] have used Latent Semantic Indexing in order to determine the most similar terms to the query from the source code and include them in the query. Yang et al. [127] use the context in which query words are found in the source code to extract synonyms, antonyms, abbreviations and related words to include them in the reformulated query. Hill et al. [61] also use word context in order to extract possible query expansion terms from the code. Shepherd et al. [117] build a code search tool that expands search queries with alternative words learned from verb-direct object pairs. Other approaches make use of external sources of information in order to determine the related words that should be included in the query. Web mining is used [27, 50] to identify web documents relevant to the query from which to extract domain terms to replace the original query.

A common feature of these automated techniques is that they utilize the same reformulation strategy, regardless of the query or system used. In contrast, Refoqus chooses and recommends the best reformulation strategy for each given query and system. In this chapter we presented the first approach using query quality measures are used as attributes for learning the best reformulation technique among several options for each individual query.

CHAPTER 5 CONCLUSIONS AND FUTURE WORK

During software development and evolution a variety of software artifacts are created, such as, requirements, change requests, bug descriptions, etc. These artifacts have different representations and contain different types of information. The textual information found in software artifacts captures knowledge about the problem and solution domain, about developers' intentions, client demands, etc. Text Retrieval (TR) techniques have been successfully used to leverage this information. Despite their advantages, the success of TR techniques strongly depends on the textual queries given as input. When poorly chosen queries are used, developers can waste time investigating irrelevant results.

In this dissertation we proposed approaches to automatically capture and predict the quality of TR queries in the context of software engineering tasks. Also, we introduced novel techniques for automating query reformulation, which can help developers in the cases when their queries lead to poor results. In particular, this dissertation makes the following main contributions:

- We developed and validated a new measure to capture the specificity of TR queries in the context of software engineering tasks. The new measure, called Query Specificity Index, is evaluated in a study on concept location, revealing that it is able to capture the quality of a query better than the leading specificity measure proposed in the field of natural language document retrieval.
- We developed and validated a novel approach, called QualQ, which is able to automatically predict the quality of queries in the context of software engineering tasks based on the statistical properties of the text they contain. We evaluated

QualQ for concept location in source code and showed that it is able to correctly predict the quality of queries in 85% of the cases.

- We proposed and validated the use of an approach based on the Rocchio algorithm, which uses developer feedback for automatic reformulation of TR queries in the context of concept location in source code. We evaluated the approach in a study on concept location and the results showed that the Rocchio-based relevance feedback can generally improve the results if TR concept location.
- We developed and evaluated a novel approach, called Refoqus, for automatically reformulating TR queries in the context of software engineering tasks by automatically determining and applying the best reformulation approach for a query based on its properties. We evaluated Refoqus in the context of concept location in source code and the results of the study revealed that Refoqus is able to improve or preserve the results of TR queries for CL in 84% of the cases.

While the work presented in this dissertation represents an important step towards addressing the problem of query formulation for TR-based approaches for software engineering, there are still steps to be made in this direction, beyond the scope of this dissertation. In particular, we aim to pursue the following research directions in the future:

- Extensive user studies for Rocchio. The study we performed for evaluating Rocchio made use of the feedback of only one developer. We plan to replicate the study in the future and involve a larger number of developers.

- Query quality range. When determining the quality of queries, QualQ currently considers queries as either high or low quality. We plan to investigate also the use of a range to describe the quality of a query, rather than the current binary approach.
- Applying QualQ and Refoqus to other software engineering tasks. So far we have evaluated our approaches for automatic query quality prediction and automatic query reformulation in the context of concept location. However, there are many more software engineering tasks that rely on TR techniques and could benefit from these approaches. We plan to investigate the applicability and results of QualQ and Refoqus in the context of other software engineering tasks.
- Investigate new measures for query quality. We observed that a new query measure, QSI, used for determining the specificity of a query, performed better than the state of the art specificity measure from the field of natural language document retrieval. We plan to investigate the use of new measures, adapted to software data for all the query properties presented in this dissertation, and refine existing measures to account also for the location of the terms in the code.
- Investigate more training and evaluation data variations for QualQ and Refoqus. The results of QualQ and Refoqus depend greatly on the data on which they are trained and evaluated. We plan to investigate the effect of evaluating a trained model on various versions of a software system, with the goal of determining the spots when the models should be retrained, due to changes in the system in response to software evolution. We will also investigate the possibility of using training data from multiple versions of a software system. We also plan to

investigate how the accuracy of the two approaches changes according to the type of the system being evaluated. In that regard, we will experiment with using systems from the same problem domain for training and testing and observe if the results improve. We will also experiment with different sizes of the training set, in order to determine how sensitive the training is to the size of the training sample and the number of systems used.

- More reformulation approaches in Refoqus. Currently Refoqus considers three query expansion and one query contraction approaches as the possible options for reformulating the queries. We plan to integrate other approaches for query reformulation in Refoqus, previously proposed in software engineering and natural language document retrieval. In order to allow for enough examples to learn from for the new reformulation approaches, more evaluation data will be collected. Also, Classification and Regression Trees may not represent the best machine learning solution when more categories are considered. We will investigate other classification approaches which allow for many categories.
- Integrate Relevance Feedback and Refoqus. Currently Refoqus relies only on statistics to determine the best reformulated query and therefore it does not make use of any semantic information. We plan to investigate an approach which combines the power of Refoqus in determining the best reformulation of a query based on its properties with relevance feedback given by developers, which can provide semantics and guide the search in the right direction.
- Investigate more IR techniques. The quality and the best reformulation of a query can depend also on the IR technique used. We plan to repeat the

studies performed so far using various IR techniques and observing how the quality and reformulation of queries change according to the IR engine used.

- Use developer judgments for concept location. So far we have used reenactment in order to determine the relevant methods for concept location. In our future work, we plan to have also ask developers to actually implement changes and determine the relevant methods to a change this way.
- We plan to performs a more in-depth analysis of the differences and similarities between user queries and automatically extracted queries and observe their quality and best reformulation technique.

APPENDIX

Table 5-1. Results for all queries from all systems in the preliminary study of seven reformulation approaches (Section 4.2)

	Rocchio	RSV	Dice	KLD	NumPseudoDoc	MutualInfo	RelevanceModel
Sum ranks	**89704**	**80100**	**97427**	105339	102870	104977	107668
Average Rank	**318**	**284**	**345**	374	365	372	382

Table 5-2. Results for all queries of Adempiere in the preliminary study of seven reformulation approaches (Section 4.2)

ID	Rocchio	RSV	Dice	KLD	NumPseudoDoc	MutualInfo	RelevanceModel
1	22	25	26	23	23	24	27
2	18	27	43	44	22	25	35
3	166	206	257	257	308	379	471
4	34	23	22	11	20	20	18
5	32	35	41	41	12	14	22
6	4	7	13	33	26	30	34
7	137	324	471	521	389	462	530
8	905	1728	1845	1845	1668	1807	1842
9	2	1	1	1	1	1	1
10	45	57	94	104	150	180	243
11	5	4	2	2	2	2	1
12	33	45	60	48	35	32	42
13	7	5	6	17	15	15	14
14	114	91	116	207	182	193	192
15	83	89	95	71	72	55	63
16	280	602	1108	1012	324	285	427
17	71	50	32	43	60	50	38
18	453	375	379	372	485	541	515
19	110	224	429	435	2166	3197	4040
20	7740	2334	1979	1980	3409	2727	2479
21	11	12	15	11	6	7	7
22	901	321	342	335	249	268	282
23	1	1	1	1	1	1	1
24	5	5	4	3	5	7	8
25	8	10	11	11	11	11	11
26	2	1	2	2	3	2	2
27	1	1	1	1	1	1	1
28	19	101	373	687	744	1007	1123
29	18	19	18	17	15	13	15
30	12	10	11	11	12	12	12
31	18	67	52	150	203	280	235
32	377	1721	2372	3031	2766	3175	3489
33	2	2	2	2	2	2	2
34	87	71	68	64	68	65	62
35	115	127	144	95	69	57	65
36	55	62	74	69	63	43	51
37	4	7	13	33	26	17	19

ID	Rocchio	RSV	Dice	KLD	NumPseudoDoc	MutualInfo	RelevanceModel
38	946	1728	1845	1845	1668	1671	1810
39	2	1	1	1	1	1	1
40	27	23	24	33	17	16	17
41	1	1	1	1	1	1	1
42	280	602	1108	1012	324	285	427
43	281	315	934	286	275	286	312
44	1926	4731	5782	13664	15498	15656	15677
45	756	5252	13552	13552	15282	15284	15410
46	1	1	1	1	1	1	1
47	11	11	11	11	13	14	18
48	1	1	1	1	1	1	1
49	9	8	9	9	9	9	9
50	52	955	1177	1455	1730	1729	1776
51	11	4	3	4	5	3	3

Note: The queries for which none of the reformulation approaches retrieved any relevant method were removed from the results for brevity, as they do not contribute to the decision.

Table 5-3. Results for all queries of ATunes in the preliminary study of seven reformulation approaches (Section 4.2)

ID	Rocchio	RSV	Dice	KLD	NumPseudoDoc	MutualInfo	RelevanceModel
1	5	8	5	4	6	4	4
2	570	47	42	29	55	49	47
3	661	727	821	609	434	475	527
4	93	83	74	85	94	106	98
5	504	197	209	260	245	204	173
6	144	119	136	155	83	83	95
7	2	2	2	3	4	4	4
8	31	20	29	40	12	14	22
9	331	110	76	65	106	106	93
10	331	110	76	65	106	106	93
11	6	6	5	9	9	9	8
12	22	38	109	63	76	38	74
13	2	3	3	2	2	2	2
14	20	17	16	17	19	19	19
15	4	4	4	4	3	3	2
16	303	362	651	599	639	723	815
17	78	75	73	93	125	128	127
18	2	2	2	2	3	3	3
19	60	36	30	25	40	41	45
20	119	31	11	17	20	21	15
21	21	20	21	18	20	20	20
22	8	13	21	21	10	10	12
23	6	7	9	8	8	8	9
24	2	1	1	2	2	2	2
25	187	209	221	224	233	242	251
26	43	45	62	67	44	51	56
27	1	1	1	1	1	1	1
28	3	2	2	2	1	1	1
29	160	167	178	168	187	196	218
30	90	161	278	308	343	367	399
31	1	1	1	1	1	1	1
32	20	21	38	32	21	19	24
33	2	3	7	3	3	7	8
34	730	446	836	836	954	873	840
35	5	9	5	3	4	5	5
36	125	202	288	280	177	178	221
37	358	56	102	107	78	81	72

ID	Rocchio	RSV	Dice	KLD	NumPseudoDoc	MutualInfo	RelevanceModel
38	31	20	29	40	12	12	13
39	318	109	76	66	106	106	93
40	142	21	86	86	72	72	85
41	13	16	22	26	27	19	28
42	249	349	421	541	436	495	593
43	1	1	1	1	1	1	1
44	11	16	22	59	57	57	48
45	36	21	19	16	12	12	13
46	6	7	9	8	8	7	8
47	131	107	284	380	237	265	300
48	2	3	3	3	3	3	3
49	152	122	262	250	235	231	260
50	6	11	19	14	13	11	13
51	842	195	147	147	154	154	130

Note: The queries for which none of the reformulation approaches retrieved any relevant method were removed from the results for brevity, as they do not contribute to the decision.

Table 5-4. Results for all queries of FileZilla in the preliminary study of seven reformulation approaches (Section 4.2)

ID	Rocchio	RSV	Dice	KLD	NumPseudoDoc	MutualInfo	RelevanceModel
3	6	7	7	6	7	6	6
4	609	752	789	790	544	544	614
5	1127	165	90	107	130	162	127
6	379	308	306	387	316	317	312
8	1957	2014	2014	2014	740	740	767
9	3	4	4	5	5	5	5
10	93	73	61	49	66	69	69
11	235	416	741	788	485	487	606
12	11	3	2	1	1	1	1
13	437	525	623	623	784	791	802
14	1718	360	230	246	381	325	272
16	8	8	8	8	9	10	9
17	1674	616	491	536	819	819	661
18	26	15	13	41	28	29	21
20	24	58	143	211	177	183	234
21	5	2	2	2	2	2	2
22	315	248	214	259	301	331	308
23	1700	600	576	576	670	668	630
24	52	39	33	35	36	37	37
25	5	5	4	5	5	5	5
26	158	198	238	257	179	164	171
27	191	120	117	117	171	171	162
28	194	157	157	166	117	117	121
29	42	26	22	26	28	26	25
32	94	67	51	73	63	53	43
33	311	332	346	361	236	257	279
34	370	135	85	136	134	134	102
35	288	307	329	433	316	320	331
37	2366	2409	2409	2409	1115	1151	1163
38	6	16	33	34	22	36	45
39	154	112	102	62	62	63	65
40	1032	761	548	705	615	514	448
41	20	17	16	16	14	11	10
42	1301	1508	1529	1358	831	926	1012
43	2042	2099	2100	2163	2090	2090	2090
45	2	2	2	4	5	4	4
46	1986	2046	2046	2046	1561	1561	1561

ID	Rocchio	RSV	Dice	KLD	NumPseudoDoc	MutualInfo	RelevanceModel
47	9	6	4	4	6	5	4
49	129	136	146	139	95	109	120
50	6	6	6	5	5	5	6
51	561	581	605	642	787	814	840
52	1110	1005	917	917	891	830	816
53	173	66	38	38	78	57	52
54	8	7	8	9	10	9	9
55	4	3	3	2	2	2	1
56	372	369	259	241	277	277	227
57	117	82	61	29	34	30	27
58	47	49	48	51	68	69	65
61	5	5	5	5	5	5	5
62	230	384	563	575	604	604	722
63	797	435	331	408	376	376	354
64	384	298	311	426	288	288	289
66	1982	2074	2074	2074	338	338	385
67	5	5	5	5	5	5	5
68	1075	509	413	198	188	191	198
69	79	39	31	31	35	44	40
70	53	40	34	34	11	11	11
71	315	405	550	549	481	553	585
72	2140	738	434	364	422	333	301
74	9	6	6	7	7	7	6
75	1298	1101	909	916	1121	1126	1039
76	22	14	13	14	14	14	13
78	23	33	49	49	25	25	26
79	4	5	6	6	6	6	6
80	339	306	322	325	310	315	320
81	1807	1877	1877	1877	773	779	779
82	45	67	106	106	119	119	158
83	5	4	4	4	5	5	5
84	36	30	31	37	24	27	27
85	77	96	127	127	108	108	127
86	111	86	86	103	30	30	33
87	26	32	67	44	35	34	34

Note: The queries for which none of the reformulation approaches retrieved any relevant method were removed from the results for brevity, as they do not contribute to the decision.

Table 5-5. Results for all queries of JEdit in the preliminary study of seven reformulation approaches (Section 4.2)

ID	Rocchio	RSV	Dice	KLD	NumPseudoDoc	MutualInfo	RelevanceModel
1	15	19	40	11	10	12	17
2	145	175	207	188	255	278	307
3	393	415	440	489	452	476	507
4	4	6	6	7	6	6	6
5	522	366	231	343	363	268	214
6	316	276	225	223	266	223	199
7	1	1	1	1	1	1	1
8	1	2	2	2	2	2	2
9	4	4	7	7	6	8	8
10	21	8	7	7	8	7	6
11	771	745	709	757	769	737	695
12	22	29	49	33	25	41	58
13	1	2	2	1	1	1	1
14	91	156	314	268	228	308	378
15	37	58	94	124	71	95	129
16	74	45	25	26	34	38	28
17	54	69	107	141	109	123	142
18	59	66	73	84	86	92	96
19	9	11	15	11	8	9	11
20	518	693	867	867	711	711	839
21	911	1211	1446	1630	836	959	1094
22	2	10	16	22	12	15	19
23	16	5	4	2	2	2	2
24	720	633	385	305	413	413	361
25	1	1	1	1	1	1	1
26	1	2	2	2	2	2	2
27	1	1	1	1	1	1	1
28	3	3	2	2	2	2	2
29	1084	706	548	479	472	497	396
30	560	713	1042	1054	677	681	859
31	1	1	1	1	1	1	1
32	1	1	1	3	2	3	2
33	106	128	163	135	90	99	113
34	3847	4235	4235	4235	3440	3441	3441
35	107	144	230	90	80	80	107
36	8	30	46	91	49	33	51
37	9	11	15	11	8	9	11

ID	Rocchio	RSV	Dice	KLD	NumPseudoDoc	MutualInfo	RelevanceModel
38	89	120	164	186	82	93	111
39	59	88	125	115	92	82	97
40	18	51	79	70	41	40	46
41	42	30	29	31	36	36	34
42	272	140	69	45	132	134	106
43	1	1	1	1	1	1	1
44	1	2	2	2	2	2	2
45	13	8	10	17	19	22	22
46	43	29	24	23	22	24	22
47	970	1059	1142	1147	1087	1087	1115
48	112	98	87	99	153	158	156
49	1	2	4	4	2	3	4
50	4	2	2	1	2	2	2
51	35	49	79	70	57	71	86
52	51	37	29	31	29	30	30
53	23	25	32	32	45	51	56
54	33	54	87	90	82	82	100

Note: The queries for which none of the reformulation approaches retrieved any relevant method were removed from the results for brevity, as they do not contribute to the decision.

Table 5-6. Results for all queries of WinMerge in the preliminary study of seven reformulation approaches (Section 4.2)

ID	Rocchio	RSV	Dice	KLD	NumPseudoDoc	MutualInfo	RelevanceModel
2	217	14	322	254	293	314	324
4	7	9	8	10	8	9	10
6	75	98	137	175	102	77	96
7	902	1023	1189	1055	907	913	920
8	7197	2323	2017	1815	2544	2544	2187
9	54	48	81	59	62	65	65
11	6	10	11	10	11	11	11
12	35	40	49	41	48	67	67
14	6791	6920	6920	6920	6960	6960	6960
15	17	16	24	18	15	18	19
16	435	337	705	703	744	698	727
17	5	17	97	91	113	223	279
18	7	2	5	2	3	3	2
19	130	35	20	26	16	18	18
20	6	4	3	3	3	2	2
21	64	55	24	37	22	18	16
22	20	5	158	158	295	295	440
23	245	185	214	214	181	188	231
25	3	2	3	2	4	4	4
27	89	81	91	81	99	90	86
29	94	109	149	122	147	160	182
30	601	211	760	619	666	699	715
31	30	32	36	36	42	49	57
32	15	14	18	19	14	16	17
34	2	2	2	2	2	2	1
35	114	117	128	113	147	147	139
37	3	3	6	6	5	6	6
38	7	8	10	11	8	7	7
39	135	141	204	204	154	204	234
40	1	1	1	1	1	1	1
41	1	1	1	1	1	1	1
42	5	3	2	2	1	2	2
43	6	4	3	3	3	3	3
44	101	69	68	52	100	100	75
45	5	5	5	5	4	4	4
46	115	128	161	151	182	208	242
48	13	8	9	8	9	10	10

ID	Rocchio	RSV	Dice	KLD	NumPseudoDoc	MutualInfo	RelevanceModel
50	165	103	73	73	104	83	77
52	217	170	180	134	203	205	176
53	35	27	39	38	36	36	44
54	1503	230	1566	1100	2063	2063	1874
55	54	48	64	59	50	54	61
57	1	1	1	1	2	2	1
58	45	60	69	60	71	65	67
60	1	1	1	1	1	1	1
61	10	10	11	10	11	12	12
62	174	203	335	235	263	284	291
63	2	1	1	1	1	1	1
64	6	2	5	2	3	4	3
65	47	30	19	31	17	19	20
66	6	4	3	3	3	3	3
67	64	55	24	37	22	18	16
68	35	5	34	28	45	61	59
69	48	83	91	134	49	49	74

Note: The queries for which none of the reformulation approaches retrieved any relevant method were removed from the results for brevity, as they do not contribute to the decision.

BIBLIOGRAPHY

[1] Abadi, A., Nisenson, M., and Simionovici, Y., "A Traceability Technique for Specifications", in Proceedings of 16th IEEE International Conference on Program Comprehension (ICPC'08), Amsterdam, Netherlands, 10-13 June 2008 2008, pp. 103-112.

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

[3] Ahn, S.-Y., Kang, S., Baik, J., and Choi, H.-J., "A Weighted Call Graph Approach for Finding Relevant Components in Source Code", in Proceedings of 10th ACIS International Conference on Software Engineering, Artificial Intelligences, Networking and Parallel/Distributed Computing (SNPD'09), Daegu, Korea, 27-29 May 2009, pp. 539-544.

[4] Ali, N., Sabane, A., Gueheneuc, Y.-G., and Antoniol, G., "Improving Bug Location Using Binary Class Relationships", in Proceedings of 12th International Working Conference on Source Code Analysis and Manipulation (SCAM'12), 2012.

[5] Antoniol, G., Canfora, G., Casazza, G., and De Lucia, A., "Identifying the Starting Impact Set of a Maintenance Request: A Case Study", in Proceedings of 4th European Conference on Software Maintenance and Reengineering (CSMR'00), Zurich, Switzerland, 29 February - 03 March 2000, pp. 227-231.

[6] Antoniol, G., Canfora, G., Casazza, G., De Lucia, A., and Merlo, E., "Recovering Traceability Links between Code and Documentation", *IEEE Transactions On Software Engineering*, vol. 28, no. 10, October 2002 2002, pp. 970-983.

[7] Asadi, F., Di Penta, M., Antoniol, G., and Gueheneuc, Y.-G., "A Heuristic-based Approach to Identify Concepts in Execution Traces", in Proceedings of 14th

European Conference on Software Maintenance and Reengineering (CSMR'10), Madrid, España, 15-18 March 2010, pp. 31-40.

[8] Bachelli, A., Lanza, M., and Robbes, R., "Linking E-mails and Source Code Artifacts", in Proceedings of 32nd ACM/IEEE International Conference on Software Engineering (ICSE'10), Cape Town, South Africa, 2-8 May 2010, pp. 375-384.

[9] Baeza-Yates, R. and Ribeiro-Neto, B., *Modern Information Retrieval*, Addison Wesley, 1999.

[10] Balasubramanian, N., Kumaran, G., and Carvalho, V. R., "Exploring Reductions for Long Web Queries", in Proceedings of SIGIR, 2010, pp. 571-578.

[11] Bavota, G., De Lucia, A., Marcus, A., and Oliveto, R., "Software Re-Modularization Based on Structural and Semantic Metrics", in Proceedings of 17th IEEE Working Conference on Reverse Engineering (WCRE'10), Beverly, MA, USA, October 13-16 2010, pp. 195-204.

[12] Bavota, G., De Lucia, A., Marcus, A., and Oliveto, R., "Using structural and semantic measures to improve software modularization", *Empirical Software Engineering* 2012, pp. 1-32.

[13] Bavota, G., De Lucia, A., Marcus, A., and Oliveto, R., "Using structural and semantic measures to improve software modularization", *Empirical Software Engineering (EMSE)* 2012.

[14] Bavota, G., De Lucia, A., and Oliveto, R., "Identifying Extract Class refactoring opportunities using structural and semantic cohesion measures", *Journal of Systems and Software*, vol. 84, no. 3, March 2011, pp. 397-414.

[15] Beard, M. D., Kraft, N. A., Etzkorn, L. H., and Lukins, S. K., "Measuring the Accuracy of Information Retrieval Based Bug Localization Techniques", in Proceedings of 18th IEEE International Working Conference on Reverse Engineering (WCRE'11), Limerick, Ireland, 17-20 October 2011, pp. 124-128.

[16] Binkley, D., Feild, H., Lawrie, D., and Pighin, M., "Software Fault Prediction using Language Processing", in Proceedings of 2nd Testing: Academic and Industrial Conference (TAIC-PART), Windsor, United Kingdom, 12-14 September 2007 2007, pp. 99-110.

[17] Blei, D. M., A.Y. Ng, and Jordan, M. I., "Latent Dirichlet Allocation", *Journal of Machine Learning Research*, vol. 3, March 2003 2003, pp. 993-1022.

[18] Bohner, S. and Arnold, R., *Software Change Impact Analysis*, IEEE Computer Society, 1996.

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

[20] Butler, S., Wermelinger, M., Yu, Y., and Sharp, H., "Improving the tokenisation of identifier names", *Lecture Notes in Computer Science*, vol. 6813, 2011, pp. 130 - 154.

[21] Canfora, G. and Cerulo, L., "Impact Analysis by Mining Software and Change Request Repositories", in Proceedings of 11th IEEE International Symposium on Software Metrics (METRICS'05), 2005, pp. 20-29.

[22] Canfora, G. and Cerulo, L., "Fine Grained Indexing of Software Repositories to Support Impact Analysis", in Proceedings of International Workshop on Mining Software Repositories (MSR'06), 2006.

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

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

[25] Cleary, B. and Exton, C., "Assisting Concept Location in Software Comprehension", in Proceedings of 19th Psychology of Programming Workshop, 2007, pp. 42-55.

[26] Cleary, B., Exton, C., Buckley, J., and English, M., "An Empirical Analysis of Information Retrieval based Concept Location Techniques in Software Comprehension", *Empirical Software Engineering*, vol. 14, no. 1, February 2009, pp. 93-130.

[27] Cleland-Huang, J., Czauderna, A., Gibiec, M., and Emenecker, J., "A Machine Learning Approach for Tracing Regulatory Codes to Product Specific Requirements", in Proceedings of 32nd ACM/IEEE International Conference on Software Engineering (ICSE'10), Cape Town, South Africa, 2-8 May 2010, pp. 155-164.

[28] Cohen, J., *Statistical power analysis for the behavioral sciences*, 2nd edition ed., Hillsdale, NJ, Lawrence Earlbaum Associates, 1988.

[29] Corazza, A., Di Martino, S., and Maggio, V., "Linsen: An efficient approach to split identifiers and expand abbreviations", in Proceedings of 28th IEEE International Conference on Software Maintenance (ICSM'12), 2012, pp. 233 – 242.

[30] Corazza, A., Di Martino, S., Maggio, V., and Scanniello, G., "Investigating the use of Lexical Information for Software System Clustering", in Proceedings of 14th

IEEE Conference on Software Maintenance and Reengineering (CSMR'11), Oldenburg, 1-4 March 2011, pp. 35-44.

[31] Cover, T. M. and Thomas, J. A., *Elements of Information Theory*, Wiley-Interscience, 1991.

[32] Cronen-Townsend, S., Zhou, Y., and Croft, W. B., "Predicting query performance", in Proceedings of 25th annual international ACM SIGIR conference on research and development in information retrieval, 2002, pp. 299-306.

[33] Cubranic, D. and Murphy, G. C., "Hipikat: Recommending pertinent software development artifacts", in Proceedings of 25th International Conference on Software Engineering (ICSE'03), 2003, pp. 408-418.

[34] Cubranic, D., Murphy, G. C., Singer, J., and Booth, K. S., "Learning from project history: a case study for software development", in Proceedings of ACM Conference on Computer Supported Cooperative Work, 2004, pp. 82-91.

[35] Cubranic, D., Murphy, G. C., Singer, J., and Booth, K. S., "Hipikat: A Project Memory for Software Development", *IEEE Transactions On Software Engineering*, vol. 31, no. 6, 2005, pp. 446-465.

[36] Davies, S., Roper, M., and Wood, M., "Using bug report similarity to enhance bug localisation", in Proceedings of 19th Working Conference on Reverse Engineering (WCRE'12), 2012, pp. 125-134.

[37] De Lucia, A., Fasano, F., Oliveto, R., and Tortora, G., "Recovering Traceability Links in Software Artefact Management Systems", *ACM Transactions on Software Engineering and Methodology*, vol. 16, no. 4, 2007.

[38] De Lucia, A., Oliveto, R., and Sgueglia, P., "Incremental Approach and User Feedbacks: a Silver Bullet for Traceability Recovery", in Proceedings of IEEE International Conference on Software Maintenance (ICSM'06), 2006, pp. 299-309.

[39] Deerwester, S., Dumais, S., Furnas, G. W., Landauer, T., and Harshman, R., "Indexing by Latent Semantic Analysis", *Journal of the American Society for Information Science*, vol. 41, 1990, pp. 391-407.

[40] Dekhtyar, A., Hayes, J. H., Sundaram, S., Holbrook, A., and Dekhtyar, O., "Technique Integration for Requirements Assessment", in Proceedings of 15th IEEE International Requirements Engineering Conference, 2007, pp. 141-150.

[41] Dit, B., Guerrouj, L., Poshyvanyk, D., and Antoniol, G., "Can Better Identifier Splitting Techniques Help Feature Location?", in Proceedings of 19th IEEE International Conference on Program Comprehension (ICPC'11), Kingston, ON, 22-24 June 2011, pp. 11-20.

[42] 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.

[43] Dit, B., Revelle, M., and Poshyvanyk, D., "Integrating Information Retrieval, Execution and Link Analysis Algorithms to Improve Feature Location in Software", *Empirical Software Engineering (EMSE)*, vol. 18, no. 2, 2013, pp. 277-309.

[44] Enslen, E., Hill, E., Pollock, L., and Vijay-Shanker, K., "Mining Source Code to Automatically Split Identifiers for Software Analysis", in Proceedings of 6th IEEE

Working Conference on Mining Software Repositories (MSR'09), Vancouver, BC, 16-17 May 2009, pp. 71-80.

[45] Frakes, W., "Software Reuse Through Information Retrieval", in Proceedings of 20th Hawaii International Conference On System Sciences (HICSS'87), 1987, pp. 530-535.

[46] 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.

[47] Gay, G., Haiduc, S., Marcus, A., and Menzies, T., "On the Use of Relevance Feedback in IR-Based Concept Location", in Proceedings of 25th IEEE International Conference on Software Maintenance (ICSM'09), Edmonton, AB, 20-26 September 2009, pp. 351-360.

[48] Gethers, M., Dit, B., Kagdi, H., and Poshyvanyk, D., "Integrated impact analysis for managing software changes", in Proceedings of International Conference on Software Engineering (ICSE'12), 2012, pp. 430-440.

[49] Gethers, M., Kagdi, H., Dit, B., and Poshyvanyk, D., "An Adaptive Approach to Impact Analysis from Change Requests to Source Code", in Proceedings of 26th IEEE/ACM International Conference on Automated Software Engineering (ASE'11), Lawrence, KS, 6-10 November 2011, pp. 540-543.

[50] Gibiec, M., Czauderna, A., and Cleland-Huang, J., "Towards Mining Replacement Queries for Hard-to-Retrieve Traces", in Proceedings of 25th IEEE/ACM International Conference On Automated Software Engineering (ASE'10), Antwerp, Belgium, 20-24 September 2010, pp. 245-254.

[51] 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.

[52] Guerrouj, L., Di Penta, M., Antoniol, G., and Gueheneuc, Y.-G., "Tidier: an identifier splitting approach using speech recognition techniques", *Journal of Software Maintenance and Evolution: Research and Practice* 2011.

[53] Haiduc, S., Bavota, G., Marcus, A., Oliveto, R., De Lucia, A., and Menzies, T., "Automatic Query Reformulations for Text Retrieval in Software Engineering", in *35th IEEE/ACM International Conference on Software Engineering (ICSE'13)*, 2013, pp. to appear.

[54] Haiduc, S., Bavota, G., Oliveto, R., De Lucia, A., and Marcus, A., "Automatic Query Performance Assessment during the Retrieval of Software Artifacts", in Proceedings of 27th IEEE/ACM International Conference on Automated Software Engineering (ASE'12), Essen, Germany, September 3-7 2012.

[55] 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 IEEE/ACM International Conference on Software Engineering (ICSE'12), Zurich, Switzerland, 2-9 June 2012, pp. 1273-1276.

[56] Hayes, J. H., Antoniol, G., and Gueheneuc, Y.-G., "PREREQIR: Recovering Pre-Requirements via Cluster Analysis", in Proceedings of 15th Working Conference on Reverse Engineering, 2008, pp. 165-174.

[57] Hayes, J. H., Dekhtyar, A., and Sundaram, S., "Text mining for software engineering: How analyst feedback impacts final results", in Proceedings of International Workshop on Mining Software Repositories, 2005, pp. 1–5.

[58] Helm, R. and Maarek, Y., "Integrating information retrieval and domain specific approaches for browsing and retrieval in object-oriented class libraries", in Proceedings of Object-oriented programming systems, languages, and applications, 1991, pp. 47-61.

[59] Hill, E., Fry, Z. P., Boyd, H., Sridhara, G., Novikova, Y., Pollock, L., and Shanker, V., "AMAP: Automatically Mining Abbreviation Expansions in Programs to Enhance Software Maintenance Tools", in Proceedings of 5th Working Conference on Mining Software Repositories, 2008.

[60] Hill, E., Pollock, L., and Shanker, V., "Exploring the Neighborhood with Dora to Expedite Software Maintenance", in Proceedings of 22nd IEEE/ACM International Conference on Automated Software Engineering (ASE'07), 2007, pp. 14-23.

[61] Hill, E., Pollock, L., and Vijay-Shanker, K., "Automatically Capturing Source Code Context of NL-Queries for Software Maintenance and Reuse", in Proceedings of 31st IEEE International Conference on Software Engineering (ICSE'09), Vancouver, BC, 16-24 May 2009, pp. 232-242.

[62] Hill, E., Pollock, L., and Vijay-Shanker, K., "Investigating How to Effectively Combine Static Concern Location Techniques", in Proceedings of 3rd IEEE/ACM International Workshop on Search-driven Development: Users, Infrastructure, Tools, and Evaluation (SUITE'11), Honolulu, HI, 28 May 2011, pp. 37-40.

[63] Hill, E., Rao, S., and Kak, A., " On the use of stemming for concern location and bug localization in java", in Proceedings of IEEE International Conference on Source Code Analysis and Manipulation (SCAM'12), 2012, pp. 184-193.

[64] Holm, S., "A Simple Sequentially Rejective Multiple Test Procedure", *Scandinavian Journal of Statistics*, vol. 6, no. 2, 1979, pp. 65-70.

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

[66] Kagdi, H., Gethers, M., Poshyvanyk, D., and Collard, M. L., "Blending Conceptual and Evolutionary Couplings to Support Change Impact Analysis in Source Code", in Proceedings of 17th IEEE Working Conference on Reverse Engineering (WCRE'10), Beverly, MA, 13-16 October 2010, pp. 119-128

[67] Krovetz, R., "Viewing Morphology as an Inference Process", in Proceedings of 16th ACM SIGIR Conference, 1993, pp. 191-202.

[68] Kuhn, A., Ducasse, S., and Girba, T., "Semantic Clustering: Identifying Topics in Source Code", *Information and Software Technology*, vol. 49, no. 3, 2007, pp. 230-243.

[69] Lawrie, D., Binkley, D., and Morrell, C., "Normalizing source code vocabulary", in Proceedings of 17th Working Conference on Reverse Engineering (WCRE'10), 2010, pp. 3–12.

[70] Lawrie, D., Feild, H., and Binkley, D., "Extracting Meaning from Abbreviated Identifiers", in Proceedings of 7th IEEE International Working Conference on Source Code Analysis and Manipulation (SCAM'07), 2007, pp. 213-222.

[71] 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.

[72] Linstead, E., Rigor, P., Bajracharya, S., Lopes, C., and Baldi, P., "Mining concepts from code with probabilistic topic models", in Proceedings of 22nd IEEE/ACM International Conference on Automated Software Engineering (ASE'07), 2007, pp. 461-464.

[73] Liu, D., Marcus, A., Poshyvanyk, D., and Rajlich, V., "Feature Location via Information Retrieval based Filtering of a Single Scenario Execution Trace", in Proceedings of 22nd IEEE/ACM International Conference on Automated Software Engineering (ASE'07), 2007, pp. 234-243.

[74] Lormans, M. and Van Deursen, A., "Can LSI help Reconstructing Requirements Traceability in Design and Test?", in Proceedings of 10th European Conference on Software Maintenance and Reengineering (CSMR'06), 2006, pp. 47-56.

[75] Lu, X. A. and Keefer, R. B., "Query expansion/reduction and its impact on retrieval effectiveness", *NIST SPecial Publication SP*, vol. 225, no. 500, 1995, pp. 231-239.

[76] Lukins, S. K., Kraft, N. A., and Etzkorn, L. H., "Source Code Retrieval for Bug Localization Using Latent Dirichlet Allocation", in Proceedings of 15th Working Conference on Reverse Engineering, 2008, pp. 155-164.

[77] Lukins, S. K., Kraft, N. A., and Etzkorn, L. H., "Bug Localization using Latent Dirichlet Allocation", *Information and Software Technology*, vol. 52, no. 9, September 2010, pp. 972-990.

[78] Maarek, Y. S., Berry, D. M., and Kaiser, G. E., "An Information Retrieval Approach for Automatically Constructing Software Libraries", *IEEE Transactions On Software Engineering*, vol. 17, no. 8, 1991, pp. 800-813.

[79] Maarek, Y. S. and Smadja, F. A., "Full Text Indexing Based on Lexical Relations, an Application: Software Libraries", in Proceedings of SIGIR89, 1989, pp. 198-206.

[80] Maletic, J. and Valluri, N., "Automatic Software Clustering via Latent Semantic Analysis", in Proceedings of 14th IEEE International Conference on Automated Software Engineering (ASE'99), 1999, pp. 251-254.

[81] Manning, C. D., Raghavan, P., and Schütze, H., *Introduction to Information Retrieval*, Cambridge University Press, 2008.

[82] Marcus, A. and Maletic, J., "Identification of High-Level Concept Clones in Source Code", in Proceedings of Automated Software Engineering (ASE'01), 2001, pp. 107-114.

[83] 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 (ICSE'03), 2003, pp. 125-135.

[84] Marcus, A., Maletic, J. I., and Sergeyev, A., "Recovery of Traceability Links Between Software Documentation and Source Code", *International Journal of Software Engineering and Knowledge Engineering*, vol. 15, no. 5, 2005, pp. 811-836.

[85] Marcus, A., Poshyvanyk, D., and Ferenc, R., "Using the Conceptual Cohesion of Classes for Fault Prediction in Object Oriented Systems", *IEEE Transactions On Software Engineering*, vol. 34, no. 2, 2008, pp. 287-300.

[86] Marcus, A., Rajlich, V., Buchta, J., Petrenko, M., and Sergeyev, A., "Static Techniques for Concept Location in Object-Oriented Code", in Proceedings of 13th IEEE International Workshop on Program Comprehension (IWPC'05), St. Louis, MO, USA, May 15-16 2005, pp. 33-42.

[87] Marcus, A., Sergeyev, A., Rajlich, V., and Maletic, J. I., "An Information Retrieval Approach to Concept Location in Source Code", in Proceedings of 11th IEEE Working Conference on Reverse Engineering (WCRE'04), Delft, The Netherlands, November 9-12 2004, pp. 214-223.

[88] McMillan, C., Poshyvanyk, D., and Revelle, M., "Combining Textual and Structural Analysis of Software Artifacts for Traceability Link Recover", in Proceedings of ICSE Workshop on Traceability in Emerging Forms of Software Engineering (TEFSE'09), Vancouver, BC, 18-18 May 2009, pp. 41-48.

[89] Medini, S., Galinier, P., Di Penta, M., Gueheneuc, Y.-G., and Antoniol, G., "A Fast Algorithm to Locate Concepts in Execution Traces", in Proceedings of 3rd

International Symposium on Search Based Software Engineering (SSBSE'11), Szeged, Hungary, 10-12 September 2011, pp. 252-266.

[90] Michail, A. and Notkin, D., "Assessing software libraries by browsing similar classes, functions and relationships", in Proceedings of IEEE International Conference on Software Engineering (ICSE'99), 1999, pp. 463-472.

[91] Moldovan, G. and Serban, G., "Aspect Mining using a Vector-Space Model Based Clustering Approach", in Proceedings of Workshop on Linking Aspect Technology and Evolution (LATE'06), 2006.

[92] Nguyen, A. T., Nguyen, T. T., Al-Kofahi, J., Nguyen, H. V., and Nguyen, T. N., "A Topic-based Approach for Narrowing the Search Space of Buggy Files from a Bug Report", in Proceedings of 26th IEEE/ACM International Conference On Automated Software Engineering, Oread, Lawrence, Kansas, 2011, pp. 263-272.

[93] Nichols, B. D., "Augmented Bug Localization Using Past Bug Information", in Proceedings of 48th ACM Annual Southeast Regional Conference (ACMSE'10), Oxford, MS, 15-17 April 2010, pp. 1-6.

[94] Oliveto, R., Gethers, M., Bavota, G., Poshyvanyk, D., and De Lucia, A., "Identifying Method Friendships to Remove the Feature Envy Bad Smell", in Proceedings of 33rd IEEE/ACM International Conference on Software Engineering (ICSE'11), NIER Track, Waikiki, Honolulu, HI, 21-28 May 2011, pp. 820-823.

[95] Petrenko, M., Rajlich, V., and Vanciu, R., "Partial Domain Comprehension in Software Evolution and Maintenance", in Proceedings of 16th IEEE International Conference on Program Comprehension (ICPC'08), 2008, pp. 13-22.

[96] Ponte, J. M. and Croft, W. B., "A Language Modeling Approach to Information Retrieval", in Proceedings of 21st Annual International ACM SIGIR Conference on Research and Development in Information Retrieval, 1998, pp. 275–281.

[97] Porter, M., "An Algorithm for Suffix Stripping", *Program*, vol. 14, no. 3, 1980, pp. 130-137.

[98] Poshyvanyk, D., Gethers, M., and Marcus, A., "Concept Location using Formal Concept Analysis and Information Retrieval", *ACM Transactions on Software Engineering and Methodology (TOSEM)*, vol. 21, no. 4, November 2012 2012, pp. to appear.

[99] Poshyvanyk, D., Gueheneuc, Y. G., Marcus, A., Antoniol, G., and Rajlich, V., "Feature Location using Probabilistic Ranking of Methods based on Execution Scenarios and Information Retrieval", *IEEE Transactions On Software Engineering (TSE)*, vol. 33, no. 6, 2007, pp. 420-432.

[100] Poshyvanyk, D. and Marcus, A., "The Conceptual Coupling Metrics for Object-Oriented Systems", in Proceedings of 22nd IEEE International Conference on Software Maintenance (ICSM'06), 2006.

[101] Poshyvanyk, D. and Marcus, A., "Combining Formal Concept Analysis with Information Retrieval for Concept Location in Source Code", in Proceedings of 15th IEEE International Conference on Program Comprehension (ICPC'07), Banff, Alberta, Canada, June 26-29 2007, pp. 37-46.

[102] Poshyvanyk, D., Marcus, A., Ferenc, R., and Gyimothy, T., "Using Information Retrieval based Coupling Measures for Impact Analysis", *Empirical Software Engineering*, vol. 14, no. 1, February 2009, pp. 5-32.

[103] Qusef, A., Bavota, G., Oliveto, R., De Lucia, A., and Binkley, D., "SCOTCH: Test-to-Code Traceability using Slicing and Conceptual Coupling", in Proceedings of 27th IEEE International Conference on Software Maintenance (ICSM'11), Williamsburg, VI, 25-30 September 2011, pp. 63-72.

[104] Rajlich, V., *Software Engineering: The Current Practice*, Boca Raton, FL, CRC Press, 2012.

[105] Rajlich, V. and Gosavi, P., "Incremental Change in Object-Oriented Programming", *IEEE Software*, vol. 21, no. 4, 2004, pp. 62-69.

[106] Rajlich, V., Marchesi, M., Succi, G., Wells, D., and Williams, L., "A Methodology for Incremental Change", in *Extreme Programming Perspectives*, Addison Wesley, 2002, pp. 201-214.

[107] Rao, S. and Kak, A., "Retrieval from Software Libraries for Bug Localization: a Comparative Study of Generic and Composite Text Models", in Proceedings of 8th IEEE/ACM Working Conference on Mining software repositories (MSR'11), Wikiki, Honolulu, WI, 21-28 May 2011, pp. 43-52.

[108] Ratanotayanon, S., Choi, H. J., and Sim, S. E., "My Repository Runneth Over: An Empirical Study on Diversifying Data Sources to Improve Feature Search", in Proceedings of 18th IEEE International Conference on Program Comprehension, Braga, Minho, June 30 -July 2 2010, pp. 206-305.

[109] Revelle, M. and Poshyvanyk, D., "An Exploratory Study on Assessing Feature Location Techniques", in Proceedings of 17th IEEE International Conference on Program Comprehension (ICPC'09), Vancouver, British Columbia, Canada, 17-19 May 2009, pp. 218-222.

[110] Robillard, M. P., "Topology Analysis of Software Dependencies", *ACM Transactions on Software Engineering and Methodology*, vol. 17, no. 4, 2008.

[111] Rocchio, J. J., "Relevance feedback in information retrieval", in *The SMART Retrieval System - Experiments in Automatic Document Processing*, Prentice Hall, 1971, pp. 313-323.

[112] Salton, G., Wong, A., and Yang, C. S., "A vector space model for automatic indexing", *Commun. ACM*, vol. 18, no. 11, 1975, pp. 613–620.

[113] Savage, T., Dit, B., Gethers, M., and Poshyvanyk, D., "TopicXP: Exploring Topics in Source Code using Latent Dirichlet Allocation", in Proceedings of 26th IEEE International Conference on Software Maintenance (ICSM'10), Timisoara, Romania, 12-18 September 2010, pp. 1-6.

[114] Scanniello, G. and Marcus, A., "Clustering Support for Static Concept Location in Source Code", in Proceedings of 19th IEEE International Conference on Program Comprehension (ICPC'11), Kingston, ON, June 22-24 2011, pp. 1-10.

[115] Settimi, R., Huang, C., Khadra, B., Mody, J., Lukasik, W., and DePalma, C., "Supporting Software Evolution through Dynamically Retrieving Traces to UML Artifacts", in Proceedings of 7th International Workshop on Principles of Software Evolution (IWPSE), 2004, pp. 49-54.

[116] Shao, P. and Smith, R. K., "Feature Location by IR Modules and Call Graph", in Proceedings of 47th ACM Annual Southeast Regional Conference (ACM-SE'09), Clemson, SC, 19-21 March 2009.

[117] Shepherd, D., Fry, Z., Gibson, E., Pollock, L., and Vijay-Shanker, K., "Using Natural Language Program Analysis to Locate and Understand Action-Oriented

Concerns", in Proceedings of International Conference on Aspect Oriented Software Development (AOSD'07), 2007, pp. 212-224.

[118] Sridhara, G., Hill, E., Pollock, L., and Vijay-Shanker, K., "Identifying Word Relations in Software: A Comparative Study of Semantic Similarity Tools", in Proceedings of 16th IEEE International Conference on Program Comprehension (ICPC'08), Amsterdam, 10-13 June 2008, pp. 123-132.

[119] Starke, J., Luce, C., and Sillito, J., "Searching and Skimming: An Exploratory Study", in Proceedings of 25th IEEE International Conference on Software Maintenance (ICSM'09), Edmonton, AB, 20-26 Sept. 2009, pp. 157-166.

[120] Tairas, R. and Gray, J., "An Information Retrieval Process to Aid in the Analysis of Code Clones", *Empirical Software Engineering*, vol. 14, no. 1, February 2009, pp. 33-56.

[121] Tian, Y., Lo, D., and Sun, C., "Information Retrieval Based Nearest Neighbor Classification for Fine-Grained Bug Severity Prediction", in Proceedings of 19th Working Conference on Reverse Engineering (WCRE'12), 2012, pp. 214-225.

[122] Vercoustre, A.-M., Pehcevski, J., and Naumovski, V., "Topic Difficulty Prediction in Entity Ranking", in Proceedings of 7th International Workshop of the Initiative for the Evaluation of XML Retrieval (INEX 2008), Dagstuhl Castle, Germany, 15-18 December 2008, pp. 280-291.

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

[124] Wang, S., Lo, D., Xing, Z., and Jiang, L., "Concern Localization using Information Retrieval: An Empirical Study on Linux Kernel", in Proceedings of 18th Working Conference on Reverse Engineering, 2011, pp. 92-96.

[125] Xue, X., Huston, S., and Croft, W. B., "Improving Verbose Queries using Subset Distribution", in Proceedings of ACM International Conference on Information and Knowledge Management, 2010.

[126] Yadla, S., Hayes, H., and Dekhtyar, A., "Tracing Requirements to Defect Reports: An Application of Information Retrieval Techniques", *Innovations in Systems and Software Engineering: A NASA Journal*, vol. 1, no. 2, 2005.

[127] Yang, J. and Tan, L., "Inferring Semantically Related Words from Software Context", in Proceedings of 9th IEEE/ACM Working Conference on Mining Software Repositories (MSR'12), Zurich, Switzerland, 2-3 June 2012, pp. 161-170.

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

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

[130] Zhao, W., Zhang, L., Liu, Y., Sun, J., and Yang, F., "SNIAFL: towards a static non-interactive approach to feature location", in Proceedings of 26th International Conference on Software Engineering (ICSE'04), 2004, pp. 293-303.

[131] Zhao, W., Zhang, L., Liu, Y., Sun, J., and Yang, F., "SNIAFL: Towards a Static Non-interactive Approach to Feature Location", *ACM Transactions on Software Engineering and Methodologies (TOSEM)*, vol. 15, no. 2, 2006, pp. 195-226.

[132] Zhou, J., Zhang, H., and Lo, D., "Where Should the Bugs Be Fixed? More Accurate Information Retrieval-Based Bug Localization Based on Bug Reports", in Proceedings of 34th International Conference on Software Engineering (ICSE'12), Zurich, Switzerland, 2012, pp. 14-24.

ABSTRACT

SUPPORTING TEXT RETRIEVAL QUERY FORMULATION IN SOFTWARE ENGINEERING

by

SONIA HAIDUC

August 2013

Advisor: Dr. Andrian Marcus

Major: Computer Science

Degree: Doctor of Philosophy

The text found in software artifacts captures important information. Text Retrieval (TR) techniques have been successfully used to leverage this information. Despite their advantages, the success of TR techniques strongly depends on the textual queries given as input. When poorly chosen queries are used, developers can waste time investigating irrelevant results.

The quality of a query indicates the relevance of the results returned by TR in response to the query and can give an indication if the results are worth investigating or a reformulation of the query should be sought instead. Knowing the quality of the query could lead to time saved when irrelevant results are returned. However, the only way to determine if a query led to the wanted artifacts is by manually inspecting the list of results. This dissertation introduces novel approaches to measure and predict the quality of queries automatically in the context of software engineering tasks, based on a set of statistical properties of the queries. The approaches are evaluated for the task of concept location in source code. The results reveal that the proposed approaches are

able to accurately capture and predict the quality of queries for software engineering tasks supported by TR.

When a query has low quality, the developer can reformulate it and improve it. However, this is just as hard as formulating the query in the first place. This dissertation presents two approaches for partial and complete automation of the query reformulation process. The semi-automatic approach relies on developer feedback about the relevance of TR results and uses this information to automatically reformulate the query. The automatic approach learns and applies the best reformulation approach for a query and relies on a set of training queries and their statistical properties to achieve this. Both approaches are evaluated for concept location and the results show that the techniques are able to improve the results of the original queries in the majority of the cases.

We expect that on the long run the proposed approaches will contribute directly to the reduction of developer effort and implicitly the reduction of software evolution costs.

AUTOBIOGRAPHICAL STATEMENT

Sonia Haiduc is a Ph.D. candidate in the Department of Computer Science at Wayne State University in Detroit. She received her M.A. in Computer Science from Wayne State University in 2009. Before coming to Wayne State University, she received her B.Sc. from Babes-Bolyai University in Cluj-Napoca, Romania.

Her research interests are in software engineering with focus on software maintenance and evolution. She has published multiple research papers in highly selective venues, including IEEE/ACM International Conference on Software Engineering, IEEE/ACM International Conference on Automated Software Engineering, IEEE International Conference on Software Maintenance, IEEE International Conference on Program Comprehension, IEEE Working Conference on Reverse Engineering, and IEEE European Conference on Software Maintenance. She has been a member of the organizing and program committee for several conferences in the field. She has also been awarded the 2011 Google Anita Borg Memorial Scholarship for her research and leadership.