Fall 2013

Source Code Retrieval from Large Software Libraries for Automatic Bug Localization

Bunyamin Sisman
Purdue University

Graduate School ETD Form 9
(Revised 12/07)

PURDUE UNIVERSITY
GRADUATE SCHOOL
Thesis/Dissertation Acceptance

This is to certify that the thesis/dissertation prepared

By Bunyamin Sisman

Entitled
Source Code Retrieval from Large Software Libraries for Automatic Bug Localization

For the degree of Doctor of Philosophy

Is approved by the final examining committee:

AVINASH C. KAK
Chair

JOHNNY PARK

MIREILLE BOUTIN

SERGEY KIRSHNER

To the best of my knowledge and as understood by the student in the *Research Integrity and Copyright Disclaimer (Graduate School Form 20)*, this thesis/dissertation adheres to the provisions of Purdue University's "Policy on Integrity in Research" and the use of copyrighted material.

Approved by Major Professor(s): AVINASH C. KAK

Approved by: M. R. Melloch 08-20-2013
Head of the Graduate Program Date

SOURCE CODE RETRIEVAL FROM LARGE SOFTWARE LIBRARIES FOR AUTOMATIC BUG LOCALIZATION

A Dissertation

Submitted to the Faculty

of

Purdue University

by

Bunyamin Sisman

In Partial Fulfillment of the

Requirements for the Degree

of

Doctor of Philosophy

December 2013

Purdue University

West Lafayette, Indiana

ACKNOWLEDGMENTS

I wish to thank various people for their contribution to this project. I would like to express my great appreciation to all the members of the Robot Vision Lab for their constructive suggestions and useful critiques. I would like to thank Shivani Rao, Gaurav Srivastava, Chad Aeschliman, Josiah Yoder and Khalil Yousef for their valuable comments and for sharing their experiences with me.

Many thanks to Prof. Sergey Kirshner for his suggestions on our Bayesian framework for source code retrieval and on exploring Conditional Markov Random Fields for bug localization. I would also like to extend my thanks to Prof. Johnny Park for his supportive comments throughout the course of my PhD.

Special thanks should be given to Prof. Avinash C. Kak, my major professor, for his patient guidance, enthusiastic encouragement and useful critiques of this research. I would also like to acknowledge Ministry of National Education of Turkey and Infosys for funding this project at Purdue.

Finally, I wish to thank my family and my friends for their support and encouragement throughout my study. I thank my wife Zubeyde for her love and support during this long journey.

TABLE OF CONTENTS

Page

Page

LIST OF TABLES

Table Page

LIST OF FIGURES

Figure Page

ABSTRACT

Sisman, Bunyamin Ph.D., Purdue University, December 2013. Source Code Retrieval from Large Software Libraries for Automatic Bug Localization. Major Professor: Avinash C. Kak.

This dissertation advances the state-of-the-art in information retrieval (IR) based approaches to automatic bug localization in software. In an IR-based approach, one first creates a search engine using a probabilistic or a deterministic model for the files in a software library. Subsequently, a bug report is treated as a query to the search engine for retrieving the files relevant to the bug.

With regard to the new work presented, we first demonstrate the importance of taking version histories of the files into account for achieving significant improvements in the precision with which the files related to a bug are located. This is motivated by the realization that the files that have not changed in a long time are likely to have "stabilized" and are therefore less likely to contain bugs. Subsequently, we look at the difficulties created by the fact that developers frequently use abbreviations and concatenations that are not likely to be familiar to someone trying to locate the files related to a bug. We show how an initial query can be automatically reformulated to include the relevant actual terms in the files by an analysis of the files retrieved in response to the original query for terms that are proximal to the original query terms. The last part of this dissertation generalizes our term-proximity based work by using Markov Random Fields (MRF) to model the inter-term dependencies in a query vis-a-vis the files. Our MRF work redresses one of the major defects of the most commonly used modeling approaches in IR, which is the loss of all inter-term relationships in the documents.

1. INTRODUCTION

Developing a large software system is a complex process requiring a long time and tremendous effort from the developers in terms of design, implementation, testing, quality assurance, deployment and maintenance. It is, nowadays, not uncommon to have a large number of developers, possibly geographically distributed, to work on the development of a single software system for many years. In order to manage this intricate development process and to ensure the quality of software, there is an increasing need for sophisticated tools that are tailored for the specific needs of the developers.

Among all the software engineering tasks, searching the code base of a project to locate the software artifacts relevant to an information need is perhaps one of the most prominent activities performed in large projects. The tasks for which code search is carried out include concept and bug localization, change impact analysis, traceability link recovery, and so on. Over the years, researchers proposed many source code retrieval tools to perform these tasks efficiently [1–7]. Retrieval for the localization of the buggy parts of a software project particularly received much attention from the research community as it is an expensive maintenance task repeated many times during the life-cycle of a project [8].

The focus of this dissertation is on Bug Localization (BL) using source code retrieval. We present all the components of a new retrieval engine for locating the relevant software artifacts in the code base of a software project with respect to the reported bugs. The underlying retrieval mechanism responds to the textual queries which may be either designed by a user or obtained more automatically by using the textual information in a given bug report. The goal is to rank the files in the target code base in such a way that the highest ranked files will be those that are relevant to the given bug. For this purpose, we first propose a Bayesian framework

for source code retrieval that incorporates the prior knowledge on the bug likelihoods of the software artifacts as stored in the Software Configuration Management (SCM) systems. Next we describe an automatic Query Reformulation (QR) model based on Pseudo-Relevance Feedback that takes into account the Spatial Code Proximity (SCP) in the software artifacts. Finally, we present a Markov Random Field (MRF) based approach that exploits SCP directly in the retrievals for improved BL accuracy.

1.1 Traditional Bug Localization Methods

Traditional methods for source code retrieval for the purpose of BL include an analysis of either the dynamic or the static properties of software [9, 10]. While dynamic approaches rely on passing and failing executions of a set of test cases to locate the parts of the program causing the bug, static approaches do not require execution of the program and aim at leveraging the static properties or interdependencies to locate bugs. The main deficiency of the dynamic approaches is that designing an exhaustive set of test cases that could effectively be used to reveal defective behaviors is very difficult and expensive. Static properties of a software project as derived from, say, call graphs, on the other hand, tend to be language specific. The static and dynamic approaches also are not able to take into account non-executable files, such as configuration and documentation files, in a code base that can also be a source of bugs in modern software.

In comparison with the dynamic and static approaches of the sort mentioned above, the proposed IR framework to bug localization in this dissertation may be found preferable in certain software development contexts because it can be used in interactive modes for the refinement of the retrieved results. That is, if a developer is not satisfied with what is retrieved with a current query (because, say, the retrieved set is much too large to be of much use), the developer has the option of reformulating his/her query in order to make the next retrieval more focused. The set of artifacts retrieved in response to the current query can often give clues as to how to reformulate

one's query for better results the next time. Being able to query the source code flexibly in this manner can only be expected to increase programmers' understanding and knowledge on the software, which, in turn, is the key to efficient bug localization.

1.2 Information Retrieval for Bug Localization

The main goal of a document retrieval system is to return a ranked list of the documents in a corpus in response to a query in such a way that the top ranked documents are more relevant to the query than the documents in the lower ranks. This is usually achieved by assigning a retrieval score to each document in the corpus. The higher the score, the higher the relevancy of a particular document to the information need. Obviously, the ranking of the documents in such a framework depends on the notion of the relevancy that is formulated by the underlying scoring algorithm. In order to accurately retrieve the set of relevant files from a large collection of documents, this scoring algorithm should capture the human notion of relevancy for the given application.

Although there has been a tremendous amount of work that has perfected and commercialized document retrieval, there are still many challenges waiting to be addressed in its particular applications. One such problem is to find the optimal model for a given retrieval task that can be scaled up to work on very large, evolving datasets and at the same time learn from the feedback information to capture the human notion of relevancy [11]. This dissertation presents such a retrieval framework that targets BL in large, ever-changing software systems.

In the early days of Information Retrieval (IR), document retrieval has taken place solely based on the presence of the query words in the documents. In this retrieval model, named Standard Boolean Model (SBM), the queries are formed by combining a set of words with logical operators. The documents that satisfy the logical expression of the query are then retrieved and shown to the user. However this model lacks the notion of ranking, an essential prerequisite for specifying the degree of relevancy

of the documents to the queries. In order to rank the set of documents in a given corpus, Vector Space Model (VSM) represents each document with a vector in which each element stores the frequency of the corresponding word in the document. VSM makes it possible to rank the documents for retrieval on the basis of their similarity to the query which is also represented as a vector of words. While VSM provides a theoretically sound platform for document retrieval, many competitive probabilistic models have been proposed over the years for a better modeling of the documents that would eventually improve the retrieval accuracy among other tasks.

In general, for large datasets, retrieval models are founded on the "bag-of-words" assumption which states that each word is sampled independently from the rest of the words in the document or in the collection. This assumption has been popular in the IR community since capturing term dependencies is intractable for large set of documents. Two commonly used document retrieval techniques that employ the bag-of-words assumption are the Language Modeling (LM) — a probabilistic framework and the Divergence From Randomness (DFR) — an information theoretic approach to document retrieval. A language model with the bag-of-word assumption is referred to as a Unigram Model, which has been commonly used in many retrieval tasks successfully [12, 13]. On the other hand, the information theoretic approaches that use the bag-of-word assumption, such as TF-IDF[1], remained competitive in document retrieval [14]. Although ignoring the structure in a document may seem too simplistic, these retrieval mechanisms perform well for text retrieval in practice. With the increased processing power, superior retrieval models that relax the bag-of-words assumption to some extent have also been proposed over the years [15].

Employing the document retrieval models mentioned above for BL brings about several issues. An important aspect of software development to mention in that regard is the ever-changing nature of software. As new features are added or bugs get fixed, the textual content of a software library is incrementally modified and each set of modifications is stored via an SCM tool. These prior set of changes is one of

[1]Term Frequency - Inverse Document Frequency

the primary sources of information to predict the potential locations of the future bugs [16]. Therefore, a retrieval system that is oblivious to the change history of the software artifacts may result in poor accuracies for BL.

Another aspect of the code search in software development is the quality of the queries for retrieval [17–19]. Constructing a query that can successfully distinguish the relevant documents from the irrelevant ones is especially challenging in source code retrieval. This is because, despite the naming conventions in all Programming Languages (PL), arbitrary abbreviations and concatenations are commonly used by programmers when creating software constructs such as variables, methods, class names etc. Searching the code base of a project without being aware of these peculiarities is not expected to yield satisfactory results.

Beside these issues that should be considered carefully when IR algorithms are employed for BL, modeling the term dependencies in queries and in source code also carries high importance in locating the software artifacts relevant to the information need [19]. Especially, when long queries comprising structural elements such as a textual narrative, a stack trace, one or more snippets of code etc. are used in the retrievals [20, 21], the BL accuracy may deteriorate if the proximity and order relationships between the terms of the query are not taken into account. In order to model the spatial term dependencies in source code, we employ Markov Random Fields (MRF).

While MRF is a powerful approach to the modeling of query-document relationship, to fully exploit its potential for long queries e.g. bug reports, it must be used in conjunction with what we refer to as *Query Conditioning* (QC). The main idea behind QC is that for a given set of terms, the inter-term relationships should not carry the same weight in the different parts of the query for obvious reasons. For example, the proximity of the terms used in the stack trace portions of a bug report should carry far more weight than that in the textual narrative. In other words, the ordering and proximity constraints are likely to be far more discriminative in those portions of the bug report that, by their very nature, are far more structured.

In this dissertation, we address all these aspects of source code retrieval for BL with *Terrier+*, our source code retrieval engine for BL.

1.3 Organization of the Dissertation

This dissertation is organized in 9 chapters. In the next chapter, we overview the related work. Chapter 3 describes the retrieval models prominent in the Natural Language (NL) document modeling. The retrieval models we employ for Bug Localization (BL) are explained in Chapter 4. Our Bayesian framework that takes into account the version histories of the software artifacts during BL is presented in Chapter 5. Chapter 6 presents our automatic QR framework to assist code search for BL. In Chapter 7, we describe the MRF modeling of the query-document relationships in source code along with the proposed Query Conditioning (QC) framework. In Chapter 8, we present the main components of our retrieval engine for Bug Localization: Terrier+. We present our conclusions along with the summary of the achievements in Chapter 9.

2. RELATED WORK

Concept location, Bug localization and feature/concern localization are closely related problems. In this chapter, we will overview the respectable approaches developed for these problems from a broad perspective.

2.1 Dynamic and Static Methods

Dynamic techniques depend on a set of passing and failing test cases to determine the possible locations of the bugs [22–31]. This type of bug localization is also referred to as spectrum or coverage based fault localization as it is based on the parts of the code base that get executed when a set of test cases are run.

In [32], Jones et al. proposed a dynamic bug localization technique that leverages visualization to indicate the bug likelihoods of the suspicious statements based on a color mapping. For this purpose, they present a bug localization tool, called *Tarantula*, which associates colors with the statements in such a way that the less suspicious statements that are invoked by fewer failing test cases appear more green; while the statements that are invoked by many failing cases appear more red.

In [9], Dallmeier et al. proposed AMPLE, a lightweight bug localization technique that compares the call sequences of the passing and the failing test cases. The main motivation behind AMPLE is that faults correlate with the differences in the call sequences of the passing and failing test cases. In order to localize faults, AMPLE compares the call sequence of a single failing test to the call sequences of many passing tests. If the methods of a class are primarily called when the failing test is run then the class is likely the cause of the bug. Such classes are ranked higher in a ranked list of suspected classes which are later examined by the programmer until the fault is located. AMPLE was implemented as an Eclipse plug-in.

Liblit et al. [29] showed that dynamic bug localization techniques perform poorly when there are multiple bugs in a program. By identifying the effect of individual bugs, their method reaches better accuracies with fewer program executions. In [27], Lie et al. proposed SOBER, another dynamic bug localization tool that uses statistical properties of the passing and the failing executions. They reported superior bug localization accuracies over the work reported by Liblit et al. [29] by considering both correct and incorrect runs during the evaluation of the predicates.

Another important class of bug localization techniques are grouped under static analysis. As the name implies, static bug localization techniques depend on the static properties of software such as class structure, inheritance, dependency graphs and so on [10, 33, 34]. One such study is reported by Hovemeyer et al. [10]. They proposed FindBugs, a static bug localization tool that uses bug pattern detectors to locate bugs.

In [33], Robillard proposed a navigation tool to assist developers with finding relevant parts of a software system by analyzing the topology of the structural dependencies. This method takes a set of program elements such as methods or fields as input and recommends a set of program elements worthy of exploration while sifting out the less interesting ones.

While static properties are important in software systems, the techniques that only leverage the static dependencies tend to return many false positives [7]. Therefore static techniques are commonly used in conjunction with IR or dynamic methods.

2.2 IR Methods

Early work on using text retrieval methods in source code retrieval includes the work by Marcus et al. [1]. They used Latent Semantic Indexing (LSI) to retrieve the software artifacts with respect to short queries. The retrievals are performed in the lower dimensional LSI space which assigns greater importance to the terms that

frequently co-occur in the source files. This framework can also be used to expand a given initial query that consists of a single query term initially.

In [35], Emily et al. leveraged source code identifiers to automatically extract relevant phrases to a given initial query. These phrases are then used to either find the relevant program elements or to manually reformulate the query for superior feature/concern localization. In [36], they also investigated the effect of the position of a query term on the accuracy of the search results. The main idea behind this study is that the location of a query term in the method signatures and in the method bodies determines its importance in the search process.

Several studies focused on the retrieval of the software artifacts with IR methods for bug localization. In a comparative study, Rao and Kak evaluated the generic and the composite Information Retrieval (IR) techniques that leverage the textual content of the bug reports to localize the files that should be modified to resolve bugs [37]. Along the same lines, Lukins et al. [2] used Latent Dirichlet Allocation (LDA) for automatic bug localization. Their comparative study shows that LDA performs at least as good as Latent Semantic Indexing (LSI), the competing deterministic topic model for text retrieval. Nguyen et al. [38] also proposed an LDA-based approach to narrow down the search space for improving bug localization accuracy. This method also takes into account the prior bug-proneness of the software files to enhance the accuracy of the retrievals.

In [21], Ashok et al. have shown how relationship graphs can be used to retrieve source files and prior bugs in response to what they refer to as "fat queries" that consist of structured and unstructured data. Another bug localization tool was proposed by Zhou et al. [39]. Their BugLocator tool, uses the textual similarities between a given bug report and the prior bug reports to enhance the bug localization accuracy. The main motivation behind BugLocator is that the same files tend to get fixed for similar bug reports during the life-cycle of a software project. Therefore the files modified in the past for similar bugs are more likely to be relevant to a given bug

report. They reported that the retrieval accuracies increase significantly when the prior bug reports are taken into account.

In [40], Gay et al. used Explicit Relevance Feedback for Query Reformulation (QR) for the purpose of concept location. This framework requires developers to engage in an iterative query/answer session with the search engine. At each iteration, the developer is expected to judge the relevancy of the returned results vis-á-vis the current query. Based on these judgments, the query is reformulated with the Rocchio's formula [41] and resubmitted to obtain the next round of retrieval results. This process is repeated until the target file is located by the developer or the developer gives up.

In [42], Haiduc et al. introduced *Refocus*, an automatic QR tool for text retrieval in software engineering. Refocus automatically reformulates a given query by choosing the best QR technique which is determined by training a decision tree on a separate query set and their retrieval results. After training, based on the statistics of the given query, the decision tree recommends an automatic query reformulation technique that is expected to perform the best among the others.

2.3 Hybrid Methods

Beside the dynamic, static and IR methods, researchers also proposed many hybrid approaches that combines these methods [43–46]. The main idea behind combining dynamic and static analysis is to narrow down the search space with dynamic approaches and then use static analysis on this smaller search space for better accuracy. One such work is reported by Eisenbarth et al. [25, 47]. Their approach clusters the execution traces extracted by dynamic analysis then these traces are used in concept analysis. Along the same lines, Antoniol and Guéhéneuc introduced Scenario-based Probabilistic Ranking (SPR) [45]. This approach assigns two probabilities to the methods in the execution traces: one indicates the probability of the method to ex-

ercise the feature and the other not to. Based on these probabilities, the method is classified as either relevant or irrelevant.

In [45], Poshyvanyk et al. introduced The Probabilistic Ranking of Methods based on Execution Scenarios and Information Retrieval (PROMESIR), a feature location technique that uses both dynamic analysis and IR. This approach first extract a set of candidate methods with dynamic analysis and then uses a query created by an expert to find the relevant methods. Along the same lines, in [48], Poshyvanyk et al. presented a hybrid framework that combines SPR with Latent Semantic Indexing (LSI) for feature location. They showed that the composition of these methods performs better than the individual methods without requiring the user to have an extensive knowledge about the code base. Another hybrid approach again reported by Poshyvanyk et al. [49, 50] extended this approach to include formal concept analysis (FCA). They showed that the irrelevant search results returned by the LSI model can be reduced with (FCA).

3. MODELS FOR DOCUMENT RETRIEVAL

Over the years, many document retrieval algorithms have been developed to solve various problems with implementations ranging from simple command line tools to large scale web retrieval engines. In this chapter, we will briefly overview the models commonly used in Natural Language (NL) document retrieval and discuss their advantages and disadvantages.

3.1 Standard Boolean Model

The Standard Boolean Model (SBM) is one of the earliest and commonly used retrieval model for textual data. With SBM, document retrieval is performed on the basis of the presence of the query words in the documents. This is accomplished with the use of the boolean logic and the set theory. Since the SBM model does not indicate any word being more important than the others, the three boolean operators AND, OR, and NOT are sufficient to construct any logical statement over the query words. After the desired logical expression is obtained, it can be used to match the documents in a corpus for retrieval [51].

Table 3.1 The notation used for SBM

$\mathcal{F}$	Retrieval function
$\mathcal{B}$	Boolean algebra over sets of words
Q	A boolean statement of a query
d	Binary representation of a document in the corpus
q_i	Binary representation of ith word in a query

$$\mathcal{F}(Q,d) = \begin{cases} 1 & \text{if } \mathcal{B}(Q,d) = true, \\ 0 & \text{if } \mathcal{B}(Q,d) = false. \end{cases} \tag{3.1}$$

With the retrieval function $\mathcal{F}$ given above, all the documents in the corpus of software library for which $\mathcal{F}$ evaluates to 1 vis-a-vis the logical expression of a query are retrieved. Note that the retrieval is binary in the sense that there is no degree of the relevance of a document to the query. For example, if a query has three words: q_1, q_2, q_3, a boolean statement $\mathcal{B}$ can be constructed as $\mathcal{B} = \neg q_1 \wedge (q_2 \vee q_3)$. Using this statement, we would retrieve all the files which do not include q_1 and have either q_2 or q_3 in them. Notice that there are also no weights for the query words. The retrieval is simply based on the presence of the words in the query. Because of the simple structure of SBM model, the implementation is very easy and the retrieval is very fast even for large datasets. However, this mechanism can retrieve too few or too many documents since it is solely based on exact matching. Moreover, there is no sense of ranking during retrieval, all the retrieved documents are equally relevant to the query.

3.2 Vector Space Model

Vector Space Model (VSM) represents a document with a vector in which each word occupies a fixed index with its frequency in the document. If we denote the vocabulary of the corpus by V then clearly the size of this vector, $\mathbf{w}$, will be equal to the size of the vocabulary $|V| = N$. Then the dth document in the corpus can be written as $\mathbf{w}_d = \begin{bmatrix} w_{1,d} & w_{2,d} & \cdots & w_{|V|,d} \end{bmatrix}^T$ where $w_{i,d}$ is the frequency of the ith word in the dth document [52]. For a better retrieval performance, instead of raw word frequencies, TFIDF[1] is commonly used in this vector representation [14]. TFIDF of a word is obtained with the following equation:

[1]Term Frequency - Inverse Document Frequency

$$tfidf(w) = w \times log_2 \frac{M}{E} \tag{3.2}$$

where M is the number of documents in the corpus and E is the number of documents in which the word appears at least once. The logarithmic expression in the right hand side of the equation is called IDF and it helps assign lower frequencies to the terms that appears in many documents in the corpus. This leads to a better document representation for retrievals as the terms that appear in many documents in a corpus are nondiscriminatory.

With this vector representation of the documents, a corpus can be represented by a matrix D where the columns are the document vectors and the rows are the term vectors. This matrix encapsulates all the semantic information of the corpus with the bag-of-word assumption.

VSM representation allows us to use the geometrical properties of the document vectors for comparisons. If we also represent a query using the same notation by $\mathbf{w}_q = \begin{bmatrix} w_{1,q} & w_{2,q} & ... & w_{|V|,q} \end{bmatrix}^T$, then the documents can be ranked based on their similarities to the query in terms of the angle between the corresponding vectors. Conventionally, the similarity between a query and a document is determined by the cosine of the angle.

$$cos(\mathbf{w}_q, \mathbf{w}_d) = \frac{\mathbf{w}_d \cdot \mathbf{w}_q}{||\mathbf{w}_d||\ ||\mathbf{w}_q||} \tag{3.3}$$

A cosine value of 1 means that the query and the document are exactly the same and a cosine value of 0 means that there is no similarity between the query and the document.

On the basis of this retrieval infrastructure, Latent Semantic Analysis (LSA) brings a new perspective to the VSM model by introducing the concept space or the topic space via Singular Value Decomposition (SVD) of the corpus matrix D [53].

$$D = U \Sigma V^T \tag{3.4}$$

where the orthonormal matrix U has the eigenvectors of DD^T and the orthonormal matrix V has the eigenvectors of D^TD in their columns. The diagonal matrix Σ holds the square roots of the eigenvalues of DD^T and D^TD called singular values for the corresponding eigenvectors in U and V. It is important to observe that only the ith row of the matrix U contributes to the ith term and only the jth column of V^T contributes to the jth document. Therefore, the matrix V^T has the transformed document vectors in its columns and the matrix U has the transformed term vectors in its rows. It turns out that when the largest k singular values and the corresponding eigenvectors from the matrices U and V are taken, rank k approximation of the matrix D, D_k, is obtained with minimum error in the least square sense. This dimension reduction not only allows us to exclude the unimportant details of the corpus but also to create a k-dimensional topic space in which the correlations between documents and terms are stressed.

$$D_k = U_k \Sigma_k V_k^T \tag{3.5}$$

In order for document retrieval to take place, clearly a query also needs to go through the same transformation as the documents.

$$\hat{\mathbf{w}}_q = \Sigma_k^{-1} U_k^T \mathbf{w}_q \tag{3.6}$$

Now using again the cosine similarity, a query can be compared to the documents in the topic space for retrieval.

These deterministic models are proven to be useful for document retrieval and data summarization. Unfortunately, VSM and LSA have some disadvantages when they are used in bug localization. Although the document comparison with VSM is very intuitive, it suffers from the inappropriate document length normalization of the source files as it tends to penalize long documents [39]. As for LSA, it is difficult to interpret the resulting document and term matrices of SVD since they can include negative values. For large datasets, SVD may also be undesirable as it is computationally expensive.

3.3 Divergence From Randomness

An important class of retrieval models are grouped under the Divergence From Randomness (DFR) framework [54]. DFR is an information theoretic approach that evaluates the appropriateness of a document to a query on the basis of the divergence of document feature probabilities from pure non-discriminative random distributions. The goal is to model the noise in the data with simple probability distributions in order to separate the discriminatory document features from the background noise. For this purpose, two probability distributions are used to capture the information content of the documents with respect to the query terms:

$$score_{DFR}(w_i, \mathbf{w}_d) = [-log_2 Prob_1(w_i, \mathbf{w}_d)] \cdot [1 - Prob_2(w_i, \mathbf{w}_d)]. \tag{3.7}$$

In this formulation, $[-log_2 Prob_1(w_i, \mathbf{w}_d)]$ measures the divergence from randomness. $Prob_1$ is the probability of having tf occurrences of the term w_i in the document $\mathbf{w}_d$ by pure chance and the lower the value of $Prob_1$, the higher the score of the term for that document. $Prob_2$, on the other hand, works as a normalizer. Using different distributions for these probabilities results in different retrieval models. We will explain this framework in detail in Chapter 4 as we will use it for bug localization.

3.4 Unigram Model

The unigram model is the simplest probabilistic model that makes the bag-of-word assumption, as the other popular models, when sampling the words from documents. With this model, we compute the probabilities of individual terms in a document using the maximum likelihood estimate:

$$P_{ML}(w_i|\mathbf{w}_d) \quad = \quad \frac{tf(w_i, \mathbf{w}_d)}{\sum_{i=1}^{N} tf(w_i, \mathbf{w}_d)} \tag{3.8}$$

where $tf(w_i)$ is the term frequency of the word. In order for document retrieval to take place, given a query $\mathbf{w}_q$, we can estimate the query likelihood $p(\mathbf{w}_q|\mathbf{w}_d)$ [13]. The documents that yield the largest values for such likelihoods could then be returned

in response to the query. If we assume that the query $\mathbf{w}_q$ consists of the words $\{w_1, w_2, ..., w_n\}$, then under the bag-of-words assumption for both the query and the corpus model, we could decompose the query likelihood as follows:

$$P(\mathbf{w}_q|\mathbf{w}_d) \quad = \quad \prod_{w_i \in \mathbf{w}_q} P_{ML}(w_i|\mathbf{w}_d) \tag{3.9}$$

Query likelihood given in this form is a widely accepted document retrieval technique with language models. Although the underlying language model $P(w_i|\mathbf{w}_d)$ can be made much more sophisticated, query likelihood is computed by the same product for other language models as well.

Because of the bag-of-words assumption, an interesting dilemma occurs when query likelihoods are used for document retrieval. If a query word is missing in a document, the likelihood formula above would become zero for that document. For such cases, even if a document was highly relevant to a query on account of containing the rest of the query words, the absent words would make it impossible to retrieve those documents.

To address this problem created by the query words not being present in a document, researchers have proposed that the estimated model be "smoothed" to account for the missing words. The goal of smoothing is to discount the probabilities assigned to the words that actually occur in a document and to then apportion the left-over probability mass among the vocabulary words not present in the document [55].

In Chapter 4, we will describe the unigram model with smoothing in detail and show how it can be used in bug localization.

3.5 Probabilistic Topic Models

Recently, probabilistic topic models have become very popular as they provide a latent topic space in which the dependencies of the documents and the words can be revealed efficiently. Especially the Latent Dirichlet Allocation (LDA) model which allows the documents to have multiple topics with continuous weights gained a con-

siderable attention from the research community [56–58]. Before going into the details of how topic models can be used for document retrieval, we will briefly explain the three commonly used topic models in the literature.

3.5.1 Mixture of Unigrams (MU)

The Mixture of Unigrams is one of the most powerful models for data classification and in particular for text classification. In this model, the probability distribution of a document is expressed in terms of the mixture components $c_j \in C = c_1, ..., c_K$ [2]. In this framework, a document is generated by first selecting a topic and then using the parameters of that topic. The likelihood of a document $\mathbf{w}_i$ is given by:

$$P(\mathbf{w}_i|\theta) = \sum_{j=1}^{K} P(z_j|\theta)P(\mathbf{w}_i|z_j;\theta) \tag{3.10}$$

where K is the number of latent topics in the corpus.

Since the documents in the collection are independent of each other, the likelihood of the whole dataset is given by:

$$P(D|\theta) = \prod_{i=1}^{M} P(y_i = z_j|\theta)P(\mathbf{w}_i|y_i = z_j;\theta) \tag{3.11}$$

where we indicate the topic labels for each document by a binary variable y_i. Smoothing plays an important role when we compute the parameters of the model. Because of the bag-of-words assumption, a dirichlet prior θ with parameter α is used to smooth the word and the topic distributions to prevent zero probabilities [59].

$$P(\theta) \propto \prod_{z_j \in C} ((\theta_{z_j})^{\alpha-1} \prod_{w_t \in V} (\theta_{w_t|z_j})^{\alpha-1}). \tag{3.12}$$

Note that when $\alpha = 2$, the smoothing that the above prior causes becomes Laplace smoothing.

[2]The model assumes that there is a one-to-one correspondence between the topics z_j and the classes c_j for $j = 1, \cdots, K$.

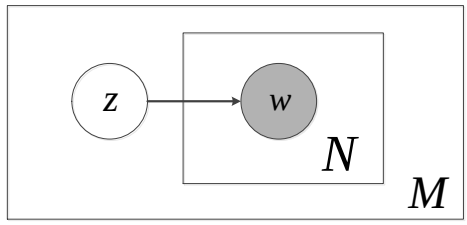


Fig. 3.1. The Mixture of Unigrams Model in its graphical representation. M represents the number of documents in the corpus and N represents the vocabulary size.

Inference includes learning the parameters of the model given the data: $D = \{\mathbf{w}_1, ..., \mathbf{w}_M\}$. The maximum a posteriori (MAP) is used to obtain the estimate $\hat{\theta}$ of the parameters θ with Laplace Smoothing.

$$P(\theta|D) \propto P(D|\theta)P(\theta) \tag{3.13}$$

There are two sets of parameters of the model which are computed by the following familiar ratios [59]. The probability of a word given a topic can be estimated by:

$$\hat{\theta}_{w_t|z_j} \equiv P(w_t|z_j;\hat{\theta}) \;=\; \frac{1+\sum_{i=1}^{M} N(w_t,\mathbf{w}_i)P(y_i=z_j|\mathbf{w}_i)}{N+\sum_{s=1}^{N}\sum_{i=1}^{M} N(w_s,\mathbf{w}_i)P(y_i=z_j|\mathbf{w}_i)} \tag{3.14}$$

where $N(w_t, \mathbf{w}_i)$ is the count of the word w_t in document $\mathbf{w}_i$ and $P(y_i = z_j|\mathbf{w}_i)$ is 1 if the label of document $\mathbf{w}_i$ is z_j, 0 otherwise.

Similarly, the probability of a topic is estimated by

$$\hat{\theta}_{z_j} \equiv P(z_j|\hat{\theta}) \;=\; \frac{1+\sum_{i=1}^{M} P(y_i=z_j|\mathbf{w}_i)}{K+M} \tag{3.15}$$

Using this classifier, we can cluster the documents or classify an unseen document into one of the $|C|$ classes. We can also rank the documents in the collection based on the likelihood of a new document (as a query).

In order to predict the label of an unlabeled document, the Bayes Rule is used to rank the topics:

$$P(y_i = z_j|\mathbf{w}_i;\hat{\theta}) = \frac{P(z_j|\hat{\theta})\prod_{k=1}^{|\mathbf{w}_i|} P(w_{k,i}|z_j;\hat{\theta})}{\sum_{r=1}^{K} P(z_r|\hat{\theta})\prod_{k=1}^{|\mathbf{w}_i|} P(w_{k,i}|z_r;\hat{\theta})}. \tag{3.16}$$

Note that, this classification assigns each document into only one topic, however in a real life scenario a document might discuss more than one topic. Despite this assumption of one-to-one correspondence of the topics and the classes, the Mixture of Unigrams model does very well in practice [59].

3.5.2 Probabilistic Latent Semantic Analysis (pLSA)

Probabilistic Latent Semantic Analysis relaxes the one topic per document assumption that the mixture of unigrams model makes [60]. As can be seen in Fig. 3.2, it associates a latent topic with each word, instead of sampling a single topic for each document. pLSA can also be viewed as the probabilistic counterpart of Latent Semantic Analysis. Since pLSA is a discrete probabilistic model, it does not suffer from the negative weights for words or documents as LSA does. It also provides a better fit to the data. With pLSA, the probability of generating a document is given by:

$$p(\mathbf{w}_d) = \prod_{i=1}^{n_d} \sum_{j=1}^{K} p(w_{i,d}|z_j)p(z_j|\mathbf{w}_d) \tag{3.17}$$

Model fitting is done using an Expectation Maximization (EM) procedure. In E-step $p(z_j|\mathbf{w}_i, w_t)$ is computed by:

$$p(z_j|\mathbf{w}_i, w_t) \;=\; \frac{p(z_j)p(\mathbf{w}_i|z_j)p(w_t|z_j)}{\sum_{k=1}^{K} p(z_k)p(\mathbf{w}_i|z_k)p(w_t|z_k)}. \tag{3.18}$$

In M-step:

$$p(w_t|z_j) \propto \sum_{i=1}^{M} N(w_t, \mathbf{w}_i)p(z_j|\mathbf{w}_i, w_t) \tag{3.19}$$

$$p(\mathbf{w}_i|z_j) \propto \sum_{t=1}^{N} N(w_t, \mathbf{w}_i)p(z_j|\mathbf{w}_i, w_t) \tag{3.20}$$

$$p(z_j) \propto \sum_{i=1}^{M} \sum_{t=1}^{N} N(w_t, \mathbf{w}_i)p(z_j|\mathbf{w}_i, w_t). \tag{3.21}$$

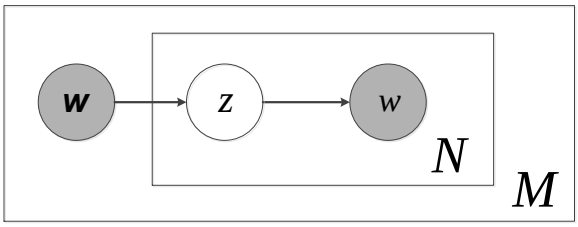


Fig. 3.2. pLSA Model in its graphical representation.

Although promising results are obtained with pLSA model, the model does not have well defined semantics and there is no natural way of computing the probability of an unseen document [61]. Moreover, the number of parameters to calculate grows linearly with the number of documents, implying overfitting to the training data.

3.5.3 Latent Dirichlet Allocation

Latent Dirichlet Allocation (LDA), introduced by Blei et al. [61], is a powerful approach for the probabilistic modeling of text corpora. It provides us with a generative framework in which topics serve as hidden variables that mediate between the documents and the words as pLSA model does. Roughly speaking, the main difference from pLSA is the use of dirichlet priors for the topic and the word distributions. This makes it possible for LDA to create a continuous topic space leading to interpretable semantics for the corpus. pLSA lacks this smoothing effect — a very important component for text modeling [62].

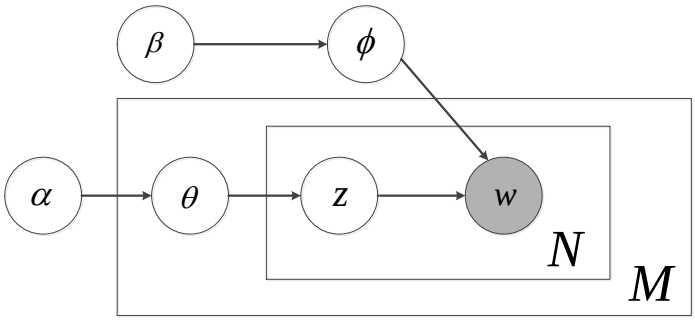


Fig. 3.3. The LDA model in its graphical representation

Table 3.2 The parameters of the LDA model

α	a K-dimensional hyper-parameter vector of the Dirichlet distribution that is used to derive the topic proportions in each document
β	a $\|V\|$-dimensional hyper-parameter vector of the Dirichlet distribution that is used to derive the word proportions in each topic
Θ	an $M \times K$ matrix of latent variables whose each row expresses the topic proportions in each document (derived from $Dir(\alpha)$))
Φ	a $K \times \|V\|$ matrix of latent variables whose each row expresses the word proportions on a corpus-wide basis in each topic (derived from $Dir(\beta)$)

The goal of LDA modeling is to estimate the parameters of Dirichlet-based probabilities for the topic proportions in the documents and the word proportions in the topics. Figure 3.3 gives the graphical representation of the LDA model and Table 3.2 gives a description of the parameters involved.

Regarding the parameters shown in Table 3.2, the corpus D of vocabulary V is modeled as a multinomial distribution over K topics, and each topic is modeled as a multinomial distribution over $|V| = N$ words. All multinomial parameters are based on Dirichlet priors because of the conjugate property of such functions. The Dirichlet prior for how the topics are distributed in a corpus is parametrized by α and the Dirichlet prior for how the words are distributed in the topics for the entire corpus is parameterized by β.

LDA derives its representational power from the fact that the various topics may "reside" in a document in any proportion over a continuous range, subject to the usual normalization constraints. Various approaches have been suggested for estimating the parameters of the LDA model for a given corpus [61, 63]. A previously constructed LDA model can be used to characterize an *unseen* document by the distribution of the K topics in the document, this distribution being similar to a row of the $M \times K$ matrix $\mathbf{\Theta}$ for all the *seen* documents in the corpus.

3.6 Document Retrieval with Probabilistic Topic Models

With topic-based document models such as MU, pLSA and LDA, document retrieval can be performed in the hidden layer of topics using a divergence based similarity metric. With LDA, given a query, one can treat the query as an unseen document onto itself and then calculate a maximum a posteriori estimate for its topic proportions using either the Blei's variational inference procedure [61] or a Gibbs sampling based approach [63, 64]. What is interesting is that MAP estimate of the topic proportions for a query (that is treated as a document) and such previously-calculated estimates for the topic proportions for the corpus documents *can all be treated as valid 1-D distributions in the form of histograms discretized to K bins.* Recall that we have K topics and that the topic proportions, being parameters of the multinomial distributions, must sum up to 1. The 1-D topic-proportion distribution representing a query can therefore be compared with similar 1-D distributions for the corpus documents with the help of, say, the Jensen-Shannon divergence. As demonstrated in [64], such a topic-based approach can also be used for document clustering.

However, comparing the documents to a query solely on the basis of their topic distributions is not enough in general. Several researchers proposed cluster based document retrieval for superior results. One such approach uses the document clustering as a filtering process in a large corpus by first selecting the most likely cluster for a given query and then retrieving the documents from that cluster only [65, 66].

Combining unigram model with topic models and/or cluster models to increase the performance is another approach [67]. Wei and Croft [68] demonstrated how LDA model can be combined with unigram and collection model to produce superior retrieval performance. As mentioned earlier, the query likelihood formula is also the skeleton of document retrieval with topic models.

$$p(\mathbf{w}_q|\mathbf{w}_d) \quad = \quad \prod_{w_i \in \mathbf{w}_q} p(w_i|\mathbf{w}_d) \tag{3.22}$$

In this formula, instead of using only unigram model, $p(w_i|\mathbf{w}_d)$ includes a weighted mixture of the following three likelihoods: (1) the unigram model (this is usually referred to as representing the *document model*); (2) the maximum-likelihood probability $P_{ML}(w_i|D)$) of a query word w_i conditioned directly on the entire corpus D (this is usually referred to as representing the *collection model*); and, finally, (3) the likelihood probability that the LDA model associates with a query word w_i conditioned on the document $\mathbf{w}_d$. Here is Wei and Lafferty's formula:

$$\begin{aligned} p(w_i|\mathbf{w}_d) &= \lambda(\frac{n_d}{n_d+\mu}P_{ML}(w_i|\mathbf{w}_d)+ \\ &\quad (1-\frac{n_d}{n_d+\mu})P_{ML}(w_i|D)) + (1-\lambda)P_{LDA}(w_i|\mathbf{w}_d) \end{aligned} \tag{3.23}$$

Where λ and μ are the weighting parameters that need to be tuned for a corpus and n_d the number of distinct words in the document $\mathbf{w}_d$. With regard to the LDA portion of the above formula, we can make explicit the dependence of that likelihood on the hidden topics of the LDA model by expressing it as:

$$P_{LDA}(w_i|\mathbf{w}_d) = \sum_{k=1}^{K} p(w_i|z_k,\Phi)p(z_k|\mathbf{w}_d,\Theta) \tag{3.24}$$

3.6.1 Discussion

Although the topic models mentioned in this chapter have been successfully applied in many applications, they do not perform as well for Bug Localization (BL) [37, 39]. Moreover, the exact inference for complex topic models is not tractable. Although the retrievals may benefit from these approaches when they are combined with simpler models, the computational overhead may be too severe to justify the minor improvements in large projects [37, 67, 68].

4. INFORMATION RETRIEVAL FOR BUG LOCALIZATION

Based on the advantages and the disadvantages of the retrieval models mentioned in Chapter 3, we adapt two IR frameworks for Bug Localization (BL): (1) Language Modeling (LM) and (2) Divergence From Randomness (DFR) [54, 69]. LM is a probabilistic approach whereas DFR is an information theoretic framework. These approaches to document retrieval do not require any training and are shown to perform well in retrieval tasks [16, 37].

4.1 Language Modeling

The language modeling approach uses the notion of query likelihood which ranks the documents in a collection according to the likelihood of relevance to the query [69]. Given a bug Q formulated as a textual query, our goal in BL is to compute the probability of a file f to be relevant to the query: $P(Q|f)$. With the *bag of words* assumption, the terms in the query, as well as in the documents, are regarded as occurring independently of one another, therefore we can write

$$P(Q|f) = \prod_{w \in Q} P(w|f) \tag{4.1}$$

where $P(w|f)$ is the likelihood of a bug term w in f and it is computed with ML estimation. Given a term w from the vocabulary V of the collection C, the ML estimate we seek is given by $P_{ML}(w|f) = tf(w, f)/\sum_{w' \in f} tf(w', f)$ where $tf(w, f)$ is the term frequency of w in $f \in C$. Scoring the set of files in this way is problematic as all the query terms may not be present in a given document, leading to zero probabilities. To overcome this problem, several *smoothing* techniques have been

proposed over the years [13]. Using the collection model for smoothing, Hiemstra Language Model (HLM) [13] computes the file likelihood of a term as follows:

$$P_{HLM}(w|f) = \lambda \cdot P_{ML}(w|f) + (1 - \lambda) \cdot P_{ML}(w|C) \tag{4.2}$$

where $P_{ML}(w|C)$ is the collection likelihood of the term and it is given by $P(w|C) = tf(w, C)/\sum_{w' \in V} tf(w', C)$ where $tf(w, C)$ represents the term frequency of w in the collection. The parameter λ is called the mixture variable and governs the amount of smoothing.

Another powerful smoothing approach is the Bayesian Smoothing with Dirichlet Priors [13]. If the Dirichlet parameters are chosen as $\mu P(w|C)$ for each term $w \in V$ then the file likelihood of a term is given by

$$P_{DLM}(w|f) = \frac{tf(w, f) + \mu P(w|C)}{\sum_{w' \in f} tf(w', f) + \mu} \tag{4.3}$$

where μ is the smoothing parameter. We denote this model by DLM (Dirichlet Language Model).

After computing the query likelihoods for all the files in a software collection, we rank the files in decreasing order of these probabilities and show a certain number of the top files to the developer as the most likely locations of the bug.

4.2 Divergence from Randomness

That brings us to the second major approach to document retrieval: the Divergence from Randomness (DFR) based approach. As mentioned earlier, DFR is an information theoretic approach. It evaluates the appropriateness of a document to a query on the basis of the divergence of document feature probabilities from pure non-discriminative random distributions. The core idea in DFR is that the terms that do not help discriminate the documents in a collection are distributed randomly while the discriminative terms tend to appear densely only in a small set of *elite*[1]

[1]A document is regarded as an elite document for a term if the term appears at least once in the document.

documents. These content bearing terms or the *specialty terms* should not follow a random distribution and the amount of divergence from randomness determines the discriminatory power of the term for retrieval. The higher the divergence, the higher the importance of the term in the retrieval. The DFR framework allows us to avoid parameter tuning to a great extent since the models are *non-parametric*.

In this framework, the score of a document with respect to a single query term is given by the product of two information content:

$$s_{DFR}(w, f) = [1 - Prob_2(w, f)] \cdot [-log_2 Prob_1(w, f)]. \tag{4.4}$$

$Prob_1$ is the probability of having tf occurrences of the term in the document by pure chance and as this probability decreases, the information content $-log_2 Prob_1$ of the document vis-a-vis the term increases. $(1 - Prob_2)$, on the other hand, is related to the risk of choosing the query term as a discriminative term and works as a normalization factor. Amati and Rijsbergen [54] uses the probability of having one more occurrence of the term in the document as $Prob_2$, which leads to penalizing the high frequency terms during retrieval. Using different probability distributions in these two information contents results in different retrieval models.

Similar to the LM, the models we present from DFR also use the bag of words assumption. Therefore the score of a file with respect to a query is given by

$$s_{DFR}(f|Q) = \sum_{w \in Q} s_{DFR}(w, f). \tag{4.5}$$

4.2.1 Tf-Idf Models for $Prob_1$

Assuming that the terms are being distributed in the documents randomly, having tf occurrences of a term in a document by pure chance is given by $Prob_1 = p^{tf}$ where p is the probability of a term to appear in any document. In order to compute the posterior distribution for p, usually a beta prior with parameters $\alpha_1 = -0.5$ and $\alpha_2 = -0.5$ is assumed. The evidence in computing p is given by the probability of the term to land in E elite documents out of M documents in a collection, which can be

modeled by a binomial distribution $P(E|p, M) = \binom{M}{E} p^E \cdot (1-p)^{M-E}$. In this case, the posterior will also be in the form of the beta distribution and the expected value of p is given by $(E + 0.5)/(M + 1)$. Therefore,

$$-log_2 Prob_1 = tf \cdot log_2 \frac{M+1}{E+0.5}. \tag{4.6}$$

This information content is denoted by "In" (Inverse document frequency).

Using the expected number of elite documents E_e instead of E in the formula above results in a separate retrieval model. For this purpose, the *expected number of elite documents* for a given term can be computed by $E_e = M \cdot P(tf \neq 0)$. In order to compute the probability $P(tf \neq 0)$, we again assume that the terms are being distributed to the documents randomly. If the probability of a term appearing in a document out of M documents is given by $1/M$, then $P(tf \neq 0) = 1 - (\frac{M-1}{M})^{TF}$ where TF is the total number of occurrences of the term in the collection. We denote this retrieval model by "InExp".

4.2.2 Normalizing Information Content ($Prob_2$)

$Prob_1$ by itself is not sufficient to accurately discriminate specialty terms since the terms with high frequencies will always produce small $Prob_1$ and thus become dominating during retrieval. To normalize the information content, two main methods have been proposed: Normalization L and Normalization B [54]. Normalization L[2] achieves this effect by estimating the probability of having one more token of the same term in the document by

$$Prob_2 = \frac{tf}{tf+1}. \tag{4.7}$$

Normalization B, on the other hand, assumes that a new token of the same term already having TF tokens is added to the collection. With this new token, the

[2]L stands for Laplace as this probability is given by so-called Laplace's law of succession

probability of having $tf + 1$ occurrences of the term in a document can be estimated by the following binomial probability:

$$Binom(E, TF + 1, tf + 1) = \binom{TF + 1}{tf + 1} p^{tf} \cdot q^{TF-tf} \tag{4.8}$$

where $p = 1/E$ and $q = 1 - p$. Then the incremental rate between $Binom(E, TF, tf)$ and $Binom(E, TF + 1, tf + 1)$ gives the normalization factor:

$$Prob_2 = 1 - \frac{Binom(E, TF + 1, tf + 1)}{Binom(E, TF, tf)}. \tag{4.9}$$

4.2.3 Document Length Normalization

Document length is another important factor in retrieval. It has been shown that the relevancy of a document to a query is dependent on the document length [13]. Normalization 2 as proposed in [54] uses the assumption that the term frequency density in a document is a decreasing function of the document length. If the effective document length is chosen as the average document length in the collection, then the normalized term frequency can be estimated by

$$tfn = tf \cdot log_2(1 + \frac{avg\_l}{l}) \tag{4.10}$$

where l is the length of the document and avg_l is the average document length in the collection. Note that with this normalization, instead of the regular tf, tfn is used in the computation of $Prob_1$ and $Prob_2$.

4.3 TF-IDF Retrieval Models

Another set of widely used retrieval algorithms are grouped under the TF-IDF scheme [14]. In this retrieval framework, the score of a document with respect to a single query term is given by the multiplication of the term frequency with the

Table 4.1 Retrieval Models used in BL with bag-of-words assumption.

Language Models (Section 4.1)	
HLM	Hiemstra Language Model
DLM	Dirichlet Language Model
Divergence from Randomness (Section 4.2)	
InB2	Inverse Document Frequency + Normalization B + Normalization 2
InExpB2	Inverse Document Frequency with expected number of elite documents + Normalization B + Normalization 2
InL2	Inverse Document Frequency + Normalization L + Normalization 2
Tf-Idf	Robertson's tf + Sparck Jones' Idf

inverse document frequency. One algorithm that produced strong empirical results in our experiments uses Robertson's tf which is given by

$$a_1 \cdot \frac{tf}{(tf + a_1 \cdot (1 - a_2 + a_2 \cdot l/avg\_l))} \tag{4.11}$$

and Spark Jones' Idf which is given by $log_2(\frac{M}{E+1})$. The parameters a_1 and a_2 in Robertson's tf provide non-linearity to the term frequencies for scoring the documents.

Table 4.1 summarizes the models explained in this section.

5. INCORPORATING VERSION HISTORIES IN IR-BASED BUG LOCALIZATION

Software defects can emanate from a large variety of sources. In addition to the type of the bugs e.g. functional errors, technical errors etc., the severity of the bugs e.g. critical, high, low, enhancement and so on, also plays an important role in Bug Localization (BL). This intricate nature of the bugs often leads to poor retrieval accuracies when the underlying retrieval algorithm depends only on the textual content of the software. In this chapter, we present a Bayesian retrieval framework that incorporates query independent prior knowledge to improve the accuracy of BL. For this purpose, we show how version histories of a software project can be used to estimate a prior probability distribution for defect proneness associated with the files in a given version of the project. Subsequently, we use these priors in our Bayesian framework to determine the posterior probability of a file being the cause of a bug.

We first introduce two models to estimate the priors, one from the defect histories and the other from the modification histories, with both types of histories as stored in the versioning tools. Referring to these as the base models, we then extend them by incorporating a temporal decay into the estimation of the priors, placing greater weight on recent maintenance efforts. We demonstrate that IR based BL accuracy can be significantly improved when such models for the priors are employed in retrieval.

With regard to leveraging prior development efforts, many researchers investigated the predictive power of the version histories of the software artifacts to assist developers in software development tasks [70]. In an IR based retrieval framework that leverages the prior evolutionary information concerning the development of the software, Kagdi et al. [71] have demonstrated how change impact analysis can be carried out by exploiting the conceptual and evolutionary couplings that exist between the different software entities. Another relevant work is reported by Nguyen

et al. [38]. They have proposed BugScout, an automated approach based on Latent Dirichlet Allocation to narrow down the search space while taking into account the defect proneness of the source files.

In support of the predictive power of version and modification histories stored in software repositories, studies such as those reported in [72–77] have demonstrated that the version histories store a wealth of information that could potentially be used to predict the future defect likelihoods of the software entities such as files, classes, methods, and so on. Along the same lines, studies such as those reported in [78, 79] have shown that prior modification history of software components can be used to guide engineers in future development tasks. Motivated by these and similar other studies, our goal here is to mine the defect and file modification related knowledge that is always buried in the software repositories and to incorporate this knowledge in well-principled retrieval models for fast and accurate BL in large software projects.

In regard to using version histories for BL, the work by Hassan and his collaborators has demonstrated that defects are mainly associated with high software modification complexity [76]. Many modifications committed by several programmers during a short period of time is a strong predictor for future defects [77]. Besides complex prior modification history, the defect histories of the files in a software project is also a good predictor of future defects. A buggy file in the early stages of the project development is likely to produce defects throughout the life cycle of a project unless the project undergoes a fundamental design change [80, 81].

5.1 Estimating Defect & Modification Based Prior Probabilities

After a software product has entered the market place, any further evolution of the software typically takes place in small steps in response to change requests such as those for adding a new feature, modifying an existing feature, bug fixing, and so on. At each step, new files may be added to the code base of the project or existing files may be removed or altered to implement the requested change. Software

Configuration Management (SCM) tools such as SVN create a new revision of the project by incrementally storing these *change-sets* with every commit operation. This incremental nature of software development plays an important role in many software engineering tasks, particularly in BL, because the modifications made to a specific set of files in response to a change request suggest strong *empirical* dependencies among the changed files that may not be captured otherwise via dynamic or static properties of the software such as call graphs, APIs or execution traces created by running a set of test cases etc.

Naturally, not all change-sets imply strong interdependencies among the involved files. For example, the change-sets for what are usually referred to as General Maintenance (GM) tasks tend to be very large. As a case in point, a change-set for removing unnecessary import statements from all Java files in a code base [82] does not carry much useful information with regard to any co-modification dependencies between the files. We do not use such change-sets in our models. We regard all non-GM change-sets accumulated during the life cycle of a software project as Feature Implementation (FI) change-sets. We consider an FI change-set to be of type BF (Bug Fixing) if it is specifically committed to implement a bug fix. The common approach to determining whether a change-set is BF is to lexically analyze the commit message associated with it [83]. Those commit messages include key phrases such as "fix for bug" or "fixed bug" etc.

Over the years, researchers have proposed several process and product related metrics to predict the defect potential of software components in order to help the software managers make smarter decisions as to where to focus their resources. Several studies have shown that bug prediction approaches based on process metrics outperform the approaches that use product metrics [74–76]. Along the same lines, in our work we use FI and BF change-sets to compute the modification and the defect probabilities respectively.

We denote the kth change-set of a software project by r_k; this represents the set of the modified files in response to the kth change request for $k = 1 \cdots K$ during

Table 5.1 The Notation Used in modeling version histories

K	Total number of change-sets
C_k	The collection of the software files after the kth commit
R_k	The set of change-sets up to and including the kth change-set
β_1	The decay parameter for modification probabilities
β_2	The decay parameter for defect probabilities

software development or maintenance. After the kth commit, with some files having being altered, a new collection of files C_k is created. If C_k exhibits defective behavior, the defect and the modification probabilities of the files in C_k can be modeled by a *multinomial distribution*[1] where $P(f|C_k)$ represents the probability of a file f to be responsible for the defective behavior reported. Obviously, $\sum_{f \in C_k} P(f|C_k) = 1$. We assume that this probability associated with a file is independent of the rest of the files in the collection. We refer to this as the "bag of files" assumption.

In the rest of this section, we first propose two base models that determine from the version histories the prior defect and modification probabilities associated with the files in a software project. We have named them *Modification History based Prior* (**MHbP**) and *Defect History based Prior* (**DHbP**). Then we extend these models by incorporating in them a time decay factor. We add the suffix 'd' to the acronyms of the base models to indicate the decay-based versions of the two models by **MHbPd** and **DHbPd** respectively. Table 5.1 summarizes the notation used in these models.

5.1.1 The MHbP Model

Several authors have established that the modification history of a file is a good predictor of its fault potential. Hassan used the code change complexity to estimate the defect potential of the files on the basis of the rationale that as the number of

[1]The multinomial distribution and the categorical distribution are used exchangeably in the IR community. Here we adhere to the tradition.

modifications to the files increases, the defect potential of the files must also increase [76]. Nagappan et al. [77] showed that the *change bursts* during certain periods of software development are good indicators of future defects. The main intuition behind their approach is that implementing many changes in a short period of time complicates the development process, leading to defects.

With the MHbP model, we translate the frequencies with which the files are mentioned in the change records into file modification probabilities. Using Maximum Likelihood (ML) estimation, the modification probability of a file f in a given collection C_k can be expressed as

$$P_{MHbP}(f|C_k, R_k) = \frac{\sum_{i=1}^{k} I_m(f, r_i)}{\sum_{f' \in C_k} \sum_{i=1}^{k} I_m(f', r_i)} \tag{5.1}$$

where

$$I_m(f, r_i) = \begin{cases} 1, & f \in r_i \ \& \ r_i \in FI \\ 0, & otherwise. \end{cases} \tag{5.2}$$

In the formulation, R_k represents the set of change-sets from the beginning of the development to the kth change-set. Obviously $R_K = FI \cup GM$. The model assigns a larger probability mass to the files modified more frequently with

$$\sum_{f \in C_k} P_{MHbP}(f|C_k, R_k) = 1. \tag{5.3}$$

5.1.2 The DHbP Model

Bug fixing change-sets mark the defect producing files during the life cycle of a project. The files mentioned in these change-sets are highly likely to produce bugs in the future as the buggy files tend to remain buggy [80, 81].

Similar to the MHbP model, we estimate the defect probability of a file by

$$P_{DHbP}(f|C_k, R_k) = \frac{\sum_{i=1}^{k} I_b(f, r_i)}{\sum_{f' \in C_k} \sum_{i=1}^{k} I_b(f', r_i)} \tag{5.4}$$

where

$$I_b(f, r_i) = \begin{cases} 1, & f \in r_i \ \& \ r_i \in BF \\ 0, & otherwise. \end{cases} \tag{5.5}$$

$I_b(f, r_i)$ is an indicator variable that becomes one if r_i implements a bug fix that results in a modification in f. This probability gives the ML estimate for the defect probabilities of the files.

5.1.3 Modeling the Priors with Temporal Decay

MHbPd

After a change request is implemented, it takes time for the files to stabilize and become bug-free. Indeed, implementing certain change requests may even take more than one commit operation perhaps from several developers [84]. However after the files have been stabilized, we expect a decrease in the modification probabilities. Therefore, even if a file had been modified frequently during a certain period of time in the past, if it has not been modified recently, the modification probability should decrease [75]. We incorporate a time decay factor into the formulation of the modification probabilities to take that facet of software development into account as follows:

$$P_{MHbPd}(f|C_k, R_k) = \frac{\sum_{i=1}^{k} e^{\frac{1}{\beta_1}(t_i - t_k)} I_m(f, r_i)}{\sum_{f' \in C_k} \sum_{i=1}^{k} e^{\frac{1}{\beta_1}(t_i - t_k)} I_m(f', r_i)} \tag{5.6}$$

where t_i represents the time at which the ith change-set was committed. This type of decay models has been used commonly in the past [75, 76, 80]. The parameter β_1 governs the amount of decay and it is related to the *expected time* for the files to stabilize with the implementation of a change request. As β_1 decreases, the amount of decay increases and therefore the expected stabilization time decreases.

DHbPd

Similar to the MHbPd model, recency of the bugs is an important factor in estimating the defect probabilities. We incorporate a time decay factor into the defect probabilities to emphasize the recent bug fixes in the estimation of prior defect probabilities as follows:

$$P_{DHbPd}(f|C_k, R_k) = \frac{\sum_{i=1}^{k} e^{\frac{1}{\beta_2}(t_i - t_k)} I_b(f, r_i)}{\sum_{f' \in C_k} \sum_{i=1}^{k} e^{\frac{1}{\beta_2}(t_i - t_k)} I_b(f', r_i)}. \tag{5.7}$$

5.2 A Bayesian Framework for BL

Our main goal is to score the files in the code base of a software project in order to rank them according to their relevance to a given bug. The models we have presented in the previous section estimates the prior probability of a file to be defective. Now we want to incorporate that prior probability into the bug localization process for an increased accuracy. Although in different contexts, incorporating query independent prior knowledge into IR systems has been extensively studied in the literature [85,86].

5.2.1 Document Priors

In the context of using document priors in a probabilistic retrieval framework, we use the Language Modeling (LM) framework of Ponte and Croft [12] and the Divergence From Randomness (DFR) framework of Amati and Risjbergen [54]. As presented in Chapter 4, these are the two main approaches to text retrieval that have been shown to produce strong empirical results. While LM presents a probabilistic approach in the Bayesian framework, DFR is an information theoretic approach that evaluates the appropriateness of a document to a query on the basis of the divergence of document feature probabilities from pure non-discriminative random distributions.

In the Bayesian framework, given the description of a bug Q as a query, we compute the posterior probability of a file f to pertain to the defective behavior by

$$P(f|Q, R_k, C_k) = \frac{P(Q|f, C_k)P(f|R_k, C_k)}{P(Q|R_k, C_k)}. \tag{5.8}$$

Since we are only interested in ranking the files and the denominator in Eq. 5.8 does not depend on the files, it can be ignored. Taking the logarithm for computational convenience, we compute the final score of a file being relevant to a given bug with the prior belief by

$$s_{LM}(f|Q, R_k, C_k) = log_2[P(Q|f, C_k)] + log_2[P(f|R_k, C_k)]. \tag{5.9}$$

In the DFR framework, the final score of a file in response to a query is altered to take the prior belief into account as follows [87]:

$$s_{DFR}(f|Q, R_k, C_k) = s_{DFR}(f|Q, C_k) + log_2[P(f|R_k, C_k)]. \tag{5.10}$$

Note that, in the previous section, we presented the estimation of $P(f|R_k, C_k)$. The estimations of $P(Q|f, C_k)$ in LM and $s_{DFR}(f|Q, C_k)$ in the DFR framework, on the other hand, are presented in Chapter 4.

5.3 Experimental Evaluation

In order to evaluate the proposed algorithms, we need to have the complete repository of a software project with a set of documented bug descriptions B. Unfortunately, software repositories and bug tracking databases are usually maintained separately [84]. Therefore, in general, we may not know the actual change-sets in the repository that are committed for implementing the fixes for bugs. One approach that has been used to get around this limitation is to look for pointers in the commit messages to the bug tracking database. The iBugs dataset created by Dallmeier and Zimmerman [88] uses this approach and it is therefore an appropriate testbed for our experiments.

5.3.1 Data Preparation for Bug Localization with Version Histories

We have evaluated the BL performance of our approach on AspectJ, an aspect-oriented extension to the Java programming language. The iBugs dataset for AspectJ contains a set of reported bugs, the collection of the files before and after a fix was implemented for each bug, and the files modified to implement the fix. We have used the pre-fix versions of the project as the corpora for our retrieval experiments.

Besides the iBugs dataset, we obtained the complete CVS repository of the AspectJ project which is publicly available[2]. Unfortunately, CVS repositories do not record the change-sets. In line with the approach presented in [89], we reconstruct the change-sets by grouping the files that are committed by the same author with the same commit message using a time fuzziness of 300 seconds. Table 5.2 gives various properties of the AspectJ project.

Indexing the Source Code

Since the code base of a software project keeps changing during development, in general, there can be significant differences in the underlying code base for any two bugs. In order to keep track of the changes in the code base, we create a distinct index for each collection that a bug $Q \in B$ was reported on.

Modern software projects tend to include a substantial number of non-executable files such as configuration files, documentation files, help files, and so on. Therefore finding only the executable files may not constitute a complete answer to the bug localization problem. XML files, for example, are used heavily in project configuration and it may be necessary to make modifications to these files to fix certain bugs. For obvious reasons, we regard the files with the extensions ".java" and ".aj" as executable. The rest of the files in the code base that may also cause defective behavior are regarded as non-executable. Figure 5.1 depicts the evolving size of the AspectJ project during the bug reporting period. On average, the non-executable

[2]http://archive.eclipse.org/arch/tools-cvs.tgz

Table 5.2 AspectJ Project Properties

K	$\|FI\|$	$\|BF\|$	$\|B\|$	Analysis Period
6,271	5,165	1,214	291	2001-01-16 - 2008-10-06

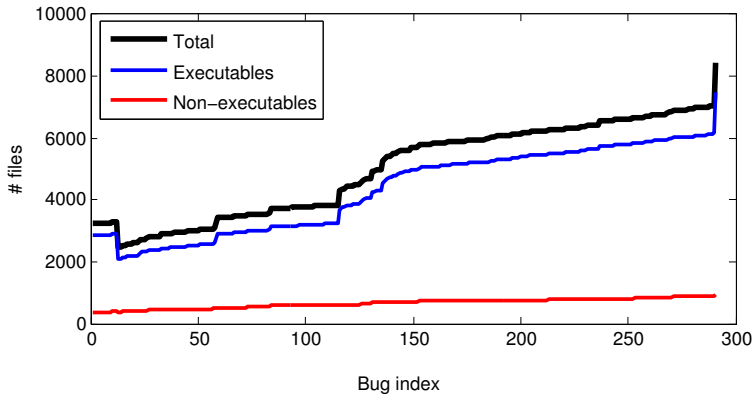


Fig. 5.1. The size of the AspectJ project as a function of the time of the bug report for each bug $Q \in B$.

files constitute 13.58% of the source code. It is important to note these different types of files have different characteristics. So, as described below, they are subject to slightly different tokenization procedures.

Treating each source-code file as a *bag-of-words*, we first use an initial stop list to remove the programming language specific tokens such as "public," "private," "transient" etc. from the executable files. Then we split the compound terms that use camel-case and punctuation characters. After these steps, we apply an English stop list to remove the noise terms such as "a," "an," "the" and so on, and apply Porter's Stemming algorithm to reduce the terms to their root forms [90]. All the terms obtained through this process are then lowercased and incorporated in the index. The non-executable files, also considered to be a bag of words, are not subject to the programming-language-specific stop list. However, we split the compound tokens in such files also.

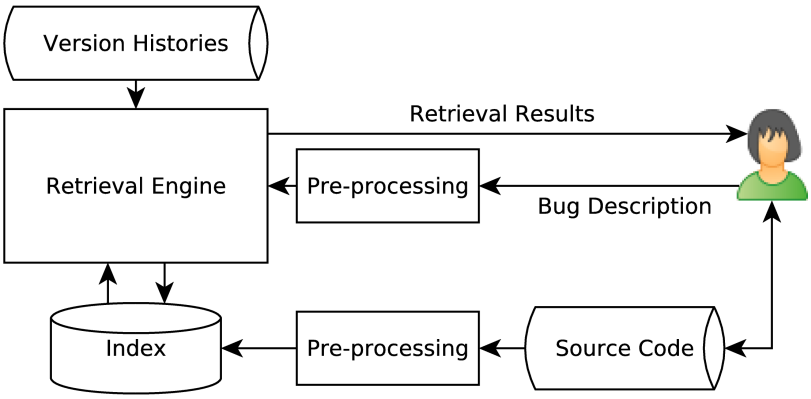


Fig. 5.2. An illustration of bug localization process with Terrier+.

Pre-processing Bug Reports

Typically, a bug report is written in ordinary English and it may include tokens from the code base. For example, a bug report may include the trace of an exception caused to be thrown by the bug, or it may include method names that are possibly related to the defective behavior.

We apply the same pre-processing steps to the bug reports as we did for the non-executables in the code base. That is, we carry out compound-term splitting and use stopping and stemming rules to prune the reports. We also drop those bug reports that have no associated files in the code base. After this pre-processing, we ended up with 291 bug reports and 1124 files associated with them, implying 3.86 files per bug on average. Figure 5.2 illustrates the BL process.

5.3.2 Retrieval Results

The accuracy of search engines is commonly measured by precision and recall metrics [13]. Let's say we have a query for which there exist four relevant files in the collection that we want the search engine to retrieve. If three of the top $r = 10$ retrieved files in the ranked list are relevant then the precision would be calculated as 3/10 and the recall as 3/4. A high precision indicates that a large fraction of the

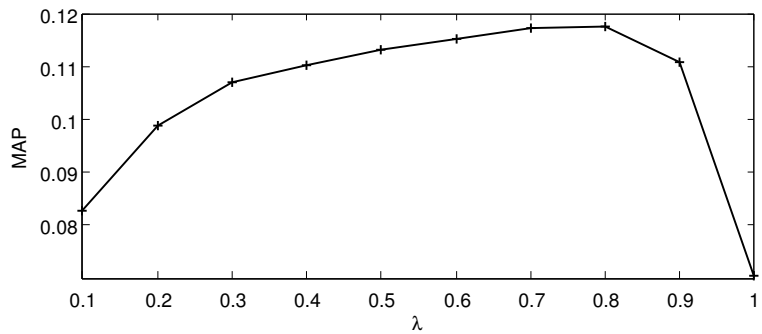


Fig. 5.3. The effect of varying λ in HLM.

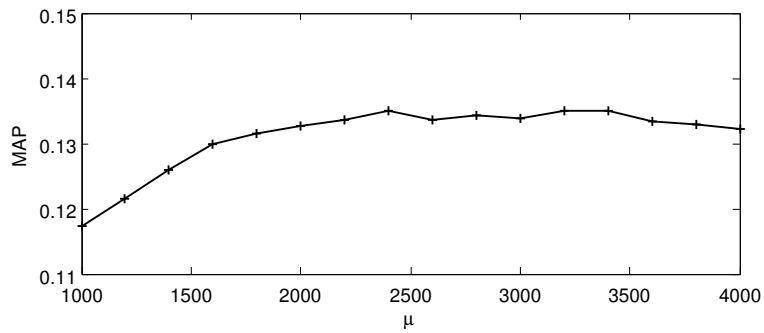


Fig. 5.4. The effect of varying μ in DLM.

retrieved set is relevant. Recall, on the other hand, measures the completeness of the results.

We have tabulated the retrieval performance using precision at rank r ($P@r$), recall at rank r ($R@r$) and Mean Average Precision (MAP) metrics. The average precision (AP) for a query $Q \in B$ is given by

$$AP(Q) = \frac{\sum_{r=1}^{RT} P@r \times I(r)}{rel_Q}, \tag{5.11}$$

where $I(r)$ is a binary function whose value is 1 when the file at rank r is a relevant file, and 0 otherwise. The parameter RT in the summation bound for the total number of highest-ranked files that are examined for the calculation of AP for a given query Q. We set $RT = 100$. The denominator rel_Q is the total number of relevant files in the collection for Q. MAP is computed by taking the mean of the average precisions for all the queries.

The parameters λ for HLM and μ for DLM need to be tuned according to the characteristic of the underlying collection. Figure 5.3 and Figure 5.4 plot the retrieval accuracies in terms of MAP for various values of these parameters. For retrievals with HLM, we obtained the highest baseline accuracy with $\lambda = 0.8$ which assigns a higher weight to the file likelihoods as compared to the collection likelihoods of the query terms. On the other hand, the optimum value of μ for DLM is 2400, although the accuracies are not very sensitive to the variations in the $[2000 - 4000]$ interval. The constants a_1 and a_2 in the Tf-Idf model are set to 1.2 and 1.0 respectively. In all of the experiments with DLM, HLM and Tf-Idf, we used these fixed values for the parameters to solely compare the improvements with the proposed defect and modification based models for the priors.

Comparison of the Retrieval Models

Table 5.3 presents the baseline retrieval performance across the models without incorporating the defect or the modification probabilities. That is, we assume a *uniform* prior for the results in Table 5.3. While DLM performs the best amongst the models, HLM performs the worst. The models InB2, InExpB2 and InL2 from the DFR framework do not require parameter tuning while performing as well as and sometimes better than the models in LM.

Table 5.4 presents the hypotheses we have formulated to investigate as to what extent using the defect and modification priors influences the retrieval of the files likely to be defective. The retrieval performance results themselves are presented in Tables 5.5, 5.6, 5.7 and 5.8. We use pairwise student's t-test for significance testing. The columns p-H1, p-H1a, p-H2 and p-H2a in the tables give the computed p-value of the pairwise significance tests for the corresponding hypotheses. The highest score in each column is given in bold. We also report the improvement percentages for MAP and P@1 compared to the baseline results. Highest improvement in each column is designated by the '*' character.

Table 5.3 Baseline Retrieval Results

Model	MAP	P@1	P@5	R@5
HLM ($\lambda = 0.8$)	0.1174	0.0859	0.0715	0.1607
DLM ($\mu = 2400$)	**0.1349**	**0.1271**	**0.0851**	**0.1642**
InB2	0.1240	0.1233	0.0774	0.1491
InExpB2	0.1327	0.1237	0.0735	0.1485
InL2	0.1268	0.0993	0.0719	0.1555
Tf-Idf ($a_1 = 1.2$, $a_2 = 1.0$)	0.1264	0.1062	0.0712	0.1488

Table 5.4 Hypotheses

H1	Using the prior modification probabilities of the files (MHbP) in a software project enhances the bug localization accuracy.
H2	Using the prior defect probabilities of the files (DHbP) in a software project enhances the BL accuracy.
H1a	MHbPd outperforms MHbP when they are employed in IR based bug localization.
H2a	DHbPd outperforms DHbP when they are employed in IR based bug localization.
H3	Prior defect history is superior to prior modification history when they are employed in IR based BL.

Table 5.5 Retrieval performances across the models with MHbP.

Model	MAP (Imp.)	P@1 (Imp.)	P@5	p-H1a
HLM	0.1318 (+12.27%)	0.0825 (-3.96%)	0.0859	8.64e-07
DLM	**0.1556** (+15.34%)	**0.1375** (+8.18%)	0.0907	7.30e-11
InB2	0.1329 (+6.83%)	0.1306 (+5.58%)	0.0818	2.90e-03
InExpB2	0.1447 (+9.04%)	0.1271 (+2.75%)	0.0838	2.49e-04
InL2	0.1481 (+16.43%)	0.1203 (+20.66%)*	**0.0914**	1.43e-04
Tf-Idf	0.1508 (+18.93%)*	0.1271 (+19.34%)	0.0893	4.61e-06

Table 5.6 Retrieval performances across the models with MHbPd ($\beta_1 = 1.0$).

Model	MAP (Imp.)	P@1 (Imp.)	P@5	p-H1a
HLM	0.1924 (+63.88%)	0.1856 (+116.07%)	0.1313	4.14e-08
DLM	0.1896 (+40.55%)	0.2131 (+67.66%)	0.1340	5.11e-05
InB2	0.1704 (+36.98%)	0.1924 (+55.54%)	0.1141	4.91e-05
InExpB2	0.1975 (+48.83%)	0.2268 (+83.35%)	0.1265	4.24e-06
InL2	0.2007 (+57.78%)	0.2337 (+134.40%)*	0.1388	3.07e-05
Tf-Idf	**0.2121** (+67.27%)*	**0.2474** (+132.30%)	**0.1416**	3.50e-06

Table 5.7 Retrieval performances across the models with DHbP.

Model	MAP (Imp.)	P@1 (Imp.)	P@5	p-H1a
HLM	0.1474 (+25.55%)	0.1100 (+28.06%)	0.0962	9.28e-09
DLM	**0.1660** (+23.05%)	**0.1546** (+21.64%)	0.0928	5.51e-07
InB2	0.1500 (+20.58%)	0.1409 (+13.90%)	0.0907	2.73e-06
InExpB2	0.1592 (+19.97%)	0.1512 (+22.23%)	0.0955	1.18e-05
InL2	0.1640 (+28.93%)	0.1409 (+41.32%)*	0.1031	3.06e-07
Tf-Idf	0.1651 (+30.21%)*	0.1409 (+32.30%)	**0.1065**	2.96e-08

Table 5.8 Retrieval performances across the models with DHbPd ($\beta_2 = 5.0$).

Model	MAP (Imp.)	P@1 (Imp.)	P@5	p-H1a
HLM	0.2114 (+80.07%)*	0.2199 (+156.00%)*	0.1326	3.84e-10
DLM	0.2041 (+51.30%)	0.2268 (+78.44%)	0.1423	1.50e-03
InB2	0.1847 (+48.47%)	0.2165 (+75.02%)	0.1175	1.93e-07
InExpB2	0.2047 (+54.26%)	0.2268 (+83.35%)	0.1320	4.29e-07
InL2	0.2194 (+72.48%)	0.2509 (+151.65%)	0.1471	5.78e-07
Tf-Idf	**0.2258** (+78.08%)	**0.2646** (+148.45%)	**0.1512**	1.77e-07

All of the improvements in the retrieval results are statistically significant at a significance level of 1% i.e. $\alpha = 0.01$, therefore the null hypotheses for H1, H1a, H2 and H2a are rejected. As can be seen in Tables 5.5, 5.6, 5.7 and 5.8, using the defect or the modification priors estimated from the version histories improves the BL accuracies significantly. Especially the amount of improvement in precision is extremely high with highest improvement being 156% in P@1 for the HLM model incorporating DHbPd.

The InL2 model from the DFR framework stands out in the experiments. This model does not need any parameter tuning and it performs comparably well, gaining the highest improvements in P@1 with MHbP, MHbPd and DHbP, and second to the

highest improvement in P@1 with DHbPd. With a MAP value of 0.2258, we obtained the highest retrieval accuracy with the Tf-Idf model incorporating DHbPd. The InL2 model incorporating DHbPd came out as the second best with a MAP of 0.2194.

MHbP vs. DHbP

Intuitively, we expect the BF change-sets to be more descriptive in terms of the defect potential for the files. Indeed, comparing the models using the prior defect history with the models using the prior modification history, we observe that DHbP consistently outperforms MHbP at a significance level of 1% i.e. $\alpha = 0.01$ for all the models except DLM for which the p-value is 0.026, indicating a significance level of 5%. From these results, we conclude that using the defect histories of the files results in superior BL in comparison to the modification histories. Therefore, the null hypothesis for H3 is rejected.

Discussion

It is reasonable to assume that the effects of modification and bug-occurrence events associated with a file should decay with time. Obviously, as developers fix the faulty parts of a software project in response to the reported bugs, some files may have caused the bugs to occur on just a one-off basis, while others may require repeated fixes for the same set of bugs. So, one could argue that the weight given to a file in relation to a given bug with just a one-off occurrence of the bug that was fixed a long time ago should be low and this weight should become even lower with the passage of time. On the other hand, a file requiring repeated fixes for a bug should get a higher weight as being a likely source of the same bug again and this weight should diminish only slowly with time.

As explained in Section 5.1.3, we incorporate time decay through the parameters β_1 and β_2. Figure 5.5 and Figure 5.6 plot the retrieval accuracies for several different

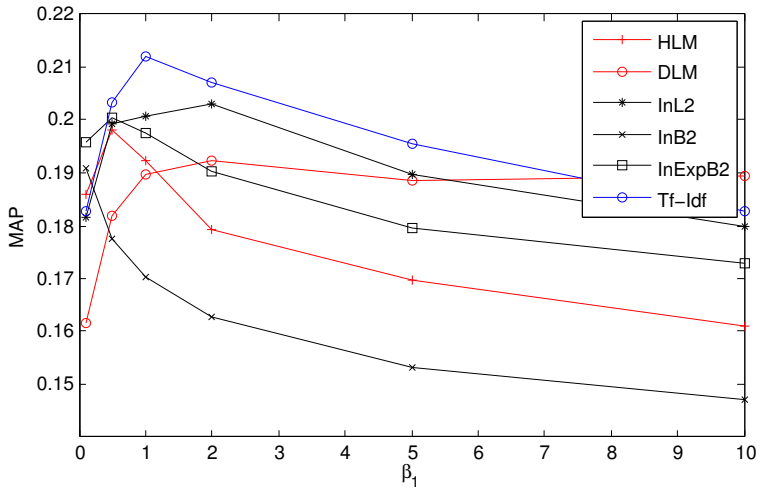


Fig. 5.5. The effect of varying β_1 in MHbPd.

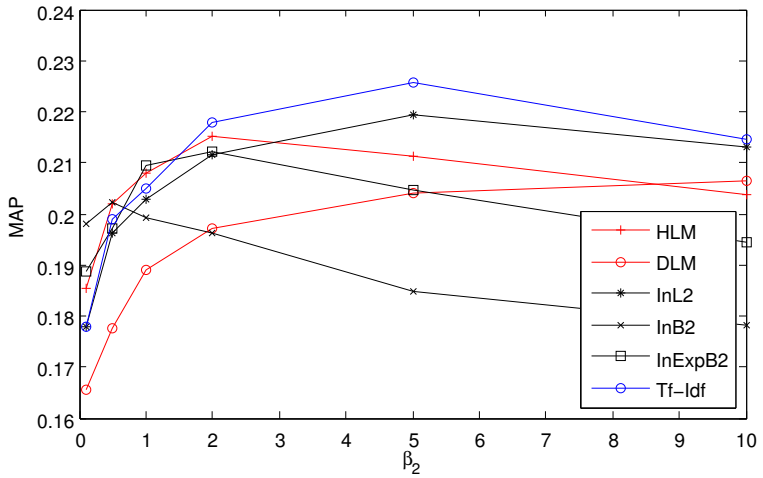


Fig. 5.6. The effect of varying β_2 in DHbPd.

values of these parameters. Retrieval with the Tf-Idf model performed the best with $\beta_1 = 1.0$ and $\beta_2 = 5.0$.

What is interesting is that the optimum value for the decay parameter β_1 is always less than the optimum value for β_2 that resulted in the highest MAP for the analyzed models as can be seen in Figure 5.5 and Figure 5.6. These results suggest that the expected stabilization time of the BF change-sets tends to be longer than that of the non-BF change-sets, i.e. the bug fixes take longer to be finally resolved.

Table 5.9 Recall with InL2 incorporating version histories

Model	R@5 (0.1%)	R@10 (0.2%)	R@25 (0.5%)	R@50 (1%)	R@100 (2%)
MHbP	0.1926	0.2644	0.3966	0.5040	0.6588
DHbP	0.2048	0.2837	0.4189	0.5231	0.6676
MHbPd($\beta_1 = 1.0$)	0.2392	0.3170	0.4478	0.5388	0.6421
DHbPd($\beta_2 = 5.0$)	0.2519	0.3532	0.4750	0.6025	0.6863
Baseline	0.1555	0.2351	0.3468	0.4485	0.5745

Completeness of Retrievals

Table 5.9 presents the recall values at several cut-off points in the ranked list with the proposed models for the priors. The row "Baseline" presents the recall results with a uniform prior. Since the size of the source code is not the same for each bug, we use the average size of the source code to designate the percentage of the code at the reported ranks in parentheses. Here, we only report the results for the InL2 model because of space limitations. By analyzing 1% of the code on the average, 60.25% of the buggy files were localized with the InL2 model incorporating DHbPd.

5.3.3 Comparison with Relevant Work

Using the iBugs dataset, Dallmeier and Zimmerman [88] experimented with FindBugs [10], a static bug pattern detection tool, and Ample [9], a dynamic BL tool. These experiments allow us to indirectly compare our results with those obtained through static and dynamic analysis. Their evaluation shows that FindBugs was not able to locate any of the 369 bugs in the iBugs dataset. Based on these results, we conclude that Terrier+ performs better than FindBugs. On the other hand, the experiments with Ample in [9] are restricted to the 44 bugs that require a single class to be fixed and that have at least one failing test. Ample locates 40% of those bugs

by searching at most 10% of the executed classes. For comparison with our work, notice that P@1 for the Tf-Idf model incorporating DHbPd is 0.2646, which indicates that, for 77 of the 291 bugs, the first file in the retrieved list is actually a relevant file. These results clearly show that our approach works extremely well for a larger set of bugs. For 77 of the bugs, the developers locates the bug without investigating a long list of classes.

Another relevant BL tool is BugScout by Nguyen et al. [38]. Their experiments on AspectJ are restricted to a single collection of 978 files and 271 bug reports. The accuracy of BugScout is reported in terms of *hitrate*. If BugScout correctly retrieves at least one relevant file for a bug in a ranked list of a certain size, it is considered to be a *hit*. The hitrate for a project is given by the ratio of the total number of hits to the total number of bugs. BugScout's hitrate for AspectJ with a ranked list of 10 files is reported as 35%. For a comparison with our work, note that the hitrate with a ranked list of 10 files for the InL2 model incorporating DHbPd is 63.5%, indicating more than 80% improvement. Additionally, P@10 for InL2 incorporating DHbPd in our experiments is 0.1065. This result indicates that, on the average, the developer is guaranteed to locate a relevant file with our approach if s/he is willing to examine the top 10 files.

6. ASSISTING CODE SEARCH WITH AUTOMATIC QUERY REFORMULATION FOR BUG LOCALIZATION

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

The problem of constructing a good query for code search is exacerbated by the fact that, despite the naming conventions in all programming languages, arbitrary abbreviations and concatenations are frequently used in source code for the naming of concepts, objects, artifacts, and so on [91]. Therefore, searching a code base for concepts, objects, artifacts, etc., using terms that are oblivious to the abbreviation and concatenations actually used can, in the worst case, miss out entirely on the files highly relevant to a given search, and, in the best, result in poor values for the relevancies. While an experienced developer — especially one who is already familiar with the code base — may be able to anticipate the peculiarities of the naming conventions used in a software library, it is easy to imagine how much harder it would be for an inexperienced developer to do the same [92].

Obviously, code search needs effective techniques for what is known as *Query Reformulation* (QR). As it turns out, researchers in the software engineering community have looked at QR in the past — *but in a way that either places additional burden on the developer in constructing a query or that are based on general statistical properties of a software library.* To elaborate, as an example of QR that places additional

burden on the developer, we have the study of Explicit Relevance Feedback (ERF) by Gay et al. [40]. They used Rocchio's method for QR to locate a target file by reformulating the original query [41] through an iterative interaction with the user. And, as an example of QR that automatically reformulates a query on the basis of general statistical properties of a library (without regard to the retrieval effectiveness of the terms in the original query), we have the work by Marcus et al. [1]. They used Latent Semantic Indexing (LSI) to determine what terms were more likely to co-occur with what other terms in software artifacts. The term co-occurrence information obtained in this manner was later used to automatically expand a given query that has a single term initially. Along the same lines, Yang and Tan proposed an approach to infer semantically related terms automatically via a pairwise comparison of the code segments and the comments in the source code [93].

In this chapter, we present an *automatic* QR approach that aims to alleviate the difficulty of designing a proper query. The proposed QR method does not require any input from a user for reformulating the query and it is also attuned to the original query. It works by enriching an initial (weak) query with certain specific additional terms drawn from the highest-ranked artifacts retrieved in response to the initial query. The important point here is that these additional terms injected into a query are those that are deemed to be "close" to the original query terms in the source code on the basis of *positional proximity*. This similarity metric, named Spatial Code Proximity (SCP), is based on the notion that terms that deal with the same concepts in source code are usually proximal to one another in the same files. We believe this framework is ideally suited for automatic QR for BL in order to improve the quality of what comes back from an IR-based search engine for the following reasons:

- Bug reports by their very nature contain terms that can be expected to result in the retrieval of a set of documents with a reasonable chance that the set will contain at least some documents directly relevant to the bug. The issue then becomes how to analyze this set of initially retrieved documents for modifications to the original query so that the set of documents retrieved with the

reformulated query will give us improved retrievals. Obviously, while a bug report will contain terms relevant to the intended search task, in general it may also contain terms that are extraneous to what a user is looking for.

- Within the naming conventions that may be recommended for a particular programming language, the programmers are generally free to use any abbreviations and concatenations at all that, at least in their minds, convey some information regarding the purpose served by a name. This obviously presents hurdles in code search for those not already familiar with the code base. And even by those who are familiar with the code base, the relative importance of the different names with regard to their ability to serve as discriminating identifiers for a software artifact may not be well understood.

For the evaluation of the proposed proximity based model, we compared it to the well-known QR methods in the literature with the following Research Questions (RQ) in mind:

- **RQ1:** Does the term proximity based QR technique we propose improve the accuracy of source code retrieval, if so, to what extent?
- **RQ2:** How do the QR techniques that are currently in the literature perform for source code retrieval?
- **RQ3:** How does the initial retrieval performance affect the performance of QR?
- **RQ4:** What are the conditions under which QR may perform poorly?

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

6.1 Background on Relevance Feedback

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

Although the main ideas underlying relevance feedback are straightforward, there is no single feedback strategy that works for different types of queries and for different domains. As a result, relevance feedback has continued to be an active area of research, with its focus being on (1) how to best acquire the feedback; and (2) how to best utilize it to improve the retrieval accuracy [41, 94–96]. In the next two subsections, we provide a brief survey of this research as it has been used in the past for QR in software context. Our comments to follow include a comparison of this framework with ours.

6.1.1 Explicit Relevance Feedback

In explicit relevance feedback (ERF), after seeing the first set of retrieval results returned by the search engine, the user provides his/her judgments *explicitly* by marking the relevant and irrelevant set of results. In [40], Gay et al. showed that ERF can assist developers in locating the target file(s) by reformulating the original query iteratively. At each iteration, the reformulated query is submitted and a new set of retrieval results is presented to the user to obtain the next round of feedback. This process is repeated until all the target files are located.

The most commonly used method for query reformulation is perhaps Rocchio's method. With Rocchio's method, both the query and the documents are regarded as term vectors with the size of the vocabulary V. Given a query Q, and the set

of top X retrieved files D with respect to Q, the user populates the set of relevant files $D_{REL} \subset D$ and the irrelevant files $D_{IR} \subset D$. Then the query is reformulated as follows:

$$Q^{ref} = \alpha \cdot Q + \beta \cdot \sum_{f \in D_{REL}} \frac{f}{|D_{REL}|} - \gamma \cdot \sum_{f \in D_{IR}} \frac{f}{|D_{IR}|}. \tag{6.1}$$

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

6.1.2 Pseudo Relevance Feedback

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

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

well satisfied in the domain of bug localization on account of how the bug reports are generally constructed.

6.2 Past Work on Automatic Query Reformulation

In this Section, we present two currently well-known methods for automatic QR. We include them here since we will use them in a comparative evaluation of automatic QR for BL in Section 6.5.

6.2.1 Rocchio's Formula for Automatic QR

Rocchio's Formula (ROCC) for automatic QR is well known in natural-language document retrieval [41]. It is therefore an automatic candidate to try in software context also. ROCC is based on the following rationale: Use the set D of the top files retrieved for the initial query Q to create a vector space model (VSM) of the term-term and term-document relationships in the set of documents in D. For each word w in D, ROCC calculates a weight that expresses its importance in a reformulated version Q^{ref} of the query Q according to:

$$tf(w, Q^{ref}) = (1 - \beta) \cdot tf(w, Q) + \beta \cdot \sum_{f \in D} \frac{tf(w, f)}{|D|} \tag{6.2}$$

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

6.2.2 Automatic QR Using the Relevance Model

Another approach to automatic QR, again in the natural language context, is the *Relevance Model* (RM) [94, 97]. What makes RM based QR different from Rocchio's formula is that as a formal probabilistic model, it incorporates the ranking scores of the documents in D as an additional component for a more accurate selection of the terms to be used for reformulating the original query. The goal is to estimate a co-occurence probability distribution $P'(w|Q)$ for the terms in the vocabulary vis-a-vis the query terms. Given the first pass retrieval results D, RM estimates this probability by:

$$P'(w|Q) = \frac{P(w,Q)}{P(Q)} \quad \propto \quad \sum_{f \in D} P(w|f)P(f) \prod_{q \in Q} P(q|f) \tag{6.3}$$

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

$$P(w|Q^{ref}) = (1 - \beta) \cdot P(w|Q) + \beta \cdot P'(w|Q) \tag{6.4}$$

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

6.3 The Proposed Approach to Query Reformulation For Source Code Retrieval

The QR methods we described in Section 6.2 consider all the terms in a file $f \in D$ equally for reformulation. However, given the distinct structure and the term

relationships in source code, one would think the importance of a term in a file to an existing term in a query ought to depend on the likelihood that the former normally shows up *proximally* to the latter in the same file. The proximity we are referring to may relate to the different terms appearing in the same long name through an underscore or a camel-case based concatenation. Proximity also refers to the different terms appearing close to one another in the same method or class definition and so on.

A number of different approaches have been proposed to capture the term-term proximity effect in the context of natural-language [95,98,99]. Inspired by these studies, we propose a simple probabilistic model, which we call *Spatial Code Proximity* (SCP) model for measuring term-term positional proximity that is particularly appropriate for source code. SCP estimates a proximity probability distribution $P'(w|Q)$ over each term w in the vocabulary V of the system on the basis of their *proximity frequencies* in D with respect to Q.

In order to use proximity for query reformulation, we index the position of each term in a file as demonstrated in Fig. 6.1. Vis-à-vis a given query Q, we next associate a term proximity frequency with each term in the file f in the following manner: (1) We initialize the value of the proximity frequency to 0 for each term in f. (2) We then identify every position in f where the term is the same as one of the terms in Q. (3) Subsequently, we place a window of size $2W + 1$ at each position identified in the previous step. The window is placed symmetrically around the identified positions. (4) The proximity frequency associated with every term in the window is now incremented by 1. The proximity frequencies (pf) obtained in this manner can be expressed as:

$$pf(w, f|Q) = \sum_{q \in Q} \sum_{i \in P_f[w]} \sum_{j \in P_f[q]} I_W(i, j) \tag{6.5}$$

where $P_f[\cdot]$ is the position operator which returns the set of position indices of a given term in f. The window operator $I_W(i, j)$ returns 1 if the positions i and j fall within the same window of length W:

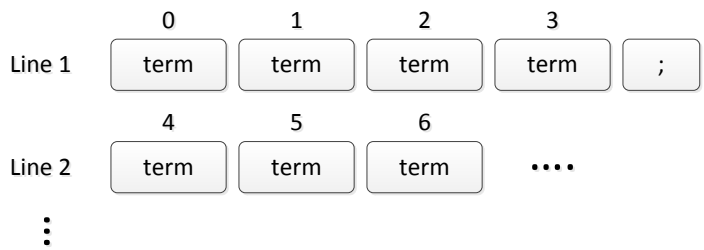


Fig. 6.1. An illustration of indexing the positions of the terms in a source file.

$$I_W(i,j) = \begin{cases} 1 & \text{if } |i-j| \leq W \\ 0 & \text{otherwise.} \end{cases} \tag{6.6}$$

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

$$P'(w|Q) = \frac{\sum_{f \in D} pf(w, f|Q)}{\sum_{w' \in V} \sum_{f \in D} pf(w', f|Q)} \tag{6.7}$$

Using Eq. 6.4, we reformulate the query by interpolating the original query with the proximity probability distribution. With this formulation, the terms that tend to co-occur with the query terms consistently in close proximity across the files in D will receive a high probability mass whereas the terms that appear far away from the query terms will be discarded in QR. Fig. 6.2 presents the Query Reformulation process.

6.3.1 A Motivating Example

The Bug 20750 filed for Google Chrome[1] has a title that reads: "drag a loading tab will cause the loading animation separate with the tab". The target file that has been modified to fix this bug is "chrome/browser/gtk/tabs/tab_strip_gtk.cc". The reformulated versions of this query along with the normalized term frequencies as estimated by the three QR methods are given in Table 6.1.

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

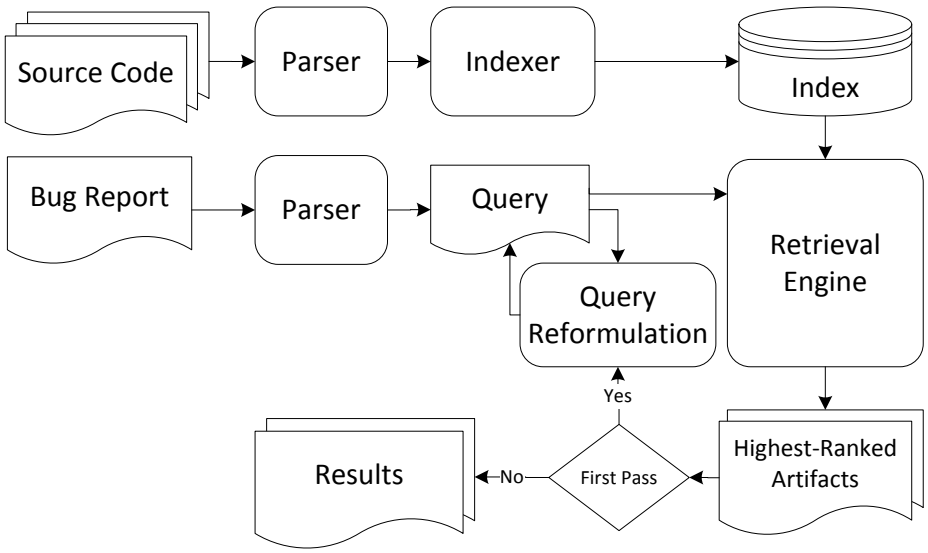


Fig. 6.2. An illustration of data flow in QR process.

Table 6.1 QR achieved with the proposed SCP method vis-à-vis the same achieved with the ROCC and RM methods for the Bug 20750 filed for the Chrome project.

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

Looking at the relevant file, we see that the most important terms of the query to retrieve this file are tab, animation and load; while the terms cause and separate are not related to the information need primarily. As can be seen in Table 6.1, all three methods are able to capture this notion of relevancy by reweighing the original query terms accordingly. Additionally, the QR methods are also able to extract relevant terms from the initial retrieval results to expand the query. For instance, the terms gtk and strip are good expansion terms that are conceptually related to the query. Interestingly, differing from the other methods, SCP extracts the terms tabstrip, bound and pin from the feedback files. These proximity terms are the key terms that help the SCP method achieve a perfect average precision.

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

6.4 Retrieval & Evaluation Framework

6.4.1 Retrieval Model

Obviously, the underlying retrieval function has a prominent effect on QR since it is first used to rank the files in the collection to obtain the set D of feedback files. Then, after the reformulation, a second pass of retrieval with respect to the updated query is performed to obtain the final retrieval set. For QR experiments, we use the TF-IDF framework [14] as our baseline retrieval model where the score of a query with respect to a file is computed by the following weight function:

$$P(Q|f) \propto score_{\text{TFIDF}}(Q, f) = \sum_{q \in Q} tf(q, f) \cdot idf(q). \tag{6.8}$$

We compute the frequency of a term in a file by Robertsons's TF and the inverse document frequency of the terms by Spark Jones' IDF. See Section 4.3 for the definitions of these frequencies. Robertson's TF contains two model parameters, a_1 and a_2. We empirically set $a_1 = 1.2$ and $a_2 = 0.75$ for the QR experiments.

6.4.2 Query Performance Prediction (QPP) Metrics

To evaluate the effect of QR on the quality of the retrievals, we employ the following QPP metrics [18, 100]:

Average Inverse Document Frequency (avgIDF)

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

$$avgIDF(Q) = \sum_{q \in Q} log_2[|C|/(N_q + 1)]/|Q|. \tag{6.9}$$

Average Inverse Collection Term Frequency (avgICTF)

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

$$avgICTF(Q) = \sum_{q \in Q} log_2[NT/(tf(q, C) + 1)]/|Q| \tag{6.10}$$

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

Query Scope (QS)

In [100], Query Scope (QS) is defined as the percentage of the documents that contain at least one query term. A small value of QS for a query with respect to a given collection indicates that the query is discriminatory. Note that as the length of the queries increase this metric becomes less effective in measuring the query specificity.

Simplified Clarity Score (SCS)

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

$$SCS(Q) = \sum_{q \in Q} P(q|Q) log_2[P(q|Q)/P(q|C)]. \tag{6.11}$$

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

6.5 Experimental Evaluation

Our goal in this section is to compare the power of query reformulation that is based on the spatial code proximity analysis with two other approaches drawn from natural language research. In the graphs and tables that show comparative results, SCP will denote the new "Spatial Code Proximity" method, RM the method based on "Relevance Model", and, finally, ROCC the method based on a direct application of Rocchio's formula for QR. Our comparative results are based on two large software projects, namely Eclipse[2], the Integrated Development Environment (IDE) and

[2]www.eclipse.org

Google Chrome[3], the WEB Browser. In order to evaluate the presented algorithms, we need a set of bug reports, B and the source files modified to fix the corresponding bugs as the ground truth. As mentioned earlier, unfortunately, the bug tracking databases such as Bugzilla[4] do not store the actual modifications committed for the reported bugs. The common approach to link the modifications to the bug reports is to look for pointers in the commit messages to the bug tracking database [39,88,101]. For Eclipse, we follow a similar approach and employ regular expressions for an accurate reconstruction of the links between the bug reports and the corresponding modifications as follows:

1. Scanning the repository logs, group the files that are modified by the same author with the same commit message with a time fuzziness of 200 seconds [89]. This step is necessary as CVS repositories store the changes made to each file separately.

2. For the target version of the software, use regular expressions to extract the bug IDs from the commit messages. Our regular expressions match the following generic phrases in the commit messages: Fix for ID, Fix ID, Fixed ID, Fixing ID, Bug ID and they are insensitive to the punctuation characters and the spacing within the identified phrase i.e. the phrases such as BugID, Bug: ID or Bug #ID and so on are also matched.

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

For Google Chrome, the commit messages follow a more specific form. They contain a separate line to indicate whether the commit fixes any bugs and if so the bug IDs are given in that line. Note that a bug may need more than one set of modifications to be finally resolved. Therefore, we accumulate the set of modifications that are committed for the same bug over the analysis period. Finally, we consider only the

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

Table 6.2 Evaluated Projects

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

bug reports that are marked as "FIXED" in the bug tracking database, which means a fix has been checked into the tree and tested for the corresponding bug. The result of this reconstruction process is a dataset that we call *BUGLinks*[5]. Table 6.2 presents the evaluated projects from this dataset and some of their statistics.

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

6.5.1 Indexing Source Code

We use the same pre-processing steps as described in Chapter 5 to index the code base of each project. Firstly, the compound terms are tokenized using punctuation characters and camel case characters. The tokens thus generated are subject to a set of stemming and stopping rules. We use Porter's stemming algorithm to trim the terms to their common root forms [90]. Regarding the stopping rules, these delete the programming-language specific terms and the terms in a list of standard-English

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

stop-words that has 733 words in it. The positions of each term in the files are recorded for the SCP method after these pre-processing steps.

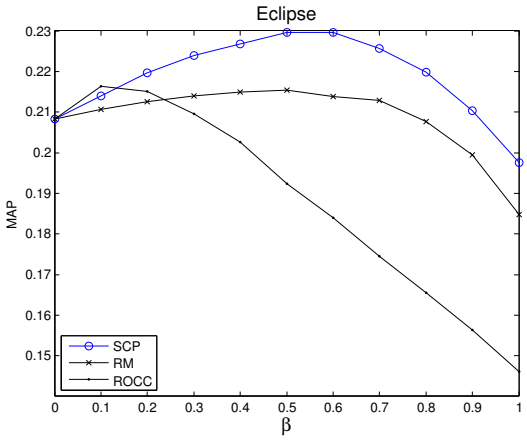
The bug reports are subject to the same preprocessing steps as the source files. Each bug report is also divided into two parts: its title and its description. The title of a bug report constitutes an explanation, albeit very brief, of the buggy behavior, with further elaboration provided by the description part. For QR experiments, we use the titles of the bug reports to populate the set of queries B as they are less noisy and they resemble the real world search queries better than the descriptions.

6.5.2 Evaluating the Query Characteristics

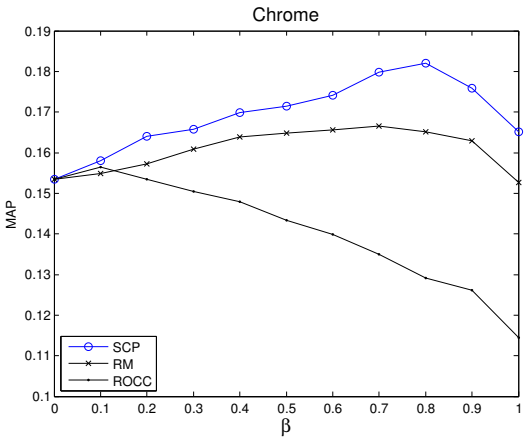
Note that any QR strategy can only be expected to give us improvements on the average, implying that we must admit to the possibility of a reformulated query giving us worse results on occasion [95]. We define the set of queries for which QR leads to improvements as *positive* queries and denote this set by B^+. Similarly, the set of queries for which QR results in a decreased performance is defined as *negative* queries and it is denoted by B^-. The remaining queries in B preserve their initial performance after QR. We consider these queries as constituting the set of *neutral* queries and we denote it by B^o. Obviously, $|B^+| + |B^-| + |B^o| = |B|$. As mentioned in Section 6.4, we use the QPP metrics to analyze the characteristics of these query sets. For comparing QPP metrics on these sets, we used unpaired two-sample t-test.

6.5.3 Parameter Sensitivity Analysis

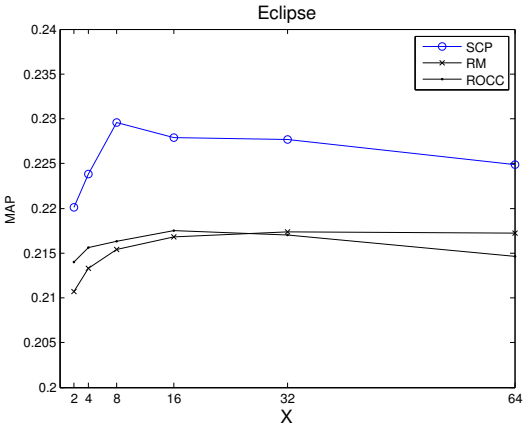
The four experimental parameters that affect the quality of retrieval from reformulated queries are: (1) The interpolation parameter β that determines the weight to be accorded to the new terms to be added to a query vis-a-vis the original terms; (2) The number of highest ranked files to be analyzed for QR; (3) The number of terms to be added to the original query; and, finally, (4) The size W of the window to



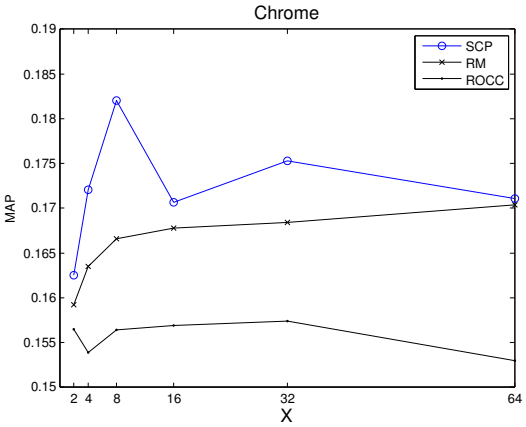
(a) The interpolation parameter



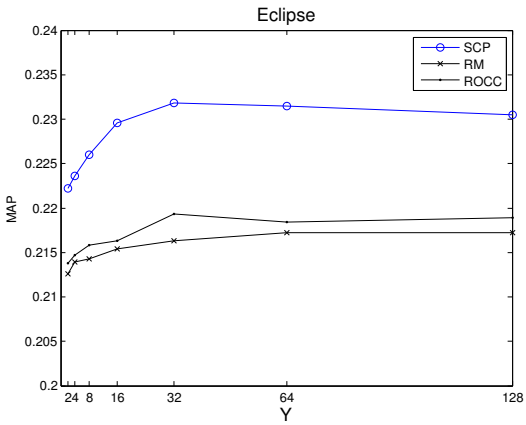
(b) The interpolation parameter



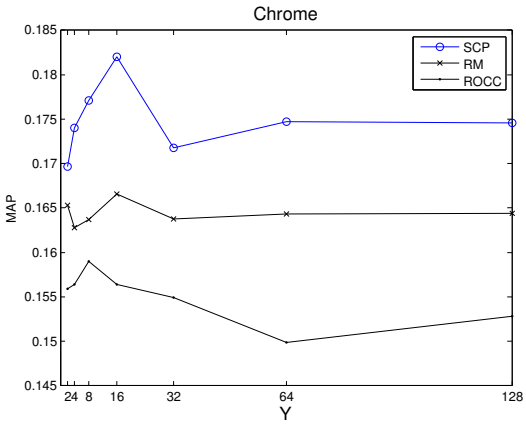
(c) The number of top files



(d) The number of top files



(e) The number expansion terms



(f) The number expansion terms

Fig. 6.3. The effects of varying the experimental parameters.

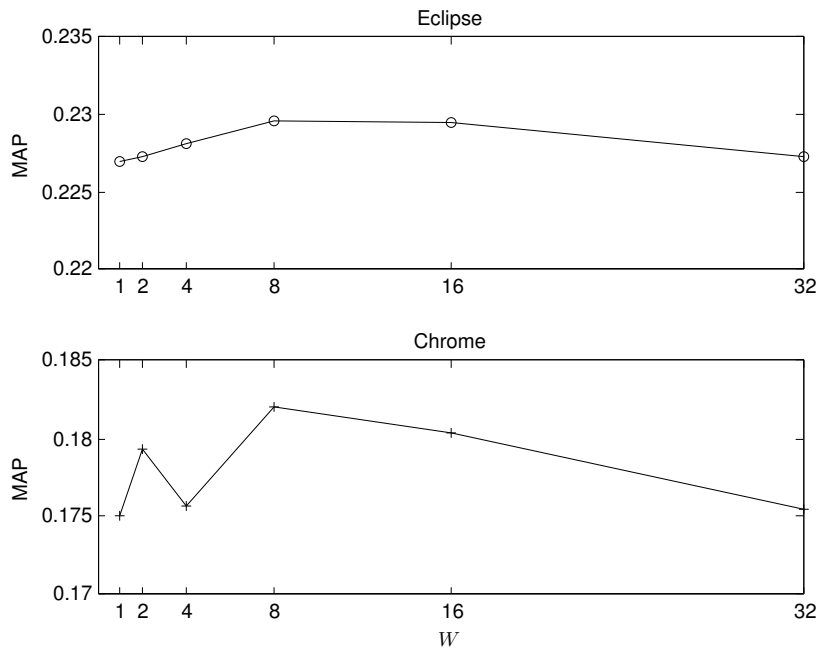


Fig. 6.4. The effect of varying window size W.

be used for proximity analysis. Fig. 6.3 and 6.4 show how the retrieval performance depends on these four parameters.

With regard to β, as can be seen in Fig. 6.3(a) and 6.3(b), while ROCC reaches its peak point for $\beta = 0.1$, RM and SCP achieve better accuracies for higher values of β. Fig. 6.3(c), 6.3(e), 6.3(d) and 6.3(f) present the retrieval accuracies as the number of feedback files and the feedback terms to expand the queries are varied for $X \in \{2, 4, 8, 16, 32, 64\}$ and $Y \in \{2, 4, 8, 16, 32, 64, 128\}$ for the analyzed projects. Fixing these parameters at certain chosen values empirically is a common approach to evaluating the QR methods [97]. Following this tradition, we set these parameters as $X = 8$ and $Y = 16$ unless stated otherwise. Note that SCP outperforms the other two methods consistently in these experiments. In addition to these parameters common to all three methods, Fig. 6.4 plots the effect of varying the window length W for the SCP method. Interestingly, in both software projects, we reached the best retrieval accuracy for $W = 8$. Notice that the retrieval accuracy starts to degrade as far away terms are considered for QR.

Table 6.3 Retrieval accuracy with QR for the three QR methods on Eclipse

	MAP	P@1	P@5	R@5	R@10	p-value
SCP	**0.2296**	**0.1893**	**0.1011**	**0.2849**	**0.3739**	1.03e-17
RM	0.2154	0.1670	0.0970	0.2729	0.3631	1.56e-04
ROCC	0.2163	0.1713	0.0961	0.2743	0.3663	4.01e-05
Baseline	0.2089	0.1653	0.0921	0.2632	0.3559	↑

Table 6.4 Retrieval accuracy with QR for the three QR methods on Chrome

	MAP	P@1	P@5	R@5	R@10	p-value
SCP	**0.1820**	**0.1788**	**0.0933**	**0.2021**	0.2775	0.0023
RM	0.1666	0.1480	0.0844	0.1966	**0.2853**	0.0140
ROCC	0.1564	0.1397	0.0793	0.1816	0.2779	0.5369
Baseline	0.1535	0.1453	0.0832	0.1838	0.2601	↑

Table 6.5 QR for positive (B^+), negative (B^-) and neutral (B^o) queries on Eclipse.

Retrieval Accuracy for B^+

	MAP (Imp.)	P@1	P@5	R@5	R@10	$\|B^+\|$
SCP	0.2530 (+66%)	0.2168	0.1325	0.3041	0.4405	1,674
Baseline	0.1524	0.0920	0.0957	0.2086	0.3501	
RM	0.2128 (+45%)	0.1707	0.1286	0.2669	0.4009	1,459
Baseline	0.1463	0.1165	0.0949	0.1755	0.3103	
ROCC	0.2143 (+44%)	0.1794	0.1206	0.2600	0.3920	1,449
Baseline	0.1486	0.0994	0.0962	0.1935	0.3320	

Retrieval Accuracy for B^-

	MAP (Imp.)	P@1	P@5	R@5	R@10	$\|B^-\|$
SCP	0.1314 (-40%)	0.0681	0.0821	0.1965	0.3046	955
Baseline	0.2205	0.1853	0.1087	0.2723	0.3870	
RM	0.1228 (-35%)	0.0577	0.0704	0.1659	0.2902	1,057
Baseline	0.1898	0.1258	0.0986	0.2556	0.3870	
ROCC	0.1372 (-34%)	0.0764	0.0803	0.1925	0.3110	942
Baseline	0.2065	0.1741	0.1011	0.2474	0.3588	

Retrieval Accuracy for B^o

	MAP	P@1	P@5	R@5	R@10	$\|B^o\|$
SCP	0.2682	0.2390	0.0767	0.3220	0.3416	1,406
RM	0.2823	0.2396	0.0852	0.3530	0.3776	1,519
ROCC	0.2634	0.2184	0.0835	0.3337	0.3752	1,644

Table 6.6 QR for positive (B^+), negative (B^-) and neutral (B^o) queries on Chrome.

Retrieval Accuracy for B^+

	MAP (Imp.)	P@1	P@5	R@5	R@10	$\|B^+\|$
SCP	0.2655 (+90%)	0.2911	0.1544	0.2934	0.4071	158
Baseline	0.1401	0.1392	0.0937	0.1715	0.2841	
RM	0.1999 (+50%)	0.2179	0.1244	0.2200	0.3392	156
Baseline	0.1337	0.1795	0.0872	0.1296	0.2352	
ROCC	0.1604 (+32%)	0.1656	0.0954	0.1557	0.2964	151
Baseline	0.1219	0.1126	0.0861	0.1290	0.2380	

Retrieval Accuracy for B^-

	MAP (Imp.)	P@1	P@5	R@5	R@10	$\|B^-\|$
SCP	0.0901 (-51%)	0.0495	0.0554	0.0988	0.1833	101
Baseline	0.1850	0.1683	0.1149	0.2247	0.3140	
RM	0.0932 (-38%)	0.0306	0.0510	0.1061	0.2318	98
Baseline	0.1508	0.0816	0.1061	0.2032	0.3053	
ROCC	0.1397 (-30%)	0.0988	0.0914	0.1874	0.2928	81
Baseline	0.1988	0.2222	0.1259	0.2466	0.3230	

Retrieval Accuracy for B^o

	MAP	P@1	P@5	R@5	R@10	$\|B^o\|$
SCP	0.1426	0.1313	0.0343	0.1616	0.1667	99
RM	0.1857	0.1538	0.0558	0.2468	0.2548	104
ROCC	0.1622	0.1349	0.0524	0.2090	0.2460	126

6.5.4 Retrieval Results & Discussions

Table 6.3 and 6.4 present the average performance of the three QR methods. The baseline retrieval accuracy without QR is presented in the last row of the tables. The best retrieval accuracy in each column is given in bold. The column "p-value" gives the p-value of the pairwise significance testing with respect to the baseline accuracy. Table 6.5 and 6.6, on the other hand, present the retrieval accuracies on the sets B^+, B^- and B^o. For each QR method, we present the baseline performance (with the original queries in each set) and the performance of the reformulated queries along with the size of the corresponding query set. All of the differences between the baselines and the performances of the respective QR methods in this table are statistically significant at $\alpha = 0.01$.

It is obvious from the results presented in Table 6.3 and 6.4 that the queries for the Eclipse project perform much better than the queries for the Chrome project, indicating that there is a higher correlation between the terms used in the bug reports and the source code of Eclipse in comparison to Chrome. This could be due to the differences in the naming conventions of the respective programming languages. In the rest of this section, we will now provide answers to the four questions listed at the end of Introduction.

Answer to RQ1

Based on the results in Table 6.3 and 6.4, we conclude that QR, in general, leads to significant improvements over the baseline retrieval accuracy for BL. Even more significantly, the improvements obtained with the new SCP approach are noticeably higher than those obtained by the other QR methods. While all three QR methods lead to large improvements for the Eclipse project at a significance level of $\alpha = 0.01$, only the proposed SCP method achieves a significant improvement on the Chrome project at this confidence level.

Looking at the sizes of the query sets B^+, B^- and B^o, as presented in Table 6.5 and 6.6, we observe that SCP either preserves or improves the performance for 76% of the 4,393 queries for the analyzed projects. Additionally, Table 6.5 and 6.6 show that the SCP method improves a larger number of queries in comparison to the other two QR methods. Notice that the amount of improvement is also very large for the positive queries with SCP, becoming as large as 66% for the Eclipse project and 90% for the Chrome project in terms of MAP.

Answer to RQ2

This question is regarding the effectiveness of the two natural-language based approaches to QR, the one based on Relevance Model (RM) and the other based on using the Rocchio's formula (ROCC). Based on the results presented in Table Table 6.3 and 6.4, we conclude that these do improve the retrieval results for BL. Of the two, RM is a robust performer; it achieves a significantly superior retrieval accuracy over the baseline for both projects. However, the improvements obtained via ROCC are not significant on the Chrome project. The main difference between these two methods is that RM incorporates the ranking scores of the files in the first pass retrieval into the QR process. With the help of these scores, the higher the rank of a file in the first pass retrieval, the higher the weights of the terms in these files for QR. Note that because of this weighting, the performance of RM is also less sensitive to the number of feedback files X.

Answer to RQ3

In order to answer this question, we analyze the sets B^+, B^- and B^o for their differences in terms of the retrieval performance. As shown in Table 6.5 and 6.6, the initial retrieval performance has a significant correlation with the QR performance. Note that the baseline retrieval accuracy for the B^+ queries is significantly lower compared to that of the B^- queries for all QR methods. From these results, we may

Table 6.7 Query Statistics on the query sets as constructed by the SCP method on Eclipse

	$\|Q\|$	avgICTF	avgIDF	QS	SCS	#RF
B^+	5.80	$10.19^{\dagger\dagger}$	$3.94^{\dagger\dagger\dagger}$	$0.49^{\dagger\dagger}$	$7.78^{\dagger\dagger\dagger}$	3.09
B^-	5.90^{**}	$10.02^{\dagger\dagger***}$	$3.80^{\dagger\dagger\dagger**}$	$0.51^{\dagger\dagger}$	$7.57^{\dagger\dagger\dagger****}$	3.69^{****}
B^o	5.72^{**}	10.20^{***}	3.91^{**}	0.50	7.83^{****}	1.73^{****}

Table 6.8 Query Statistics on the query sets as constructed by the SCP method on Chrome

	$\|Q\|$	avgICTF	avgIDF	QS	SCS	#RF
B^+	6.35	11.38	4.03	0.53	8.84	$5.17^{\dagger\dagger}$
B^-	6.14	11.22^{*}	3.98^{*}	0.51	8.73	$3.13^{\dagger\dagger*}$
B^o	6.44	11.61^{*}	4.27^{*}	0.50	9.10	2.36^{*}

conclude that QR improves poorly performing queries, while it may lead to retrieval degradation for the queries that are performing relatively well to begin with. Also note that a large number of queries preserve their initial performance and the average retrieval accuracies of those queries are also high. Especially, P@1 for Eclipse is significantly high for these sets.

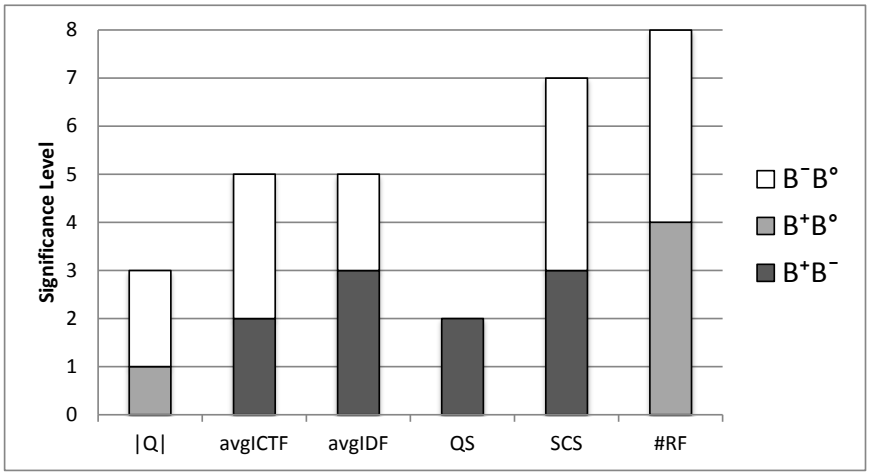
Answer to RQ4

With regard to this question, Table 6.7 and 6.8 present the mean values for the QPP metrics together with the average query lengths ($|Q|$) and the average number of Relevant Files (#RF) per bug for the respective query sets. We only present these metrics for the query sets as obtained via the new SCP method for lack of space. However, we obtained similar results for all three methods. We use the '†' symbol to indicate the statistical significance in the differences between the sets B^+ and B^-,

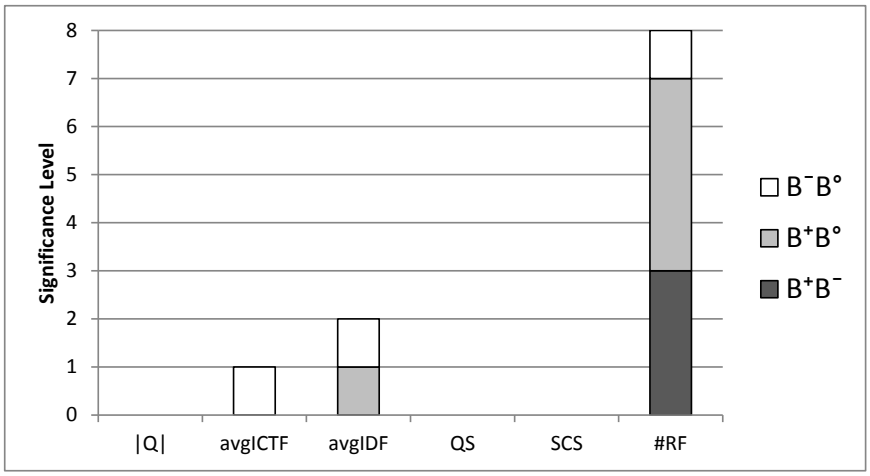
and the '$*$' symbol for the same purpose between the sets B^- and B^o. The number of symbols indicates the level of significance at 10%, 5%, 1% and 0.1% respectively. That is, one such symbol indicates a 10% level of significance, two symbols indicate 5%, and so on. Additionally, Fig. 6.5 shows the significance levels of the differences for a pairwise comparison on the query sets. The heights of the bars in these plots give the significance level i.e. the number of symbols. We do not show the significance levels for the pairwise comparison of the sets B^+ and B^o in Table 6.7 and 6.8 in order to keep the table uncluttered. See Fig. 6.5 for these differences.

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

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



(a) Eclipse



(b) Chrome

Fig. 6.5. Significance levels on the differences between the query sets obtained with the new SCP method. The heights of the bars indicate how significant the difference is: Longer bars mean more significant at the significance levels of 10%, 5%, 1% and 0.1%.

Remarks

The retrieval results for the query sets B^+, B^- and B^o reveal many important characteristics of QR for BL. These results clearly show that QR with the evaluated methods improves poorly performing queries. Obviously, then, we can claim that the presented QR framework helps out when it is most needed by a user. Additionally, looking at Table 6.5 and 6.5, we observe that if the original query is already performing well, QR is likely to just preserve the accuracy rather than to decrease it on average. This is evident from the fact that compared to the negative queries, the number of the neutral queries is larger and the average retrieval accuracies of those queries are also high. Based on the differences presented in Table 6.7 and 6.8, we may conclude that these properties of QR are more likely to occur as the discriminatory power of the queries increases.

7. EXPLOITING SOURCE CODE PROXIMITY AND ORDER WITH MARKOV RANDOM FIELDS

This chapter focuses on query representation especially when long queries such as bug reports are directly used for automatic Bug Localization (BL). In the widely used *bag-of-words* representations for both the queries and the source code documents, all positional and ordering relationships between the terms are lost [16, 37, 39]. This is tantamount to a serious loss of information considering that bug reports, in general, are a composition of structured and unstructured textual data that frequently includes (a) patches; (b) stack traces when the software fault throws an exception; (c) snippets of code; (d) natural language sentences; and so on [20, 21]. Patches and stack traces, especially, contain vital proximity and ordering relationships between the terms that ought to be exploited for the purpose of retrieval. Say, if two terms are proximal to each other in a query patch or stack trace, you'd want a source code file containing similar code to have the same two terms in a similar proximal relationship. It therefore stands to reason that the quality of retrieval for BL would improve if these structural elements could be taken into account when searching for the most relevant files.

As to how to incorporate ordering and positional relationships in a retrieval framework, we have several possibilities at our disposal that have been examined in the past mostly in the context of retrieval from natural language corpora. At one end of the spectrum, we have *ad hoc* approaches such as those that compute term co-occurrence frequencies and proximity-based term-term dependencies. And, at the other end of the spectrum, we have more principled approaches, such as those based on Markov Random Fields (MRF) [15] that are based on modeling the term-term dependencies by graphs whose arcs capture the inter-term relationships in the queries vis-a-vis the same in the documents.

With regard to the investigation of such approaches in retrieval from software libraries, in Chapter 6 we presented an *ad hoc* approach that demonstrated how the inter-term proximities can be used to reformulate the queries in order to improve the quality of retrievals. We now show that even more significant improvement in retrieval precision can be obtained by using a more principled alternative to *ad hoc* approaches. In particular, we will show that when an MRF is used to model the ordering and the positional dependencies between the query terms vis-a-vis the documents, we end up with a framework that not only yields a higher retrieval precision with the simplest of the ordering and proximity constraints, but that can also be generalized to the investigation of more general such constraints. In the MRF based approach, certain subsets of the terms in a bug report are used for scoring the software artifacts while taking into account term-term proximity and order. This approach exploits the fact that the software artifacts that contain the query terms in the same order and/or in similar proximities as in the query itself are more likely to be relevant to a given query.

While, as demonstrated by our results, MRF is a powerful approach to the modeling of query-document relationships, to fully exploit its potential it must be used in conjunction with what we refer to as *Query Conditioning* (QC). The goal of QC is to recognize the fact that a bug report constitutes a highly structured query whose various parts consist, as we mentioned previously, of a textual narrative, a stack trace, etc [20]. These different parts are disparate in the sense that the inter-term relationships do not carry the same weight in them. For example, the proximity of the terms used in the stack trace portions of a bug report carries far more weight than in the textual narrative. Therefore, the ordering and proximity constraints are likely to be far more discriminative in those portions of bug report that contain structural elements. To further underscore the importance of these structured portions of the bug reports, past studies have shown that stack traces and patches are the most valuable source of information to pinpoint the location of the bugs [102, 103].

Note that whereas Terrier+ applies MRF modeling to all queries, QC becomes an important factor only when structural elements are present in the queries. In general, detecting the structural elements in bug reports is a difficult task as they may have different formats and they are usually surrounded by other types of textual data [20]. It is also not uncommon for these constructs to undergo unexpected format changes, such as those caused by accidental line breaks, when they are copied into a bug report. In order to overcome these challenges, Terrier+ employs several regular expressions to detect and extract these structural elements from bug reports.

7.1 Research Questions

Our empirical evaluation on large open-source software projects shows that the proposed retrieval framework leads to significant improvements in automatic BL accuracy. We compare the performance of Terrier+ to the other IR approaches developed for the retrieval of the buggy source files. Among them is the Spatial Code Proximity (SCP) based Query Reformulation (QR) which is presented in Chapter 6. Another important class of IR tools developed for automatic BL leverage the past development efforts for superior accuracies [16, 39]. Our Bayesian retrieval framework presented in Chapter 5 is such an approach that has been also shown to improve the BL accuracy significantly [16].

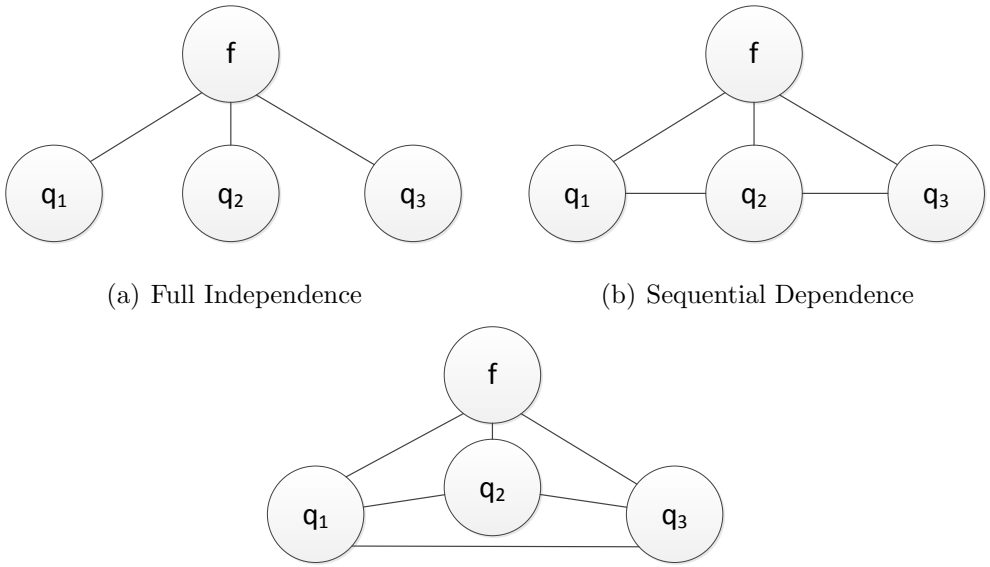
For the evaluation of Terrier+, we conducted extensive validation tests with the following Research Questions (RQ) at hand:

RQ1: *Does including code proximity and order improve the retrieval accuracy for BL. If so, to what extend?*

RQ2: *Is QC effective on improving the query representation vis-a-vis source code?*

RQ3: *Does including stack traces and/or patches in the bug reports improve the accuracy of the BL?*

RQ4: *How does the MRF-based retrieval framework compare to the Query Reformulation (QR) based retrieval frameworks for BL?*



(a) Full Independence (b) Sequential Dependence

(c) Full Dependence

Fig. 7.1. Markov Networks for Dependence Assumptions for a file and a query with three terms.

RQ5: *How does the MRF-based retrieval framework compare to the other BL frameworks that leverage the past development efforts?*

7.2 Markov Random Fields

Over the years, researchers in the machine learning community have devoted much energy to the investigation of methods for the probabilistic modeling of arbitrary dependencies amongst a collection of variables. The methods that have been developed are all based on graphs. The nodes of such graphs represent the variables and the arcs the pairwise dependencies between the variables. The graphs may either be directed, as in Bayesian Belief Networks [104], or undirected, as in the networks derived from Markov Random Fields [15, 104]. In both these methods, the set of variables that any given variable directly depends on is determined by the node connectivity patterns. In a Bayesian Belief Network, the probability distribution at a node q is conditioned on only those nodes that are at the tail ends of the arcs incident on q, taking the

causality into account. In a Markov Network, on the other hand, the probability distribution at a node q depends on the nodes that are immediate neighbors of q without considering any directionality. In the context of retrieval from natural language corpora, the work of Metzler and Croft [15] has shown that Markov Networks are particularly appropriate for the modeling of inter-term dependencies vis-a-vis the documents.

In general, given a graph G whose arcs express pairwise dependencies between the variables, MRF modeling of the probabilistic dependencies amongst a collection A of variables is based on the assumption that the joint distribution over all the variables in the collection can be expressed as product of non-negative potential functions over the cliques in the graph:

$$P(A) \quad = \quad \frac{1}{Z}\prod_{k=1}^{K}\phi(C_k) \tag{7.1}$$

where $\{C_1, C_2, \ldots, C_K\}$ represents the set of all cliques in the graph G, and $\phi(C_k)$ a non-negative potential function associated with the clique C_k. In the expression above, Z is merely for the purpose of normalization since we want the sum of $P(A)$ over all possible values that can be taken by the variables in A to add up to unity.

Our end goal with MRF is to rank the files in the code base according to the probability of a file f in the software library to be relevant to a given query Q. As mentioned in Chapter 5, we denote this probability by $P(f|Q)$. Using the definition of the conditional probability, we can write

$$P(f|Q) = \frac{P(Q,f)}{P(Q)}. \tag{7.2}$$

As we are only interested in ranking the files and the denominator in Eq. 7.2 does not depend on files, it can be ignored. Hence $P(f|Q) \stackrel{rank}{=} P(Q,f)$. In order to separate out the roles played by the variables that stand for the query terms (since we are interested in the inter-term dependencies in the queries) vis-a-vis the contents of a source file f, as suggested by Metzler and Croft [15], we will use the following variation of the general form expressed in Eq. 7.1 to compute this joint probability:

$$P(Q,f) \;=\; \frac{1}{Z}\prod_{k=1}^{K}\phi(C_k) \;\overset{rank}{=}\; \sum_{k=1}^{K} log(\phi(C_k)) \tag{7.3}$$

where Q is assumed to consist of the terms $q_1, q_2, ..., q_{|Q|}$. The nodes of the graph G in this case consist of the query terms, with one node for each term. G also contains a node that is reserved for the file f whose relevancy to the query is in question. As before, we assume that this graph contains the cliques $\{C_1, C_2, \ldots, C_K\}$. As shown in the formula, for computational ease it is traditional to express the potential $\psi(C_k)$ through its logarithmic form, that is through $\psi(C_k) = log(\phi(C_k))$.

The fact that a fundamental property of any Markov network is that probability distribution at any node q is a function of only the nodes that are directly connected to q may now be expressed as

$$P(q_i|f, q_{j\neq i} \in Q) = P(q_i|f, q_j \in neig(q_i)) \tag{7.4}$$

where $neig(q_i)$ denotes the terms whose nodes are directly connected to the node for q_i. As observed in [15], this fact allows arbitrary inter-term relationships to be encoded through appropriate arc connections amongst the nodes that represent the query terms in the graph G. At one end of the spectrum, we can assume that the query terms are all independent of one another by the absence of any arcs between them. This assumption, known as the usual bag-of-words assumption in information retrieval, is referred to as the *Full Independence (FI)*. And at the other end of the spectrum, we may assume a fully connected graph in which the probability distribution at each node representing a query term depends on all the other query terms (besides being dependent on the file f). This is referred to as the *Full Dependence (FD)*. Fig. 7.1(a) and 7.1(c) depict the graph G for FI and FD assumptions for the case when a query Q consists of exactly three terms.

What makes MRF modeling particularly elegant is that it gives us a framework to conceptualize any number of other "intermediate" forms of dependencies that are between the two extremes of the FI and the FD assumptions. This we can do by simply choosing graphs G of different connectivity patterns. Whereas FI is based on

the absence of any inter-term arcs in G and FD on there being an arc between each query term and every other term, we may now think of more specialized dependencies such as the one depicted in Fig. 7.1(b). This dependency model, referred to as the *Sequential Dependency (SD)* model in [15], incorporates both order and proximity between a sequence $(q_1, q_2, \ldots, q_{|Q|})$ of query terms.

At this point, the reader may wonder as to how one would know in advance as to which connectivity pattern to use for the graph G. The connectivity pattern is obviously induced by the software library itself. Suppose a phrase level analysis of the files in the library indicates that the phrase "interrupt sig handler" occurs in the files and *can be used to discriminate between them*, you would want the nodes for the terms "interrupt," "sig," and "handler" to be connected in the manner shown in Fig. 7.1(b). This is because the SD model shown in that figure would only allow for pairwise (but ordered) occurrences of the words "interrupt," "sig," and "handler" to be matched in the files. The frequencies with which these ordered terms appear in the files may also carry discriminatory power. The relative importance of the words occurring individually or in ordered pairs would be determined by their relative frequencies in the files. In contrast to the case depicted in Fig. 7.1(b), should it happen that the queries and the relevant files contain the three terms "interrupt," "sig," and "handler" in all possible orders, you would want to use the FD assumption depicted in Fig. 7.1(c). In this case, the number of times these terms occur together within a window of a certain size would carry discriminatory power for choosing the files relevant to a query. Finally, should it happen, that the three terms occur in the relevant files without there being a phrasal sense to their appearance in the files, you would want to use the bag-of-words, FI, assumption.

We are particularly interested in the graph connectivity induced by the notion of Spatial Code Proximity (SCP) that we introduced in Chapter 6 [19]. SCP consist of first associating a positional index with each term in a query and in the documents as shown in Fig. 7.2. Our goal is to translate the values of the positional indexes into graph models based on the FI, SD, and FD assumptions. In the next three

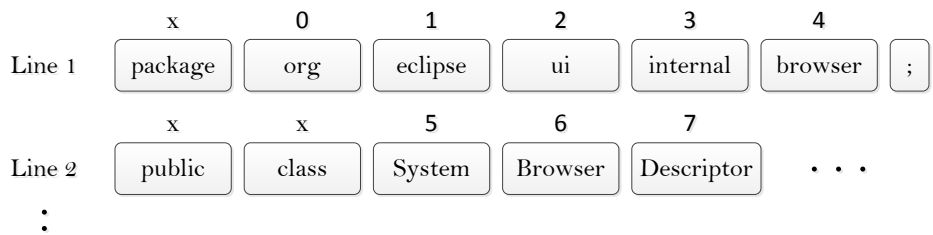


Fig. 7.2. An illustration of indexing the positions of the terms in an Eclipse Class: SystemBrowerDescriptor.java. The 'x' symbol indicates the stop-words that are dropped from the index.

subsections, we will present formulas that show how these models can be derived from SCP based indexes.

7.2.1 Full Independence (FI)

As already stated, the FI assumption reduces an MRF model to the usual bag-of-words model that has now been extensively investigated for automatic BL [16,37,39]. As should be clear from the graph representation of this model depicted in Fig. 7.1(a) for the case of a query with exactly three terms, FI modeling involves only 2-node cliques. Therefore, under MRF modeling, the probability of a query given a file is simply computed by summing over the 2-node cliques: $P_{FI}(f|Q) \stackrel{rank}{=} \sum_{i=1}^{|Q|} \psi_{FI}(q_i, f)$. The choice of the potential function, obviously critical in computing this probability, should be in accord with the fact that MRF under FI assumption amount to the bag-of-words modeling. Therefore, a good choice for the potential $\psi_{FI}(q_i, f)$ is the Dirichlet Language Modeling (DLM) as explained in Chapter 4. Shown below is a formula for the potential $\psi_{FI}(q_i, f)$ that includes Dirichlet smoothing:

$$\psi_{FI}(q_i, f) \;=\; \lambda_{FI} log\Big(\frac{tf(q_i, f) + \mu P(q_i|C)}{|f| + \mu}\Big) \tag{7.5}$$

where $P(q_i|C)$ denotes the probability of the term in the whole collection, $tf(q_i, f)$ is the term frequency of q_i in a file f, $|f|$ denotes the length of the file and μ is the Dirichlet smoothing parameter. The model constant λ_{FI} has no impact on the

rankings with this model. However, we keep it in the formulation as we will use it later in SD and FD modeling.

The probability expression shown above for the relevance of a term to a file is exactly the same as it appears in the widely used bag-of-words model known as the *Smoothed Unigram Model (SUM)* whose usefulness in automatic bug localization has been demonstrated in [16, 37, 39]. We will use the retrieval results obtained with FI as the baseline in order to determine the extent of improvements one can obtain with the other two models, SD and FD.

7.2.2 Sequential Dependence (SD)

The SD model takes the order and the proximity of the terms into account in such a way that the probability law for a query term q_i given a file f obeys $P(q_i|f, q_j \in \{q_1, \ldots, q_{i-1}, q_{i+1}, \ldots, q_{|Q|}\}) = P(q_i|f, q_{i-1}, q_{i+1})$.

To see how a software library can be processed to induce the SD model, note from the example shown in Fig. 7.1(b) that we now have 3-node cliques in addition to the 2-node cliques of the FI model. Therefore, we must now count the frequencies with which pairs of terms occur together, with one following the other (without necessarily being adjacent) in a specific order, in addition to counting the frequencies for the terms occurring singly as in the FI model [105]. Again incorporating Dirichlet smoothing, we employ the following potential function for the 3-node cliques corresponding to a file f and two consecutive query terms q_{i-1} and q_i:

$$\psi_{SD}(q_{i-1}, q_i, f) = \lambda_{SD} log\Big(\frac{tf_W(q_{i-1}q_i, f) + \mu P(q_{i-1}q_i|C)}{|f| + \mu}\Big) \tag{7.6}$$

where $tf_W(q_{i-1}q_i, f)$ is the number of times the terms q_{i-1} and q_i appear in the same *order* as in the query within a window length of $W \geq 2$ in the file. For $W > 2$, the terms do not have to be adjacent in the file and the windows may also contain other query terms. The smoothing increment $P(q_{i-1}q_i|C)$ is the probability associated with the pair $(q_{i-1}q_i)$ in the entire software library. To the potential function shown above,

we must now add the potential function for 2-node cliques the reader has already seen for the FI model:

$$P_{SD}(f|Q) \stackrel{rank}{=} \sum_{i=2}^{|Q|} \psi_{SD}(q_{i-1}, q_i, f) + \sum_{i=1}^{|Q|} \psi_{FI}(q_i, f). \tag{7.7}$$

As the reader would expect, the ranking of the files with the potential function shown in Eq. 7.7 is only sensitive to the relative weights expressed by the model parameters λ_{FI} and λ_{SD}, the overall scaling of these weights being inconsequential on account of the unit summation constraints on probabilities. We therefore set $\lambda_{FI} + \lambda_{SD} = 1$. We can think of λ_{SD} as an interpolation or a mixture parameter that controls the relative importance of the 3-node cliques vis-a-vis the 2-node cliques.

7.2.3 Full Dependence (FD)

As demonstrated previously by Fig. 7.1(c), the FD assumption implies a fully connected graph G whose nodes correspond to the individual query terms, with one node being reserved for the file f under consideration. The graph being fully connected allows for a file f to be considered relevant to a query regardless of the order in which the query terms occur in the file. (Compare this to the SD case where, for a file f to be considered relevant to a query, it must contain the query terms in the same order as in the query.) Therefore, the FD assumption provides a more flexible matching mechanism for retrievals.

The price to be paid for the generality achieved by FD is the combinatorics of matching all possible ordering of the query terms with the contents of a file. To keep this combinatorial explosion under control, following [105], we again limit ourselves to just 2-node and 3-node cliques. While this may sound the same as for the SD assumption, note that the 3-node cliques are now allowed for a pair of query terms for both ordering of the terms. Therefore, for any two terms q_i and q_j of the query, the potential function takes the following form for the 3-node cliques:

$$\psi_{FD}(q_i, q_j, f) = \lambda_{FD} log\Big(\frac{tf_W(q_i q_j, f) + \mu P(q_i q_j|C)}{|f| + \mu}\Big) \tag{7.8}$$

where λ_{FD} again works as a mixture parameter similar to λ_{SD}, i.e. $\lambda_{FI} + \lambda_{FD} = 1$; μ is the smoothing parameter and $tf_W(q_i q_j, f)$ is the frequency for the pair $q_i q_j$ in f. Summing over the cliques, we obtain the ranking score of a file by

$$P_{FD}(f|Q) \overset{rank}{=} \sum_{i=1}^{|Q|} \sum_{j=1, j \neq i}^{|Q|} \psi_{FD}(q_i, q_j, f) + \sum_{i=1}^{|Q|} \psi_{FI}(q_i, f). \tag{7.9}$$

7.2.4 A Motivating Example

We will now use a simple example to compare the retrieval effectiveness of the three models, FI, SD, and FD. We will limit our example to the simplest of the bug reports, one that only contains a one-line text narrative which corresponds to the title of the bug report.

The Bug 98995[1] filed for Eclipse v3.1 has a title that reads: *"Monitor Memory Dialog needs to accept empty expression"*. The target source files that were eventually modified to fix this bug are:

1. org.eclipse.debug.ui/.../ui/views/memory/MonitorMemoryBlockDialog.java
2. org.eclipse.debug.ui/.../ui/views/memory/AddMemoryBlockAction.java
3. org.eclipse.debug.ui/.../ui/DebugUIMessages.java.

After removing the stop-words from the title, the final query consists of seven unique terms. Table 7.1 presents the retrieval accuracies obtained for this query, along with the number of cliques utilized for each dependency assumption. In the table, the column "Rank" gives the ranks of the three relevant files in the ranked lists retrieved. AP is the resulting Average Precision.

Investigating the ranked lists returned for the three models, we see that FI ranks several irrelevant files above the relevant ones. One such file is ASTFlattener.java.

[1]https://bugs.eclipse.org/bugs/show_bug.cgi?id=98995

Table 7.1 Retrieval accuracies for the Bug 98995 with three different MRF models.

Method	2-node cliques	3-node cliques	Ranks	AP
FD	7	42	1-3-2	1.0000
SD	7	6	2-3-4	0.6389
FI	7	0	6-5-10	0.2778

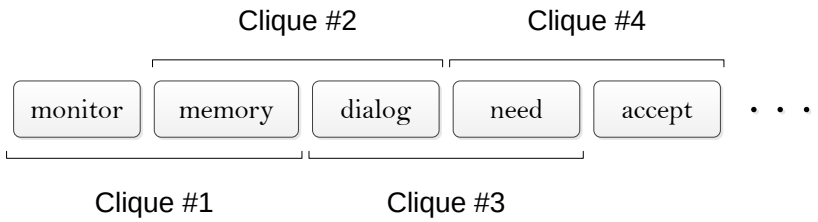


Fig. 7.3. Clique creation for the Bug 98995. The figure shows the first 4 query term blocks for the 3-node cliques utilized by the SD modeling.

Although this file does not contain any of the terms "monitor", "memory" and "dialog"; it is retrieved at the top rank by this model because, as a file related to parsing the *Abstract Syntax Trees (AST)*, it contains the terms "accept", "empty" and "expression" with very high frequencies. Clearly, the model misses the context of the query.

In comparison to FI, SD is able to retrieve the relevant files at higher ranks, as shown in Table 7.1. The improvement obtained with SD is a consequence of the discriminations achieved by requiring that the query terms, when they appear together in a source file, do so in a specific order. The creation of the 3-node cliques with the query terms is illustrated in Fig. 7.3. A 3-node clique is formed by the two words depicted together with an under-bracket and the node corresponding to a file. Since the relevant files contain these term blocks in close proximity with high frequencies; with this model, they receive higher ranking scores in comparison to the irrelevant files.

Despite the improvements, SD still ranks one irrelevant file, ASTRewriteFlattener.java, above all the relevant ones. This file also does not contain any of the

terms “monitor”, “memory” and “dialog”. However, it contains in close proximity the term pairs from the 2 of the 3-node cliques: “accept empty” and “empty expression”. The file manages to receive a high ranking score with these term pairs in addition to the AST related terms.

FD captures the context of the query better than the other two models by considering all the term pairs in the query regardless of their position and order. It assumes that any pair of query terms can depend on one another, hence the number of cliques it uses is higher. This modeling approach ranks the three relevant files at the top ranks above any irrelevant files and reaches a perfect average precision of 1.0.

7.3 Query Conditioning

When a bug report contains highly structured components, such as a stack trace and/or a source code patch [20], such information can be crucial to locating the files relevant to the bug [103]. Being highly structured, these components must first be identified as such and subsequently processed to yield the terms that can then be used to form a query for IR based retrieval. The processing steps needed for that purpose will be different for the two different types of components we consider: stack traces and source code patches. We refer to the collection of these steps as *Query Conditioning (QC)*. QC is carried out with a set of regular expressions that, while custom designed for the different types of structured components encountered, are sufficiently flexible to accommodate small variations in the structures.[2]

As already stated, our retrieval framework has been packaged as an enhancement to the popular research IR retrieval engine Terrier and we refer to our enhancement as Terrier+. With regard to the flow of processing related to QC in Terrier+, it uses regular expressions to first identify the patches and the stack traces from a given bug report if any of these elements are available in the report. Then, it processes

[2]Our QC only takes into account the stack traces and source code patches when they can be identified in a bug report. Note that a bug report may also contain additional source code snippets that are not meant to be patches [20]. QC treats any additional such code on par with the main textual part of the report.

```
java.util.EmptyStackException
  at java.lang.Throwable.<init>(Throwable.java)
  at java.util.Stack.peek(Stack.java)
  at java.util.Stack.pop(Stack.java)
  at org.eclipse.jdt.internal.debug.eval.ast.engine.Interpreter.pop(Interpreter.java:89)
  at org.eclipse.jdt.internal.debug.eval.ast.instructions.Instruction.popValue(Instruction.java:111)
  at org.eclipse.jdt.internal.debug.eval.ast.instructions.PushFieldVariable.execute(PushFieldVariable.java:54)
  at org.eclipse.jdt.internal.debug.eval.ast.engine.Interpreter.execute(Interpreter.java:50)
      ⋮
  at org.eclipse.core.internal.jobs.Worker.run(Worker.java:66)
```

Fig. 7.4. The stack trace that was included in the report for Bug 77190 filed for Eclipse. With QC, the trace is first detected in the report with regular expression based processing. Subsequently, the highlighted lines are extracted as the most likely locations of the bug and fed into the MRF framework.

them separately to sift out the most relevant source code identifiers to be used in the retrievals. The final query is composed from the terms extracted from the stack traces and the patches if one or both of these components are available. If these structured components are not available, Terrier+ makes do with the entire bug report such as it is and feeds it into the MRF framework.

Stack Traces

When Terrier+ detects the stack traces in a bug report, it automatically extracts the most likely locations of the bug by identifying the methods in the trace. As the call sequence in a stack trace starts from the most recent method call,[3] we extract only the topmost T methods while discarding the rest of the trace since the methods down in the trace have a very little chance of containing any relevant terms and they are likely to introduce noise into the retrieval process. Fig. 7.4 illustrates the stack trace that was included in the report for Bug 77190 filed for Eclipse[4]. The bug caused the *EmptyStackException* to be thrown by the code in PushFieldVariable.java and was subsequently fixed in a revised version of this code. The figure highlights

[3]We do realize that for some languages the methods in a stack trace are in the opposite order. That is, the most recent method call appears as the last entry in the trace. The logic of identifying the methods most relevant to a bug would obviously need to be reversed for such languages.

[4]https://bugs.eclipse.org/bugs/show_bug.cgi?id=77190

the extracted portion of the stack trace that is used in forming the final query. Note that we only extract the methods that are present in the code base to which the bug report applies. That is, we skip the methods from the libraries belonging to the Java platform itself, as illustrated in the figure. During the experiments, we empirically set $T = 3$, as this setting resulted in the best retrieval accuracies on the average. As we show in our experimental evaluation, this filtering approach increases the precision of the retrievals significantly.

Patches

Source code patches are included in a bug report when a developer wishes to also contribute a possible (and perhaps partial) fix to the bug. When contributed by an experienced developer, these components of a bug report can be directly used for pinpointing the files relevant to a bug.

A patch for a given bug is usually created with the *Unified Format* to indicate the differences between the original and the modified versions of a file in a single construct. With this format, the textual content of the patch contains the lines that would be removed or added in addition to the contextual lines that would remain unchanged in the file after the patch is applied. For term extraction from the patches, Terrier+ does not use the lines that would be added after the suggested patches are applied to the files as those lines are not yet present in the code base.

Obviously, the files mentioned by a developer in a patch may not correspond to the actual location of the bug. And, there may be additional files in the code base that may require modifications in the final fix for the bug. While the importance of information in such source code patches cannot be overstated, it is important to bear in mind that their inclusion in the bug reports is more the exception than the rule. Out of the 4,035 bug reports we analyzed for Eclipse v3.1, only 8 contained a patch. Along the same lines, out of the 291 bug reports we analyzed for the AspectJ project, only 4 contained a patch. Nonetheless, considering the importance of the information

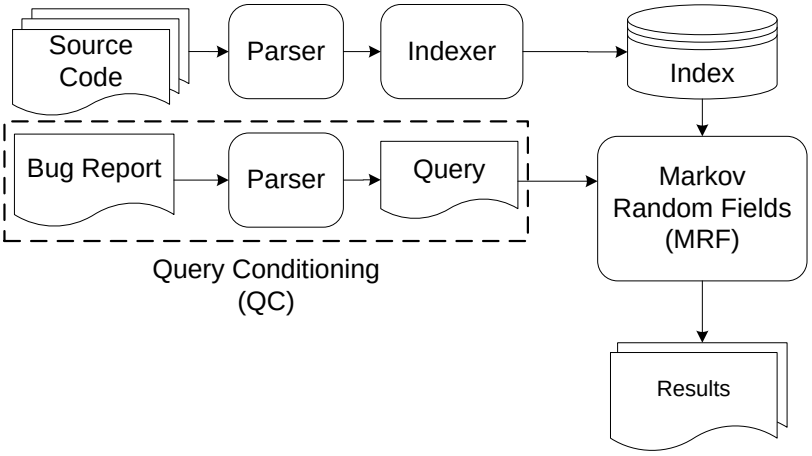


Fig. 7.5. An illustration of the data flow in the proposed retrieval framework.

Table 7.2 Evaluated Projects

Project	Language	$\|B\|$	$\|RF\|$	$\|Q_{Title}\|$
AspectJ	Java	291	3.09	5.78
Eclipse v3.1	Java	4,035	2.76	5.80
Chrome v4.0	C/C++	358	3.82	6.21

contained in the patches when they are included in a bug report, Terrier+ takes advantage of that information whenever it can.

Fig 7.5 illustrates the data flow in the retrieval framework with QC and MRF.

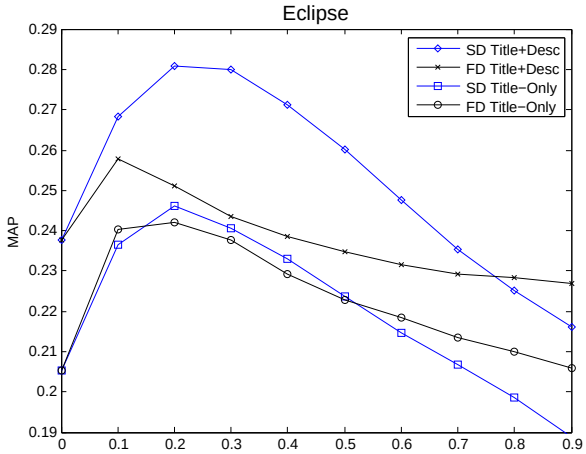
7.4 Experimental Evaluation

We evaluate the effect of incorporating term dependencies on the retrievals for BL on three large software projects, namely Eclipse IDE[5], AspectJ[6] and Google Chrome[7]. We evaluate Query Conditioning (QC) on only Eclipse and AspectJ as the bug reports for Chrome do not contain stack traces or patches. Table 7.2 and 7.3 present the evaluated projects from these datasets and some of their statistics. In Table 7.2, $|B|$ gives the number of bug reports used in querying the code base of each project, $|RF|$ denotes the average number of relevant files per bug and $|Q_{Title}|$ gives the average lengths of the bug report titles that are used in the retrievals. Table 7.3, on the other hand, presents the statistics of the bug reports used in the evaluation of the MRF framework along with QC. In the table, #Patches and #Stack Traces show the number of bug reports containing patches and stack traces respectively and $|Q_{Title+Desc}|$ is the average lengths of the bug reports, including both the title and the description parts without any filtering, in terms of the number of tokens used in querying the code base.

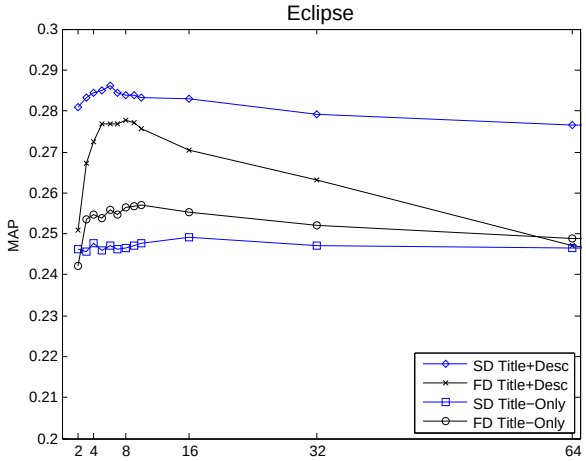
[5]www.eclipse.org
[6]http://eclipse.org/aspectj/
[7]www.google.com/chrome

Table 7.3 Statistics of the bug reports used in the experiments for MRF and QC.

Project	#Patches	#Stack Traces	$\lvert Q_{Title+Desc} \rvert$
AspectJ	4	81	56.77
Eclipse v3.1	8	519	44.11

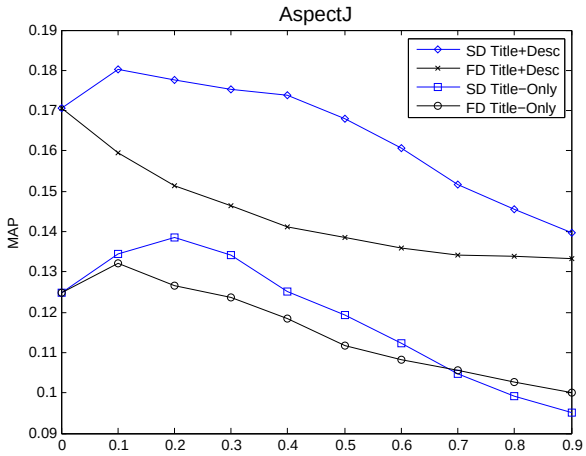


(a) The mixture parameters

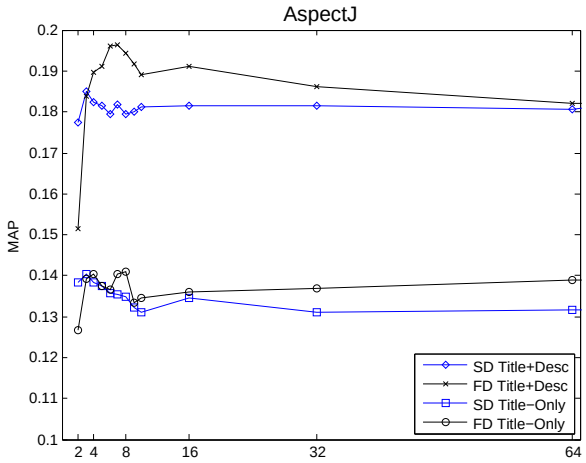


(b) Window Length (W)

Fig. 7.6. The effects of varying model parameters on MAP for Eclipse. The figure on the left shows the MAP values as the mixture parameters (λ_{SD} for SD assumption and λ_{FD} for FD assumption) are varied while the window length parameter is fixed as $W = 2$. The figure on the right shows the MAP values as W is varied while the mixture parameters are fixed at 0.2.



(a) The mixture parameters



(b) Window Length (W)

Fig. 7.7. The effects of varying model parameters on MAP for AspectJ. Same as Fig. 7.6

7.4.1 Bug Localization Experiments

For an in-depth analysis of the retrievals, we divide each bug report into two parts, namely *Title* and *Description*. We first conduct two sets of experiments using these two parts for each bug report *without* Query Conditioning (QC): (1) Retrievals with MRF modeling using only the titles of the bug reports. The queries used for these retrieval are denoted "title-only". And (2) Retrieval with MRF modeling using the complete bug reports, that is, including both the titles and the descriptions for the bug reports. The queries used for these retrievals are denoted "title+desc". Then, we incorporate QC in the second category of retrievals and analyze the usefulness of including stack traces and patches in queries by comparing the overall retrieval accuracy for the set of bug reports that contain these elements to the remaining set of the bug reports in our query sets.

Parameter Sensitivity Analysis

The model parameters that affect the quality of the retrievals in our retrieval framework are: (1) The window length parameter (W). (2) The mixture parameters of the respective dependency models (λ_{SD}, λ_{FD}). And (3) The Dirichlet smoothing parameter (μ). While W sets the upper bound for the number of intervening terms between the terms of the 3-node cliques, λ_{SD} and λ_{FD} simply adjust the amount of interpolation of the scores obtained with the 2-node cliques with those obtained with the 3-node cliques as explained in Section 7.2. As the ranking is invariant to a constant scaling in the mixture parameters, we enforce the constraints $\lambda_{FI} + \lambda_{SD} = 1$ and $\lambda_{FI} + \lambda_{FD} = 1$ for the SD and the FD modeling, respectively. In all experiments, we empirically set the Dirichlet smoothing parameter as $\mu = 4000$. Note that the retrieval accuracy is not very sensitive to the variations on this parameter [16].

Fig. 7.6 and 7.7 plot the retrieval accuracies for bug localization in terms of MAP as the window length and the mixture parameters are varied for the "title-only" and the "title+desc" queries. As shown in Fig. 7.6(a) and 7.7(a), a value of

Table 7.4 Retrieval accuracy with the "title-only" queries on Eclipse.

Method	MAP	P@1	P@5	R@5	R@10	H@10
FD	**0.2564 (+24.83%)**	**0.2198**	**0.1110**	**0.3199**	**0.4070**	**2,100**
SD	0.2466 (+20.06%)	0.2116	0.1069	0.3083	0.3934	2,042
FI	0.2054	0.1710	0.0883	0.2556	0.3417	1,805

Table 7.5 Retrieval accuracy with the "title-only" queries on AspectJ.

Method	MAP	P@1	P@5	R@5	R@10	H@10
FD	**0.1410 (+12.89%)**	**0.1409**	**0.0832**	**0.1794**	**0.2420**	124
SD	0.1348 (+7.93%)	0.1340	0.0790	0.1675	0.2382	**125**
FI	0.1249	0.1375	0.0708	0.1498	0.2079	111

0.2 consistently works well for the mixture parameters in general. Note that when $\lambda_{SD} = \lambda_{FD} = 0.0$, SD and FD use only the 2-node cliques hence they reduce to FI, the Smoothed Unigram Model (SUM). As for the window length, Fig. 7.6(b) and 7.7(b) illustrate the effect of varying this parameter for $\lambda_{SD} = \lambda_{FD} = 0.2$. On average, $W = 8$ results in the best retrieval accuracies for the analyzed projects for both types of queries.

As shown in Fig. 7.6(a) and 7.7(a), when the window length is set as $W = 2$, SD performs better than FD in all experiments across the projects. This is because the terms are required to be adjacent to be matched in the code base when we use this setting and therefore the order of the terms becomes more important. Interestingly, as the window length increases, FD catches up with SD. Overall, SD is less sensitive to the window length parameter, achieving high retrieval accuracies.

Table 7.6 Retrieval accuracy for the "title+desc" queries on Eclipse.

Method	MAP	P@1	P@5	R@5	R@10	H@10
FD	0.2778 (+16.87%)	0.2496	0.1201	0.3427	0.4317	2,249
SD	**0.2840 (+19.48%)**	**0.2543**	**0.1232**	**0.3530**	**0.4391**	**2,268**
FI	0.2377	0.2020	0.1060	0.3046	0.3859	2,044

Table 7.7 Retrieval accuracy for the "title+desc" queries on AspectJ.

Method	MAP	P@1	P@5	R@5	R@10	H@10
FD	**0.1945 (+14.08%)**	**0.2131**	**0.0997**	**0.2322**	0.2996	142
SD	0.1794 (+5.22%)	0.1856	0.0990	0.2203	**0.3096**	**148**
FI	0.1705	0.1856	0.0880	0.2003	0.2693	126

Table 7.8 Retrieval accuracy for the "title+desc" queries with Query Conditioning (QC) on Eclipse.

Method	MAP	P@1	P@5	R@5	R@10	H@10
FD	**0.3019 (+17.93%)**	0.2696	0.1274	0.3709	**0.4640**	**2,386**
SD	0.3014 (+17.74%)	**0.2704**	**0.1290**	**0.3749**	0.4599	2,354
FI	0.2560	0.2186	0.1114	0.3263	0.4102	2,147

Table 7.9 Retrieval accuracy for the "title+desc" queries with Query Conditioning (QC) on AspectJ.

Method	MAP	P@1	P@5	R@5	R@10	H@10
FD	**0.2307 (+8.31%)**	**0.2715**	**0.1155**	**0.2703**	0.3400	161
SD	0.2263 (+6.24%)	0.2646	0.1148	0.2658	**0.3554**	**167**
FI	0.2130	0.2440	0.1052	0.2438	0.3215	147

Retrieval Results

In this section, we compare the retrieval performances of the dependence models (SD and FD) to the Full Independence (FI) model. We fix the interpolation parameters as $\lambda_{SD} = \lambda_{FD} = 0.2$ and the window lengths as $W = 8$ in all of the experiments presented in the remainder of this Chapter.

Table 7.4 and 7.5 present the BL accuracies on the evaluated projects for the "title-only" queries. With these experiments we explore the retrieval accuracy of Terrier+ for short queries comprising only a few terms and no patches or stack traces. The last row of the table shows the "baseline" accuracy which is obtained with the FI assumption or the Smoothed Unigram Model (SUM) [16,37,39]. The highest score in each column is given in bold. All the improvements reported in this table obtained with the dependency models over FI are statistically significant at $\alpha = 0.05$. Note that incorporating the term dependencies into the retrievals improves the accuracy of BL substantially in terms of the 6 metrics presented in the table.

Table 7.6 and 7.7 present the BL accuracies for the "title+desc" queries without QC. That is, the whole textual content of the bug reports are used in querying the code base. The reported improvements obtained with FD and SD over FI are also statistically significant at $\alpha = 0.05$ in this table. Note that the retrieval accuracies improves greatly when the description part of the bug reports are also included in retrievals even without QC. While SD and FD perform comparably well in these experiment on the Eclipse project, FD outperforms SD on AspectJ on average in terms of MAP.

We are now in a position to answer the first of the five Research Questions (RQ) formulated in Section 7.1 of this chapter. For convenience, here is the question again:

RQ1: *Does including code proximity and order improve the retrieval accuracy for BL. If so, to what extend?*

Based on the results presented in Tables 7.4, 7.5, 7.6, 7.7, 7.8 and 7.9, we conclude that incorporating the spatial code proximity and order into the retrievals improves

the accuracy of automatic BL significantly. On average, for both short and long queries which may contain stack traces and/or patches, SD and FD modeling consistently enhance the retrieval performance of Terrier+ over FI across the projects. The improvements are up to 24.83% for the Eclipse project and up to 14.08% for the AspectJ project in terms of MAP.

The effect of QC on Retrievals

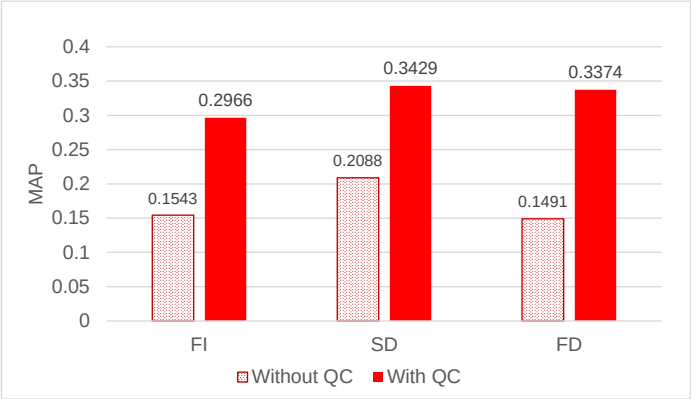
The retrieval accuracies obtained with QC on the "title+desc" queries are presented in Table 7.8 and in 7.9, where each bug report is first probed for stack traces and patches in order to extract the most useful source code identifiers to be used in BL. The results shown in these tables help us answer the following research question that was presented in Section 7.1.

RQ2: *Is QC effective on improving the query representation vis-a-vis source code?*

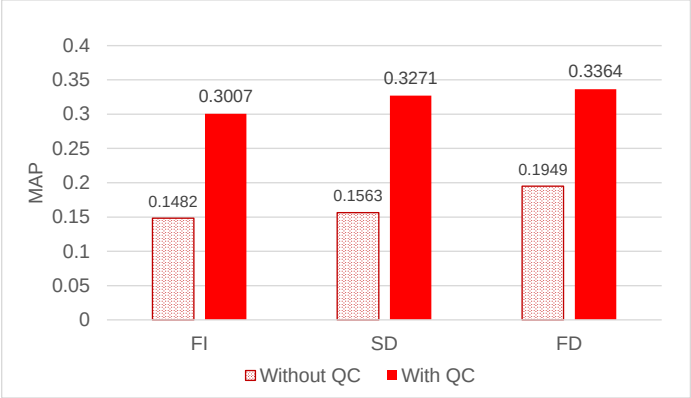
Comparing the results presented in Table 7.6, 7.7, 7.8 and 7.9, we conclude that QC indeed leads to a better query formulation for source code retrieval. For all of the dependency assumptions, we obtained significant improvements with QC in terms of the 6 evaluation metrics mentioned in the tables.

The main contribution of QC comes with the bug reports containing stack traces. This is because patches are included only in a few bug reports. Fig. 7.8 presents the retrieval accuracies of Terrier+ obtained specifically with the bug reports that contain stack traces. The figure shows that accuracy of the retrievals doubles on average with QC for both projects in term of MAP, reaching values above the 0.3 threshold.

Fig. 7.8 also demonstrates the effect of the MRF modeling with stack traces. Comparing the results obtained with the dependency models, we observe that FD and SD outperform FI consistently when QC is used in retrievals with the stack traces. The main reason for these results is that the order and the proximity of the terms in stack traces are extremely important in locating the relevant source files.

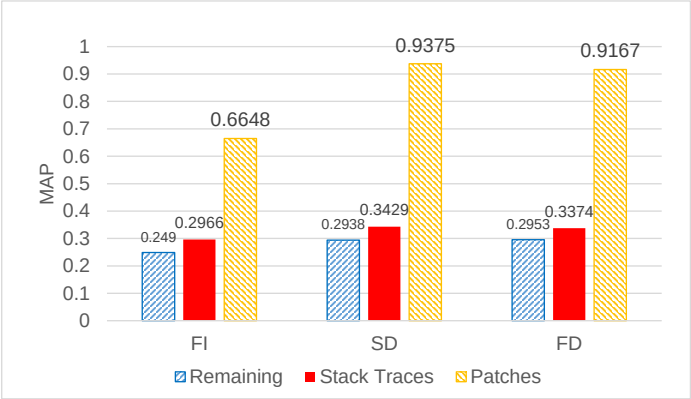


(a) Eclipse

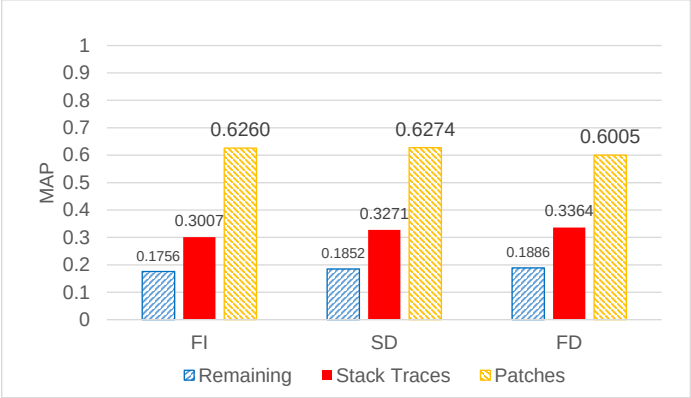


(b) AspectJ

Fig. 7.8. The Effect of Query Conditioning (QC) on BL with bug reports containing stack traces.



(a) Eclipse



(b) AspectJ

Fig. 7.9. The effect of including structural elements in bug reports on automatic BL accuracy. Patches lead to the highest retrieval scores while the bug reports with no structural elements perform the worst in terms of MAP.

As we also mentioned in Section 7.2, the likelihood of a file to be relevant to a query increases when it contains longer phrases from the stack trace with the same order and proximity relationships. Interestingly, when the queries are not processed with QC, the average retrieval accuracy with FD on the Eclipse project is slightly lower than FI. This is clearly due to the noise in the lengthy stack traces that contain many method signatures most of which are irrelevant to the bug. The QC framework effectively removes the irrelevant method signatures from the trace for a better query representation.

The Role of Stack Traces in Automatic Bug Localization

Studies showed that, in bug reports, developers look for stack traces, steps to reproduce the bug, and test cases since these are the most useful structural elements for comprehending the underlying cause of the bugs and fixing them [102,103]. Among these structural elements, stack traces are very important for the work we report in this paper. They are not only frequently included in bug reports but also a good source of discriminative source code identifiers for automatic BL. Fig. 7.9 presents the retrieval accuracies obtained with the bug reports containing different types of structural elements. In the figure, "remaining" denotes the bug reports that do not contain any stack traces or patches.

That sets us up to answer the research question RQ3 that was previously articulated in Section 7.1:

RQ3: *Does including stack traces and/or patches in the bug reports improve the accuracy of the BL?*

As demonstrated in Fig. 7.9, bug reports with patches lead to the highest accuracies as expected. After the patches, stack traces hold the second position in terms of their usefulness in locating the relevant source code. The retrieval results for the remaining bug reports that do not contain any stack traces or patches are the worst. Based on these results, we conclude that including stack traces and/or patches in the

bug reports does improve the BL accuracy. Note that the retrieval accuracies we obtained with the stack traces are consistently above the 0.3 threshold for the analyzed projects and the retrieval accuracies obtained with the patches are consistently above the 0.6 threshold in terms of MAP.

7.4.2 Comparison to Automatic Query Reformulation (QR)

In Chapter 6, we proposed an automatic Query Reformulation (QR) framework to improve the query representation for BL [19]. For experimental evaluation, we used the title of a bug report as an initial query which is reformulated via Pseudo Relevance Feedback based on the retrieval results obtained with the initial query. We showed that the proposed Spatial Code Proximity (SCP) based QR model outperforms the state-of-the-art QR models [19]. In order to compare the effectiveness of QR to MRF-based modeling, we pose the following question that was originally given in Section 7.1.

RQ4: *How does the MRF-based retrieval framework compare to the QR-based retrieval framework for BL?*

Comparing the retrieval accuracies presented in Table 7.10 and Table 7.11 to the retrieval accuracies of the QR models described in Chapter 6, we observe that MRF framework outperforms the SCP-based QR on average. For the Chrome project, while the differences between the average precisions obtained with the respective models are not statistically significant at $\alpha = 0.05$, the differences in terms of the presented recall metrics are. Additionally, H@10 values obtained with the MRF framework are considerably higher than the values obtained with the SCP-QR. For the Eclipse project, both SD and FD performs better than SCP-based QR in terms of the reported metrics. The differences are statistically significant at $\alpha = 0.05$.

Table 7.10 QR vs. MRF on Eclipse with the "title-only" queries.

Method	MAP	P@1	P@5	R@5	R@10	H@10
FD	**0.2564**	**0.2198**	**0.1110**	**0.3199**	**0.4070**	**2,100**
SD	0.2466	0.2116	0.1069	0.3083	0.3934	2,042
SCP-QR	0.2296	0.1906	0.1014	0.2853	0.3746	1,915

Table 7.11 QR vs. MRF on Chrome with the "title-only" queries.

Method	MAP	P@1	P@5	R@5	R@10	H@10
FD	**0.1951**	**0.1844**	**0.1061**	**0.2394**	**0.3159**	**178**
SD	0.1814	0.1760	0.1039	0.2288	0.3137	177
SCP-QR	0.1820	0.1788	0.0933	0.2021	0.2775	151

7.4.3 Comparison to Bug Localization Techniques that use Prior Development Efforts

In [39], Zhou et al. proposed BugLocator, a retrieval tool that uses the textual similarities between a given bug report and the prior bug reports to enhance the BL accuracy. The main motivation behind BugLocator is that the same files tend to get fixed for similar bug reports during the life-cycle of a software project. In Chapter 5, we also described another approach that leverages the past development efforts for BL. Our experimental study on iBugs dataset showed that the TFIDF[8] retrieval model incorporating Defect History based Prior with decay (DHbPd) performs the best among other composite models. This brief review of this class of approaches to bug localization takes us to the last of the research questions stated in 7.1:

RQ5: *How does the MRF-based retrieval framework compare to the BL frameworks that leverage the past development efforts?*

[8]TFIDF stands for Term Frequency Inverse Document Frequency.

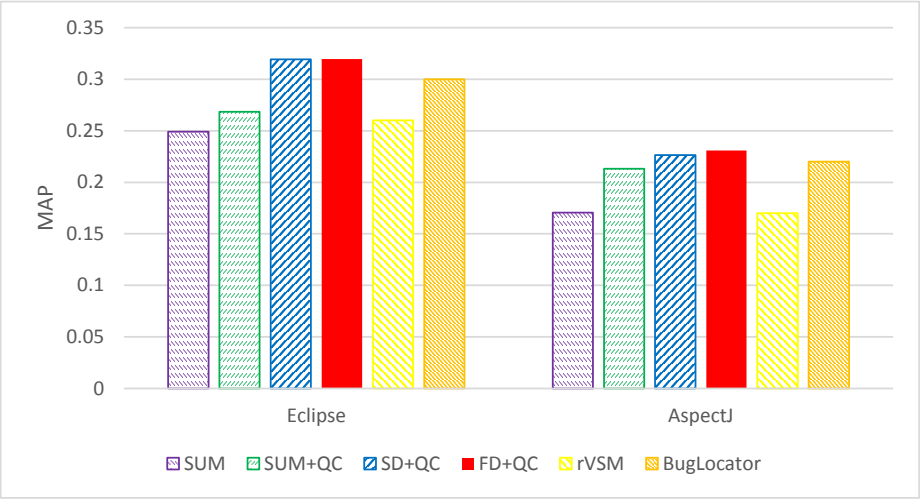


Fig. 7.10. Comparison of the retrieval models for Bug Localization

The accuracy of BugLocator is also evaluated on Eclipse v3.1 and iBugs datasets. The evaluations on the Eclipse project are performed using 3,075 bug reports filed for version 3.1 while our experiments with Terrier+ were performed using 4,035 bug reports filed for the same version. In order to compare the performance of Terrier+ to that of BugLocator, we repeated the experiments using only the bug reports with which the BugLocator was evaluated. Fig. 7.10 plots the accuracy of the proposed framework along with the BugLocator in terms of MAP. In the figure, we also included the accuracies obtained with the revised Vector Space Model (rVSM) [39] which is reported to perform better than the classical Vector Space Model (VSM). As can be seen on the figure, FD+QC and SD+QC outperform BugLocator with MAP values above 0.32 threshold for the Eclipse project. In comparison, BugLocator performs better than the FI modeling i.e. SUM and SUM+QC with a MAP value of 0.3. The comparison results are similar for the AspectJ project. In addition to these results, in Chapter 5, we showed that TFIDF+DHbPd reaches a MAP value of 0.2258 on the AspectJ project.

Based on these results, we conclude that *Terrier+ performs better than these BL techniques without leveraging the past development efforts.*

8. A RETRIEVAL ENGINE FOR BUG LOCALIZATION: TERRIER+

In order to address the issues that arise in code search for Bug Localization (BL), as mentioned earlier, we present a new source code search engine developed by extending Terrier[1], a flexible open source IR platform written in Java [106]. This new platform, named *Terrier+*, extends the functionalities of Terrier for the following components:

- *Indexing:* In the indexing phase, the source code artifacts are first parsed and tokenized. Then they are stored for fast access during retrieval time. The queries posed to the retrieval engine are also subject to the same parsing and tokenization procedures as the source code artifacts.
- *Document Score Modifiers:* This component computes additional scores for each artifact in the code base after the first round of term matching takes place and it is responsible for the incorporation of version histories and leveraging Spatial Code Proximity (SCP) in the retrieval process.
- *Querying:* This component has been extended to implement the proposed SCP-based Query Expansion/Reformulation framework with Pseudo Relevance Feedback. This component is also responsible for Query Conditioning (QC).

8.1 Indexing

A vital component of any scalable retrieval framework that is expected work on very large databases is indexing. This component enables fast access to the features associated with each document and term in the corpus of a document collection via

[1]http://terrier.org

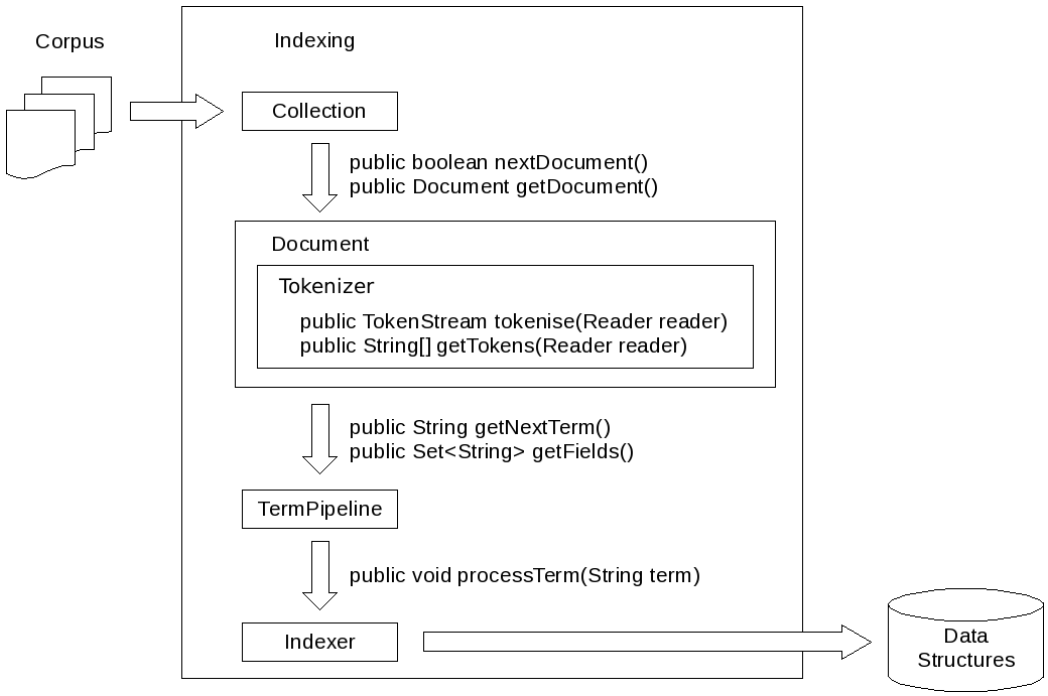


Fig. 8.1. Indexing the corpus of a document collection with Terrier.

compression and search algorithms. Before storing the documents and the terms in an index, each document in the corpus goes through three steps for term extraction: (1) Tokenization (2) Stop-word removal and (3) Stemming. In comparison to Natural Language (NL) document retrieval, these three stages demonstrate significant differences when the code base of a software library is the target corpus for indexing. Firstly, while the terms in NL documents are simply separated with spacing, the textual content of software libraries are created by using abbreviations and/or concatenation of several terms via punctuation characters or camel casing. Recognizing the constituent tokens in these composite phrases is crucial for accurate document representation for BL.

After the terms are extracted via tokenization, the next step in the indexing process is the stop-word removal [16, 19]. In this step, the nondiscriminatory terms are dropped from the index as these terms add noise into the retrievals. Different from NL corpora, source code collections contain Programming Language (PL) specific

terms, definition and declaration of software elements, comments and so on. Due to this composite nature of source code, the set of stop-words to be used in indexing should contain both PL specific terms and NL terms.

The final step in indexing is stemming [90]. The stemming method which is used to reduce the tokens into their common root forms also has a significant impact on BL as it leads to a smaller vocabulary to index and a better document representation that increase the chance of term matching [107].

Fig. 8.1 shows the indexing framework and Fig. 8.2 shows the retrieval components of Terrier.

8.2 Document Score Modifiers (DSM)

Document Score Modifiers (DSM) assign secondary scores to the source code elements after the initial term matching takes place. In order to incorporate the version histories of the software artifacts in the retrieval process, we generate a separate index that stores the change history of each document in the code base. After scoring the documents based on the terms present in the query and in the software artifacts, a DSM is created and it uses this index to compute the *prior bug/modification likelihoods* of each artifact and updates the document rankings accordingly. This component is also responsible for taking into account the term proximity and order in the scoring phase. In order to exploit SCP for an improved retrieval accuracy, DSM employs Markov Random Fields (MRF) in the matching phase.

8.3 Querying

DSM phase completes the retrieval process with respect to the original query. If the query is to be reformulated then the initial set of retrieval results obtained via the original query are fed into the *Post-processing* unit. Automatic Query Reformulation (QR) is performed by this unit where the original query and the initial (first-pass) retrieval results are analyzed for QR using the notion of *Pseudo Relevance Feedback*

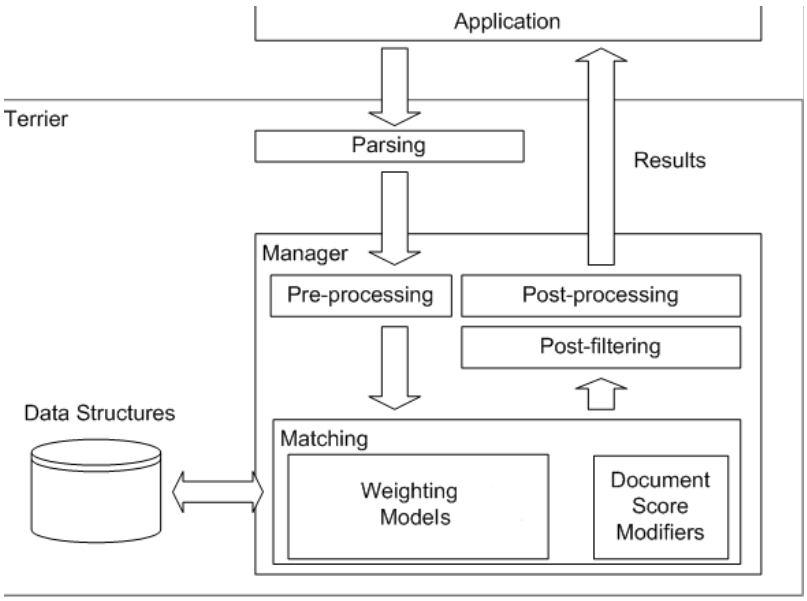


Fig. 8.2. Retrieval components of Terrier.

[19]. During the reformulation, the original query is enriched with certain additional informative terms drawn from the first-pass retrieval results. Our proposed Query Reformulation framework uses the positional proximities of the terms in the first-pass retrieval results vis-a-vis the query terms for QR [19]. After the initial query is reformulated, the new query is used to perform a second run of retrieval to obtain the final set of results.

Fig. 8.3 presents the overall data flow in the proposed retrieval framework with Terrier+.

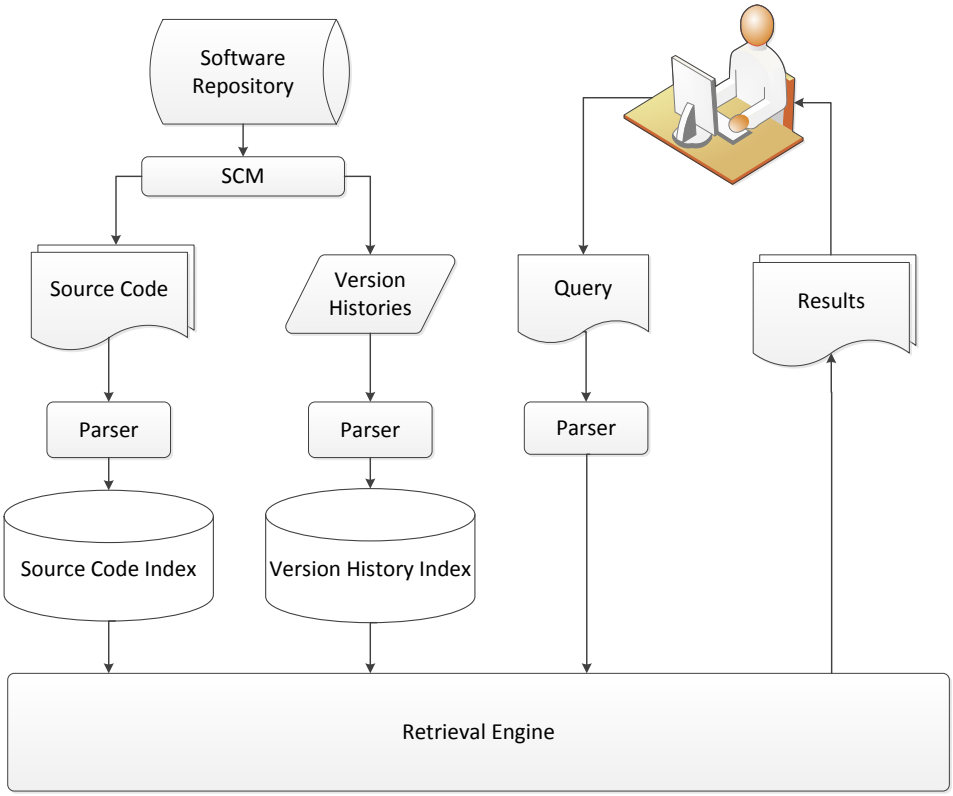


Fig. 8.3. The overall retrieval framework with Terrier+ for BL.

9. CONCLUSIONS

In this dissertation, we advance the state-of-the-art in Information Retrieval (IR) based Bug Localization (BL). To achieve accurate BL in large, ever-changing software systems, we presented methods to address the issues that arise when IR algorithms are applied in source code retrieval. On the one hand, the proposed IR framework exploits the prior development efforts to support BL. On the other hand, our retrieval mechanism leverages the spatial features in the textual content of the software either through automatic Query Reformulation (QR) or more directly with a well-principled Markov Random Field (MRF) based approach. The functionalities in this BL framework, implemented as an IR tool named Terrier+, are utilized via a GUI. Based on the nature of the information need, the developer can easily turn on/off these functionalities during BL.

In order to exploit the prior software development and maintenance history, our retrieval framework offers a theoretically well-principled approach for incorporating version histories of software files for BL. Our approach uses the information stored in software versioning tools regarding the frequency with which a file is associated with defects and its modifications in order to construct estimates for the prior probability of any given file to be the source of a bug. Incorporating these priors in an IR based retrieval framework, as we have done, significantly improves the retrieval performance for bug localization. What is even more remarkable is that when we associate a time decay factor with the priors, the improvement in BL goes up even more dramatically. For the retrieval itself, our work used two different algorithms, one based on Bayesian reasoning and other on the Divergence from Randomness principle.

The automatic QR framework we propose for BL becomes essential in assisting inexperienced users for constructing high quality queries. This is mainly because designing a query that can accurately retrieve the source files from a software library

is particularly challenging for a novice user on account of the fact that developers use made-up words as identifiers for variables, method names, class names, etc. A user not already familiar with the library would have a difficult time conjuring up these names. Our QR approach uses what is likely to be a weak initial query as a jumping off point for a stronger query.

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

With regard to exploiting the spatial features in software systems for an improved BL accuracy, the proposed MRF framework focuses especially on long queries which may be in the form of a bug report containing various structural elements such as stack traces or patches. We showed that these structural elements in bug reports contain vital proximity and order attributes that is not well represented via-a-vis the source code with the widely used bag-of-words assumption. Our experimental validation involving large open-source software projects and over 4 thousand bugs has established that the retrieval performance improves significantly when both the proximity and ordering relationships between the terms are taken into account. Our experimental evaluation also demonstrated that in conjunction with the MRF model, the proposed Query Conditioning (QC) approach is essential for exploiting the different types of structural elements in bug reports.

Comparing the retrieval accuracies obtained with Terrier+ to those obtained with the other state-of-the-art IR tools, we conclude that, on average, Terrier+ reaches higher bug localization accuracies.

LIST OF REFERENCES

LIST OF REFERENCES

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

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

[3] D. Liu, A. Marcus, D. Poshyvanyk, and V. Rajlich, "Feature location via information retrieval based filtering of a single scenario execution trace," in *Proceedings of the twenty-second IEEE/ACM international conference on Automated software engineering*, pp. 234–243, ACM, 2007.

[4] A. D. Lucia, F. Fasano, R. Oliveto, and G. Tortora, "Recovering traceability links in software artifact management systems using information retrieval methods," *ACM Transactions on Software Engineering and Methodology (TOSEM)*, vol. 16, no. 4, p. 13, 2007.

[5] D. Poshyvanyk, A. Marcus, R. Ferenc, and T. Gyimóthy, "Using information retrieval based coupling measures for impact analysis," *Empirical Software Engineering*, vol. 14, no. 1, pp. 5–32, 2009.

[6] M. Gethers, B. Dit, H. Kagdi, and D. Poshyvanyk, "Integrated impact analysis for managing software changes," in *34th International Conference on Software Engineering (ICSE'12)*, pp. 430–440, IEEE, 2012.

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

[8] I. Vessey, "Expertise in debugging computer programs: A process analysis," *International Journal of Man-Machine Studies*, vol. 23, no. 5, pp. 459–494, 1985.

[9] V. Dallmeier, C. Lindig, and A. Zeller, "Lightweight bug localization with ample," in *Proceedings of the 6th international symposium on Automated analysis-driven debugging*, pp. 99–104, ACM, 2005.

[10] D. Hovemeyer and W. Pugh, "Finding bugs is easy," *ACM SIGPLAN Notices*, vol. 39, no. 12, pp. 92–106, 2004.

[11] C. Zhai, "Statistical language models for information retrieval," *Synthesis Lectures on Human Language Technologies*, vol. 1, no. 1, pp. 1–141, 2008.

[12] J. M. Ponte and W. B. Croft, "A language modeling approach to information retrieval," in *Proceedings of the 21st annual international ACM SIGIR conference on Research and development in information retrieval*, pp. 275–281, ACM, 1998.

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

[14] S. Robertson and K. Spärck Jones, "Simple, proven approaches to text retrieval," Tech. Rep. UCAM-CL-TR-356, University of Cambridge, Computer Laboratory, Dec. 1994.

[15] D. Metzler and W. Croft, "A markov random field model for term dependencies," in *Proceedings of the 28th annual international ACM SIGIR conference on Research and development in information retrieval*, pp. 472–479, ACM, 2005.

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

[17] S. Haiduc, "Automatically detecting the quality of the query and its implications in ir-based concept location," in *Proceedings of the 2011 26th IEEE/ACM International Conference on Automated Software Engineering*, pp. 637–640, IEEE Computer Society, 2011.

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

[19] B. Sisman and A. C. Kak, "Assisting code search with automatic query reformulation for bug localization," in *Proceedings of the 10th Working Conference on Mining Software Repositories*, MSR '13, (Piscataway, NJ, USA), pp. 309–318, IEEE Press, 2013.

[20] N. Bettenburg, R. Premraj, T. Zimmermann, and S. Kim, "Extracting structural information from bug reports," in *Proceedings of the 2008 international working conference on Mining software repositories*, pp. 27–30, ACM, 2008.

[21] B. Ashok, J. Joy, H. Liang, S. Rajamani, G. Srinivasa, and V. Vangala, "Debugadvisor: a recommender system for debugging," in *Proceedings of the the 7th joint meeting of the European software engineering conference and the ACM SIGSOFT symposium on The foundations of software engineering*, pp. 373–382, ACM, 2009.

[22] N. Wilde and M. C. Scully, "Software reconnaissance: Mapping program features to code," *Journal of Software Maintenance: Research and Practice*, vol. 7, no. 1, pp. 49–62, 1995.

[23] M. D. Ernst, J. Cockrell, W. G. Griswold, and D. Notkin, "Dynamically discovering likely program invariants to support program evolution," *IEEE Transactions on Software Engineering*, vol. 27, no. 2, pp. 99–123, 2001.

[24] S. Hangal and M. S. Lam, "Tracking down software bugs using automatic anomaly detection," in *Proceedings of the 24th international conference on Software engineering*, pp. 291–301, ACM, 2002.

[25] T. Eisenbarth, R. Koschke, and D. Simon, "Locating features in source code," *IEEE Transactions on Software Engineering*, vol. 29, no. 3, pp. 210–224, 2003.

[26] M. Renieres and S. P. Reiss, "Fault localization with nearest neighbor queries," in *Proceedings. 18th IEEE International Conference on Automated Software Engineering, 2003.*, pp. 30–39, IEEE, 2003.

[27] C. Liu, X. Yan, L. Fei, J. Han, and S. P. Midkiff, "Sober: statistical model-based bug localization," *ACM SIGSOFT Software Engineering Notes*, vol. 30, no. 5, pp. 286–295, 2005.

[28] C. Liu, L. Fei, X. Yan, J. Han, and S. P. Midkiff, "Statistical debugging: A hypothesis testing-based approach," *IEEE Transactions on Software Engineering*, vol. 32, no. 10, pp. 831–848, 2006.

[29] B. Liblit, M. Naik, A. X. Zheng, A. Aiken, and M. I. Jordan, "Scalable statistical bug isolation," *ACM SIGPLAN Notices*, vol. 40, no. 6, pp. 15–26, 2005.

[30] W. E. Wong, T. Wei, Y. Qi, and L. Zhao, "A crosstab-based statistical method for effective fault localization," in *1st International Conference on Software Testing, Verification, and Validation, 2008*, pp. 42–51, IEEE, 2008.

[31] G. K. Baah, A. Podgurski, and M. J. Harrold, "The probabilistic program dependence graph and its application to fault diagnosis," *IEEE Transactions on Software Engineering*, vol. 36, no. 4, pp. 528–545, 2010.

[32] J. A. Jones, M. J. Harrold, and J. Stasko, "Visualization of test information to assist fault localization," in *Proceedings of the 24th International Conference on Software Engineering*, ICSE '02, (New York, NY, USA), pp. 467–477, ACM, 2002.

[33] M. P. Robillard, "Automatic generation of suggestions for program investigation," in *ACM SIGSOFT Software Engineering Notes*, vol. 30, pp. 11–20, ACM, 2005.

[34] M. P. Robillard, "Topology analysis of software dependencies," *ACM Transactions on Software Engineering and Methodology (TOSEM)*, vol. 17, no. 4, p. 18, 2008.

[35] E. Hill, L. Pollock, and K. Vijay-Shanker, "Automatically capturing source code context of nl-queries for software maintenance and reuse," in *31st International Conference on Software Engineering, 2009. ICSE 2009.*, pp. 232–242, 2009.

[36] E. Hill, L. Pollock, and K. Vijay-Shanker, "Improving source code search with natural language phrasal representations of method signatures," in *Proceedings of the 2011 26th IEEE/ACM International Conference on Automated Software Engineering*, pp. 524–527, IEEE Computer Society, 2011.

[37] S. Rao and A. Kak, "Retrieval from software libraries for bug localization: a comparative study of generic and composite text models," in *Proceedings of the 8th working conference on Mining software repositories*, pp. 43–52, 2011.

[38] A. T. Nguyen, T. T. Nguyen, J. Al-Kofahi, H. V. Nguyen, and T. Nguyen, "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 (ASE'11)*, pp. 263 –272, nov. 2011.

[39] J. Zhou, H. Zhang, and D. Lo, "Where should the bugs be fixed? more accurate information retrieval-based bug localization based on bug reports," in *34th International Conference on Software Engineering (ICSE), 2012*, pp. 14–24, IEEE, 2012.

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

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

[42] S. Haiduc, G. Bavota, A. Marcus, R. Oliveto, A. De Lucia, and T. Menzies, "Automatic query reformulations for text retrieval in software engineering," in *Proceedings of the 2013 International Conference on Software Engineering*, ICSE '13, (Piscataway, NJ, USA), pp. 842–851, IEEE Press, 2013.

[43] M. D. Ernst, "Static and dynamic analysis: Synergy and duality," in *WODA 2003: ICSE Workshop on Dynamic Analysis*, pp. 24–27, Citeseer, 2003.

[44] B. Dufour, B. G. Ryder, and G. Sevitsky, "Blended analysis for performance understanding of framework-based applications," in *Proceedings of the 2007 international symposium on Software testing and analysis*, pp. 118–128, ACM, 2007.

[45] D. Poshyvanyk, Y.-G. Guéhéneuc, A. Marcus, G. Antoniol, and V. Rajlich, "Feature location using probabilistic ranking of methods based on execution scenarios and information retrieval," *IEEE Transactions on Software Engineering*, vol. 33, no. 6, pp. 420–432, 2007.

[46] B. Dit, M. Revelle, and D. Poshyvanyk, "Integrating information retrieval, execution and link analysis algorithms to improve feature location in software," *Empirical Software Engineering*, vol. 18, no. 2, pp. 277–309, 2013.

[47] T. Eisenbarth, R. Koschke, and D. Simon, "Aiding program comprehension by static and dynamic feature analysis," in *Proceedings of the IEEE International Conference on Software Maintenance, ICSM 2001*, pp. 602–611, IEEE, 2001.

[48] D. Poshyvanyk, Y.-G. Guéhéneuc, A. Marcus, G. Antoniol, and V. Rajlich, "Combining probabilistic ranking and latent semantic indexing for feature identification," in *14th IEEE International Conference on Program Comprehension, 2006.*, pp. 137–148, IEEE, 2006.

[49] D. Poshyvanyk and A. Marcus, "Combining formal concept analysis with information retrieval for concept location in source code," in *15th IEEE International Conference on Program Comprehension, 2007. ICPC'07.*, pp. 37–48, IEEE, 2007.

[50] D. Poshyvanyk, M. Gethers, and A. Marcus, "Concept location using formal concept analysis and information retrieval," *ACM Transactions on Software Engineering and Methodology (TOSEM)*, vol. 21, no. 4, p. 23, 2012.

[51] A. Lashkari, F. Mahdavi, and V. Ghomi, "A Boolean Model in Information Retrieval for Search Engines," in *International Conference on Information Management and Engineering, (ICIME'09)*, pp. 385–389, IEEE, 2009.

[52] G. Salton and M. Lesk, "The SMART automatic document retrieval systemsan illustration," *Communications of the ACM*, vol. 8, no. 6, pp. 391–398, 1965.

[53] S. Deerwester, S. Dumais, G. Furnas, T. Landauer, and R. Harshman, "Indexing by latent semantic analysis," *Journal of the American society for information science*, vol. 41, no. 6, pp. 391–407, 1990.

[54] G. Amati and C. Van Rijsbergen, "Probabilistic models of information retrieval based on measuring the divergence from randomness," *ACM Transactions on Information Systems (TOIS)*, vol. 20, no. 4, pp. 357–389, 2002.

[55] C. Zhai and J. Lafferty, "A study of smoothing methods for language models applied to information retrieval," *ACM Transactions on Information Systems (TOIS)*, vol. 22, no. 2, pp. 179–214, 2004.

[56] A. Barua, S. W. Thomas, and A. E. Hassan, "What are developers talking about? an analysis of topics and trends in stack overflow," *Empirical Software Engineering*, pp. 1–36, 2012.

[57] S. W. Thomas, B. Adams, A. E. Hassan, and D. Blostein, "Studying software evolution using topic models," *Science of Computer Programming*, 2012.

[58] S. W. Thomas, H. Hemmati, A. E. Hassan, and D. Blostein, "Static test case prioritization using topic models," *Empirical Software Engineering*, pp. 1–31, 2012.

[59] K. Nigam, A. McCallum, S. Thrun, and T. Mitchell, "Text classification from labeled and unlabeled documents using EM," *Machine learning*, vol. 39, no. 2, pp. 103–134, 2000.

[60] T. Hofmann, "Probabilistic latent semantic indexing," in *Proceedings of the 22nd annual international ACM SIGIR conference on Research and development in information retrieval*, pp. 50–57, ACM, 1999.

[61] D. M. Blei, A. Y. Ng, and M. I. Jordan, "Latent Dirichlet Allocation," *the Journal of machine Learning research*, vol. 3, pp. 993–1022, 2003.

[62] J. Ponte, "Is information retrieval anything more than smoothing," in *Proceedings of the Workshop on Language Modeling and Information Retrieval*, pp. 37–41, 2001.

[63] L. Yao, D. Mimno, and A. McCallum, "Efficient methods for topic model inference on streaming document collections," in *Proceedings of the 15th ACM SIGKDD international conference on Knowledge discovery and data mining*, pp. 937–946, ACM, 2009.

[64] G. Heinrich, "Parmater Estimation for Text Analysis," tech. rep., 2008.

[65] E. Voorhees, "The cluster hypothesis revisited," in *Proceedings of the 8th annual international ACM SIGIR conference on Research and development in information retrieval*, pp. 188–196, ACM, 1985.

[66] M. Spitters and W. Kraaij, "TNO at TDT2001: Language model-based topic detection," 2001.

[67] L. X and W. B. Croft, "Cluster-Based Retrieval Using Language Models," in *ACM SIGIR Conference on Research and Development in Information Retrieval*, ACM, 2004.

[68] X. Wei and W. B. Croft, "Lda-based document models for ad-hoc retrieval," in *Proceedings of the 29th annual international ACM SIGIR conference on Research and development in information retrieval*, ACM, 2006.

[69] J. Lafferty and C. Zhai, "Document language models, query models, and risk minimization for information retrieval," in *Proceedings of the 24th annual international ACM SIGIR conference on Research and development in information retrieval*, pp. 111–119, ACM, 2001.

[70] H. Kagdi, M. L. Collard, and J. I. Maletic, "A survey and taxonomy of approaches for mining software repositories in the context of software evolution," *Journal of Software Maintenance and Evolution: Research and Practice*, vol. 19, no. 2, pp. 77–131, 2007.

[71] H. Kagdi, M. Gethers, D. Poshyvanyk, and M. Collard, "Blending conceptual and evolutionary couplings to support change impact analysis in source code," in *Proceedings of the 17th Working Conference on Reverse Engineering (WCRE'10)*, pp. 119 –128, oct. 2010.

[72] B. Livshits and T. Zimmermann, "Dynamine: finding common error patterns by mining software revision histories," in *ACM SIGSOFT Software Engineering Notes*, vol. 30, pp. 296–305, ACM, 2005.

[73] G. Canfora and L. Cerulo, "Impact analysis by mining software and change request repositories," in *11th IEEE International Symposium in Software Metrics, 2005.*, pp. 9 pp. –29, sept. 2005.

[74] R. Moser, W. Pedrycz, and G. Succi, "A comparative analysis of the efficiency of change metrics and static code attributes for defect prediction," in *30th International Conference on Software Engineering (ICSE'08)*, pp. 181–190, IEEE, 2008.

[75] M. D'Ambros, M. Lanza, and R. Robbes, "An extensive comparison of bug prediction approaches," in *7th IEEE Working Conference on Mining Software Repositories (MSR'10)*, pp. 31–41, IEEE, 2010.

[76] A. Hassan, "Predicting faults using the complexity of code changes," in *Proceedings of the 31st International Conference on Software Engineering (ICSE'09)*, pp. 78–88, IEEE Computer Society, 2009.

[77] N. Nagappan, A. Zeller, T. Zimmermann, K. Herzig, and B. Murphy, "Change bursts as defect predictors," in *21st International Symposium on Software Reliability Engineering (ISSRE'10)*, pp. 309–318, IEEE, 2010.

[78] A. Ying, G. Murphy, R. Ng, and M. Chu-Carroll, "Predicting source code changes by mining change history," *IEEE Transactions on Software Engineering*, pp. 574–586, 2004.

[79] T. Zimmermann, P. Weissgerber, S. Diehl, and A. Zeller, "Mining version histories to guide software changes," *IEEE Transactions on Software Engineering*, pp. 429–445, 2005.

[80] A. Hassan and R. Holt, "The top ten list: Dynamic fault prediction," in *Proceedings of the 21st IEEE International Conference on Software Maintenance (ICSM'05)*, pp. 263–272, IEEE, 2005.

[81] A. Bernstein, J. Ekanayake, and M. Pinzger, "Improving defect prediction using temporal features and non linear models," in *9th international workshop on Principles of software evolution: in conjunction with the 6th ESEC/FSE joint meeting*, pp. 11–18, ACM, 2007.

[82] A. Hindle, D. German, and R. Holt, "What do large commits tell us?: a taxonomical study of large commits," in *Proceedings of the 2008 international working conference on Mining software repositories (MSR'08)*, pp. 99–108, ACM, 2008.

[83] A. Mockus and L. Votta, "Identifying reasons for software changes using historic databases," in *Proceedings of International Conference on Software Maintenance (ICSM'00).*, pp. 120 –130, 2000.

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

[85] W. Kraaij, T. Westerveld, and D. Hiemstra, "The importance of prior probabilities for entry page search," in *Proceedings of the 25th annual international ACM SIGIR conference on Research and development in information retrieval*, pp. 27–34, ACM, 2002.

[86] J. Peng, C. Macdonald, B. He, and I. Ounis, "Combination of document priors in web information retrieval," in *Large Scale Semantic Access to Content (Text, Image, Video, and Sound)*, pp. 596–611, LE CENTRE DE HAUTES ETUDES INTERNATIONALES D'INFORMATIQUE DOCUMENTAIRE, 2007.

[87] C. Macdonald, B. He, V. Plachouras, and I. Ounis, "University of glasgow at trec 2005: Experiments in terabyte and enterprise tracks with terrier," in *Proceedings of TREC 2005*, 2005.

[88] V. Dallmeier and T. Zimmermann, "Extraction of bug localization benchmarks from history," in *Proceedings of the twenty-second IEEE/ACM international conference on Automated software engineering*, pp. 433–436, ACM, 2007.

[89] T. Zimmermann and P. Weißgerber, "Preprocessing cvs data for fine-grained analysis," in *Proceedings of the 1st International Workshop on Mining Software Repositories (MSR 2004)*, pp. 2–6, 2004.

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

[91] F. Deissenboeck and M. Pizka, "Concise and consistent naming," *Software Quality Control*, vol. 14, pp. 261–282, Sept. 2006.

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

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

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

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

[96] D. Kelly and J. Teevan, "Implicit feedback for inferring user preference: a bibliography," *SIGIR Forum*, vol. 37, pp. 18–28, Sept. 2003.

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

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

[99] J. Miao, J. X. Huang, and Z. Ye, "Proximity-based rocchio's model for pseudo relevance," in *Proceedings of the 35th international ACM SIGIR conference on Research and development in information retrieval*, SIGIR '12, (New York, NY, USA), pp. 535–544, ACM, 2012.

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

[101] R. Wu, H. Zhang, S. Kim, and S. Cheung, "Relink: recovering links between bugs and changes," in *SIGSOFT FSE*, pp. 15–25, 2011.

[102] N. Bettenburg, S. Just, A. Schröter, C. Weiss, R. Premraj, and T. Zimmermann, "What makes a good bug report?," in *Proceedings of the 16th ACM SIGSOFT International Symposium on Foundations of software engineering*, pp. 308–318, ACM, 2008.

[103] A. Schroter, N. Bettenburg, and R. Premraj, "Do stack traces help developers fix bugs?," in *Mining Software Repositories (MSR), 2010 7th IEEE Working Conference on*, pp. 118–121, IEEE, 2010.

[104] D. Kollar and N. Friedman, *Probabilistic graphical models: principles and techniques*. The MIT Press, 2009.

[105] J. Peng, C. Macdonald, B. He, V. Plachouras, and I. Ounis, "Incorporating term dependency in the DFR framework," in *Proceedings of the 30th annual international ACM SIGIR conference on Research and development in information retrieval*, pp. 843–844, ACM, 2007.

[106] I. Ounis, G. Amati, V. Plachouras, B. He, C. Macdonald, and C. Lioma, "Terrier: A High Performance and Scalable Information Retrieval Platform," in *Proceedings of ACM SIGIR'06 Workshop on Open Source Information Retrieval (OSIR 2006)*, 2006.

[107] E. Hill, S. Rao, and A. Kak, "On the use of stemming for concern location and bug localization in Java," in *2012 IEEE 12th International Working Conference on Source Code Analysis and Manipulation (SCAM)*, pp. 184–193, IEEE, 2012.

VITA

VITA

Bunyamin Sisman received his BS degree in Computer Engineering from Istanbul Technical University, Istanbul, Turkey in 2006. He received his MS degree in Computer Engineering from Purdue University in 2009. Since then he has been pursuing his PhD studies in Computer Engineering at Purdue University. His research interests include Source Code Retrieval from Large Software Repositories and Mining Software Repositories.