STRUCTURAL INFORMATION BASED TERM WEIGHTING IN TEXT RETRIEVAL FOR FEATURE LOCATION

by

RICHARD B. BASSETT

NICHOLAS A. KRAFT, COMMITTEE CHAIR
LETHA H. ETZKORN
JEFF GRAY
RANDY K. SMITH

A THESIS

Submitted in partial fulfillment of the requirements
for the degree of Master of Science
in the Department of Computer Science
in the Graduate School of
The University of Alabama

TUSCALOOSA, ALABAMA

2013

Copyright Richard B. Bassett 2013
ALL RIGHTS RESERVED

ABSTRACT

Feature location is a program comprehension activity in which a developer identifies source code entities that implement a feature of interest. Recent feature location techniques apply text retrieval techniques to corpora built from text embedded in source code. These techniques are highly configurable, but many of the available parameters remain unexplored in the software engineering context. For example, while the natural language processing community has developed several term weighting schemes meant to highlight the importance of certain terms in a particular document, the software engineering community has thus far not developed new term weighting schemes for use with source code. Thus, we propose a new term weighting scheme that is based on the structural information in source code. We then report the results of an empirical study in which we evaluated the performance effects of the proposed term weighting scheme on a latent Dirichlet allocation (LDA) based feature location technique (FLT). In all, we studied over 400 bugs and features from five open source Java systems. Our key finding is that the accuracy of the LDA-based FLT improves when a structural term weighting scheme is used rather than a uniform term weighting scheme.

ACKNOWLEDGMENTS

I would like to extend my sincere gratitude to my advisor, Dr. Nicholas A. Kraft for his continued guidance and assistance throughout this research. I would also like to thank the members of my thesis committee, Dr. Jeffrey C. Carver, Dr. Jeff Gray, Dr. Letha H. Etzkorn, and Dr. Randy K. Smith, for their advice and their encouragement. I am also grateful to the graduate students and faculty in the Software Engineering Research Group for being willing to share their experience and advice. Finally, I would like to thank Catie, for her love and support.

CONTENTS

ABSTRACT ii

ACKNOWLEDGMENTS iii

LIST OF TABLES vi

LIST OF FIGURES vii

1 INTRODUCTION 1

1.1 Overview 1

1.2 Structural Term Weighting 3

1.3 Research Question 4

1.4 Organization 4

2 BACKGROUND & RELATED WORK 5

2.1 Source Code Indexing and Retrieval Process 5

2.1.1 Terminology 5

2.1.2 Indexing 6

2.1.3 Model Generation 8

2.1.4 Retrieval 8

2.2 Latent Dirichlet Allocation 9

2.3 Feature Location 10

3 EVALUATION OF STRUCTURAL TERM WEIGHTING 14

3.1 Definition and Context 14

3.1.1 Weighting configurations 14
3.1.2 Subject systems 15
3.1.3 Benchmarks and gold sets 15
3.1.4 Effectiveness measure 17
3.1.5 Methodology 17
3.2 Research Question and Hypotheses 18
3.3 Data Collection & Analysis 19
3.4 Results 21
3.4.1 Descriptive 21
3.4.2 Mean reciprocal rank 24
3.4.3 Statistical analysis 24
3.4.4 Qualitative analysis 26
3.5 Discussion 28
3.6 Threats to Validity 30
4 CONCLUSION & FUTURE WORK 32
4.1 Summary of Findings 32
4.2 Future Work 32
REFERENCES 34

LIST OF TABLES

3.1 Subject Systems 16

3.2 Numbers of methods in the gold sets. 16

3.3 Mean Reciprocal Rank for each Configuration and System 24

3.4 Friedman Test Results 25

3.5 Query document for ArgoUML feature 3911 26

3.6 Method documents for ArgoUML `ActionRemoveArgument` class constructor . 27

3.7 Query document for jEdit feature 1931333 28

3.8 Method documents for jEdit method `CompleteWord.keyTyped(keyEvent)` 28

LIST OF FIGURES

2.1 Source Code Indexing and Retrieval Process 5

3.1 Boxplots of Effectiveness Measures 22

3.1 Mean Reciprocal Rank for each Configuration over all Queries 25

3.2 The source code for ArgoUML `ActionRemoveArgument` class constructor . . 27

3.3 The source code for the jEdit method `CompleteWord.keyTyped(keyEvent)` 29

CHAPTER 1

INTRODUCTION

During the evolution of a software system, developers change the source code to add new features, enhance existing features, and remove defective features (bugs). When developers are tasked with changing the source code of a large or unfamiliar system, they must spend considerable time and effort on program comprehension activities to gain the knowledge needed to make correct and complete changes. Thus, techniques and tools that can reduce the effort required for program comprehension are are key to minimizing software costs.

1.1 Overview

Feature location is a program comprehension activity in which a developer identifies the source code entities that implement a feature. When attempting to locate a defective feature (bug), the activity is sometimes referred to as *bug localization* [Lukins, Kraft, and Etzkorn, 2008, 2010]. Because feature location is key to software evolution, as well as difficult and expensive to perform manually, the software evolution research community has devoted much effort to developing automated feature location techniques (FLTs). Recently, the focus of much of this effort is on the development of FLTs based on text retrieval (TR) techniques such as the vector space model (VSM) [Salton, 1989], latent semantic indexing (LSI) [Deerwester, Dumais, Furnas, Landauer, and Harshman, 1990], and latent Dirichlet allocation (LDA) [Blei, Ng, and Jordan, 2003].

The subject of this research is an LDA-based FLT. LDA is a probabilistic topic model. Given a document corpus and the number of topics, LDA models the corpus by estimating a topic

distribution for each document. Because LDA is generative, it also supports inference of the topic distribution for a new document. When applied to TR, the inital corpus consists of the documents to be searched, and the new documents are the queries for the search. For each query, the output is a ranked list of documents, where the top-ranked document is the one whose topic distribution is most similar to the topic distribution of the query. Thus, in the context of the LDA based FLT, the initial corpus consists of documents that correspond to source code entities (typically methods), and the new documents are descriptions of the features to be located. For each feature, the FLT produces a ranked list of source code entities, where the top-ranked source code entity is the one whose topic distribution is most similar to the topic distribution of the feature.

TR-based FLTs are highly configurable, both because of technique-specific parameters and because of the many decisions that must be made when indexing source code to produce a corpus. Examples of technique-specific parameters include the LSI parameter k, which indicates the (reduced) dimensionality desired, the LDA hyperparameters α and β, which indicate the amount of smoothing to apply to the model, and the LDA parameter K, which indicates the number of topics. The indexing process can be divided into text extraction, preprocessing, and term weighting stages, all of which are configurable.

Although there are standard parameter configurations used in the natural language processing (NLP) domain, Marcus and Menzies state that the optimal configurations of TR tools when used in the software engineering domain are different than these standards [2010]. They also claim that the accuracy of TR techniques is highly dependent on the configuration of the tool. Their claim is supported by recent research (e.g., [Biggers, Bocovich, Capshaw, Eddy, Etzkorn, and Kraft, 2012; Dit, Guerrouj, Poshyvanyk, and Antoniol, 2011; Hill, Rao, and Kak, 2012]).

Configuring the text extraction and preprocessing stages for the LDA-based FLT and others has been the subject of recent research. Configuration of the text extraction phase and LDA parameters is studied by Biggers et al. [2012]. Hill et al. investigate the effect of stemmer selection on FLT accuracy [2012], and Biggers and Kraft [2012] similarly compare the effects of various stemmers on LSI and LDA based FLTs. Dit et al. discuss how identifier splitting affects FLT performance [2011]. However, no research has been performed on the selection or configuration of term weighting schemes in the software engineering domain. In this research we investigate configuring the term weighting stage for LDA.

1.2 Structural Term Weighting

Term weighting is the process of adjusting the importance of a term when computing the similarity between documents in a corpus. The intention is to place more weight on important terms in a document. This can accomplished by reducing the weight of common words, or increasing the weight of terms shared by a small number of documents. Numerous text weighting schemes have been developed and evaluated for algebraic TR techniques such as VSM and LSI [Salton and Buckley, 1988], but it is not common to use term weighting with probabilistic models such as LDA. Recent research has investigated methods for weighting LDA in the NLP and machine learning domains [Wang, Zhang, Xie, Anvik, and Sun, 2008; Wilson and Chew, 2010], but analogous work has not been attempted in the software engineering domain. Moreover, we believe that our approach is unique across both the NLP and software engineering domains.

We propose *structural term weighting*, a new approach to informing a probabilistic TR system about term relevancy. We define structural term weighting as a term weighting scheme that modifies term relevancy based on the structural information in source code. For example, a function's name is typically indicative of its behavior or purpose and is likely a better indicator

of a function's relevance to query than is a local variable's name. However, when a function is converted into a document during the source code indexing process, terms derived from the function name may have a relatively low number of occurrences, especially if the function is non-recursive. To more accurately reflect the importance of the function name, the terms associated with the name can be weighted by some factor in the resulting document.

1.3 Research Question

We perform a case study to address the following research question:

- *RQ: Does structural term weighting affect the accuracy of an LDA based FLT?*

We specifically apply a multiplier to the counts of terms originating from certain structural classes of identifiers. The structural classes we chose for the study are method names and method calls. We then use these modified term counts as part of the input to an LDA based FLT and compare the effectiveness of sixteen configurations.

1.4 Organization

The remainder of this thesis is organized as follows. In Chapter 2 we outline the source code indexing and retrieval process used by text retrieval based FLTs. We also discuss prior research of LDA and feature location. In Chapter 3 we present the design and results of our empirical study. In Chapter 4 we conclude and describe our plans for future work.

CHAPTER 2

BACKGROUND & RELATED WORK

In this chapter we describe the source code indexing process, including terms often used in the indexing domain and the steps in the process. In particular we focus on the term weighting step. We also review LDA and related work on feature location.

2.1 Source Code Indexing and Retrieval Process

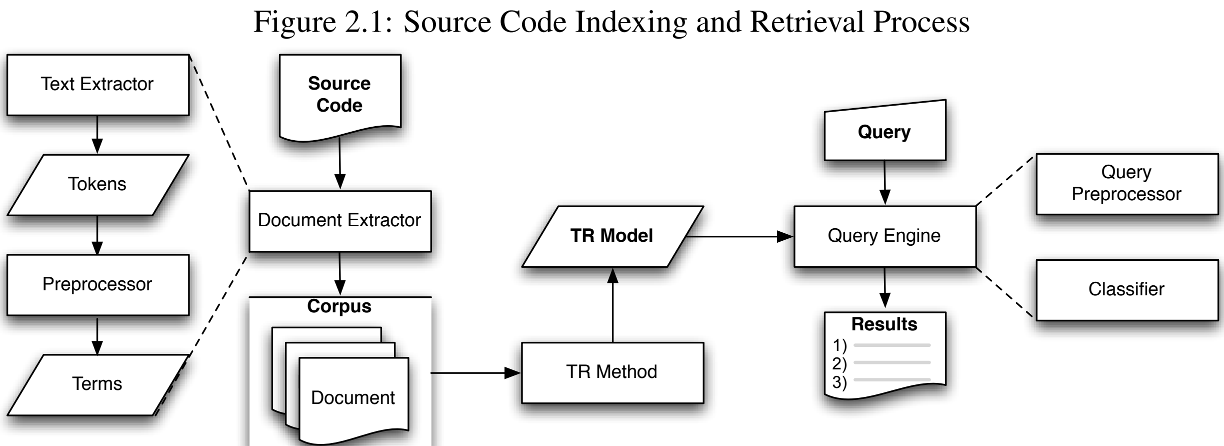
The source code indexing and retrieval process, as illustrated in Figure 2.1, defines a method for processing source code into a format suitable for efficient querying of the system, and producing the result of such a query. Terminology is defined in this section, followed by an overview of each step of the process.

2.1.1 Terminology

We use the terminology of Biggers et al. [2012]. They provide the following definitions:

- *Term*: a sequence of letters and the basic unit of discrete data in a lexicon

Figure 2.1: Source Code Indexing and Retrieval Process



- *Token*: a sequence of non-whitespace characters; contains one or more terms
- *Entity*: a source element such as a class or method
- *Identifier*: a token representing the name of an entity
- *Comment*: a sequence of tokens delimited by language specific markers, e.g., /* */
- *String literal*: a sequence of tokens delimited by “ ”
- *Word*: the smallest free form in a language

Further, a term is described as being one of: word, abbreviation of a word, contraction of one or two words, or acronym of a series of words.

2.1.2 Indexing

The indexing process is illustrated in the left section of Figure 2.1. It converts source code into a corpus, or collection of documents. A document is a collection of terms that appear in a source code entity. Each entity has an associated document in the resulting corpus.

2.1.2.1 Text Extraction

The text extraction stage receives as input source code and produces a list of tokens. The text extractor can be configured to tokenize some combination of the comments, string literals, and identifiers present in the source code. Biggers et al. evaluate the relative performance of these combinations [2012].

2.1.2.2 Preprocessing

The preprocessing stage takes each token produced by the text extractor and applies a series of processing steps, producing one or more terms. The steps most commonly used [Marcus and Menzies, 2010; Marcus, Sergeyev, Rajlich, and Maletic, 2004] are the following:

- *Splitting*: divide a token into terms based on common coding conventions such as camel case and underscores and on punctuation or other symbols. The original term can be kept or discarded.

- *Normalizing*: convert all uppercase characters to lowercase, or vice-versa.

- *Filtering*: discard articles (e.g., 'the', or 'a'), programming language specific keywords, common words as defined by a stop-list, and short words.

- *Stemming*: strip words of prefixes and suffixes to leave a common root and to allow for conflation of different forms of the same word (e.g., 'stem', 'stemmer', 'stemming').

2.1.2.3 Term Weighting

The term weighting stage assigns a numeric value, or weight, to each term in the corpus. Common term weighting schemes described in the literature are binary, term count, term frequency, and term frequency-inverse document frequency (tf-idf) [Salton and Buckley, 1988]. Binary term weighting assigns a 1 for all terms that appear in the associated source code entity, and 0 for those that do not. It is used to represent term sets when the corpus is represented as a matrix. Term count assigns each term a number corresponding to the number of times that term appears in a document. Term frequency normalizes this number relative to the highest term count in the document. Finally, in tf-idf, the terms are assigned a weight that increases with term frequency in a specific document, but decreases with the number of distinct documents containing the term.

These term weighting schemes are frequently used in the realm of algebraic TR methods (e.g., VSM and LSI), while probabilistic TR methods (e.g., LDA) have been assumed to not need to be term weighted. Wilson and Chew evaluatedd term weighting schemes through the modification of Gibbs sampling [2010]. Wang et al. present a modification to LDA for cluster ensembles in

which weights drawn from a multinomial distribution are assigned to base clusters [2008]. However, we are unaware of any previous method that is similar to what we propose, and ours is the first research investigating the use of term weighting with probabilistic TR methods in the software engineering domain.

2.1.3 Model Generation

Model generation is depicted in the center of Figure 2.1. This stage takes a corpus as input and processes it using a TR method to produce a model of the source code as output. The TR methods commonly used in this stage include the VSM [Salton, 1989], LSI [Deerwester et al., 1990], and LDA [Blei et al., 2003].

2.1.4 Retrieval

The retrieval process is displayed as the right side of Figure 2.1. It takes a query as input and produces a ranked list of similar documents, each associated with a distinct source code entity, as output.

2.1.4.1 Querying

The query is a string generated manually or automatically (e.g., from an issue report). This query must be processed the same way as a document in the corpus, and so the query processor shown in Figure 2.1 will be internally similar to the document extractor. This produces a document associated with the query. It may have to be further processed to be directly comparable with elements in the TR model.

2.1.4.2 Ranking

The classifier shown in Figure 2.1 is applied pairwise to the query document and each document in the model to score each pairing on similarity. These scores are then used to rank the documents by descending similarity.

2.2 Latent Dirichlet Allocation

Latent Dirichlet allocation (LDA) is a probabilistic generative model of a corpus used for TR, introduced by Blei et al. [2003]. Each document in the corpus is modeled as a mixture of latent topics, and each topic is modeled as a mixture of the terms in the corpus. In other words, each document is represented by a probability distribution describing the likelihood of being related to a topic generated by LDA, while each topic is represented by a probability distribution describing the likelihood of a term in the vocabulary belonging to that topic.

Inputs to LDA are:

- D, the documents
- K, the number of topics
- α, the Dirichlet hyperparameter for topic proportions
- β, the Dirichlet hyperparameter for topic multinomials

The documents provided are considered a bag-of-words, represented as a vector of length V, the size of the vocabulary. The original word ordering and grammatical information is discarded.

Outputs of LDA are:

- ϕ, the term-topic probability distribution
- θ, the topic-document probability distribution

The parameters α and β are used to control the smoothing of the model. Topic distribution per document is influenced by α, and term distribution per topic is influenced by β. Decreasing the value of α allows for fewer topics to be associated with a document, while decreasing the value

of β generates topics that produce fewer terms (increasing the number of topics needed to model the corpus). Decreasing these values will then make the computed probability distributions, ϕ and θ, more specific, increasing the decisiveness of the model.

Griffiths and Steyvers [2004] observe that β controls the model granularity. Decreasing β increases the number of topics required and influences each topic to be more detailed and specific.

Model inference is generally intractable [Blei et al., 2003]. This necessitates approximate inferencing algorithms, such as Collapsed Gibbs Sampling (CGS) [2003], a form of Markov-chain Monte Carlo (MCMC). CGS iteratively computes term topic distributions until the model converges or a maximum number of iterations, σ, have completed [Griffiths and Steyvers, 2004].

It is possible to infer a probability distribution for a new document, for example some query Q, given an existing LDA model. A similarity measure for probability distributions, such as Hellinger distance, can be used to make pairwise comparisons between the topic distribution for the new document, θ_Q, and for a document in the corpus, θ_i. Hellinger distance (H) is given by:

$$H(\theta_Q, \theta_i) = \left[\frac{1}{2} * \sum_k \left(\sqrt{(\theta_{Qk})} - \sqrt{\theta_{ik}} \right)^2 \right]^{\frac{1}{2}} \tag{2.1}$$

where θ_{Qk} and θ_{ik} are the kth probabilities in distributions θ_Q and θ_i.

2.3 Feature Location

Feature location is a program comprehension activity involving the location of a source code element involved in the implementation of a feature (i.e, an observable functionality) [Rajlich and Wilde, 2002]. If the feature is unwanted (e.g., a bug), the process is sometimes referred to as bug localization [Lukins et al., 2008, 2010]

Feature location techniques are often categorized as static, dynamic, or a blend of the two.

A static technique (e.g., [Biggers et al., 2012]) uses source code text, while a dynamic technique (e.g., [Eisenberg and De Volder, 2005]) instead uses an execution trace from a run of the system. Blended techniques (e.g., SITIR [Liu, Marcus, Poshyvanyk, and Rajlich, 2007] or PROMESIR [Poshyvanyk, Gueheneuc, Marcus, Antoniol, and Rajlich, 2007]) use some combination of source code text and execution traces to locate a feature. The use of a dynamic feature location technique requires a working system and an execution suite to generates traces that exercise the feature of interest, neither of which is necessary for static techniques. Latent Dirichlet allocation based feature location, the focus of this research, is a static technique.

A taxonomy and survey of feature location techniques is presented by Dit et al. [2011]. Scanniello and Marcus [2011] compare a technique combining VSM and clustering to VSM alone. They test using three distinct queries and find that the effectiveness of the technique varies by system.

Marcus et al. [2004] introduce the first LSI based FLT. This work was extended to include relevance feedback, a process in which a user is able select which results given by an initial run are relevant to produce a new query [Gay, Haiduc, Marcus, and Menzies, 2009].

Multiple tools (SNIAFL [Zhao, Zhang, Liu, Sun, and Yang, 2006], Dora [Hill, Pollock, and Vijay-Shanker, 2007], LSICG [Shao, Atkison, Kraft, and Smith, 2012]) incorporate the use of call graph analysis as additional static data with the use of a TR method.

Ratanotayanon et al. [2010] evaluate using varied sources of static data (e.g., change sets, issue trackers, dependency graphs) to improve feature location. However, the results of their study show that using a variety of sources does not always improve performance.

Liu et al. [2007] present SITIR, which is a blended technique. A user will execute the system once to obtain an execution trace exercising the feature and then provide a query. LSI is then

used to rank the executed methods based on similarity to the provided query. The performance of SITIR is compared to LSI, Scenario-based Probabilistic Ranking (SPR, a dynamic technique), and PROMESIR (a combination of LSI and SPR) [Poshyvanyk et al., 2007]. They find that both PROMESIR and SITIR perform marginally better than LSI, and much better than SPR. PROMESIR is shown to perform slightly better than SITIR, but SITIR uses only a single execution trace, while SPR and PROMESIR must be provided multiple traces — some that do exercise the feature, and some that do not. SITIR is also shown to improve LSI by being less sensitive to badly formulated queries.

Revelle et al. [2010] extend SITIR by incorporating dependence information gathered by web-mining as another source of static data.

Lukins et al. [2008; 2010] evaluate an LDA based feature location technique. The results of Poshyvankyk et al. [2007] are used to compare the performance of their method to an LSI-based technique. It is shown that for Eclipse, LDA outperforms LSI. The results for Mozilla show an improvement when using LDA, but note a sensitivity to badly formulated queries. This behavior is also present in LSI-based techniques [Liu et al., 2007]. The stability of the source code does not affect the performance of the technique.

Dit et al. [2011] study the effect of three identifier splitters on the performance of LSI and LSI incorporating dynamic analysis. Their evaluation was performed on two open-source Java systems. Their results indicate that feature location techniques using TR methods could benefit from better identifier splitting, and that manually splitting identifiers yielded higher accuracy than the current state-of-the-art.

Biggers et al. [2012] investigate possible configurations for an LDA based feature location technique. They find that excluding comments in literals when extracting text can decrease the

accuracy of the technique. They conclude that standard parameter values used in information retrieval literature should not be applied directly when working with a corpus derived from source code.

CHAPTER 3

EVALUATION OF STRUCTURAL TERM WEIGHTING

We conducted an empirical study to evaluate the effect of a structural term weighting scheme on the performance of an LDA-based feature location technique. In this section, we describe the experimental context and design, as well as present and evaluate the resulting data.

3.1 Definition and Context

This study compares the accuracy of an LDA-based feature location technique when using sixteen structural term weighting configurations. We search for over 400 features and bugs from five open source Java systems. The purpose of the study is to determine if using structural term weighting can improve the performance of an LDA-based FLT.

3.1.1 Weighting configurations

We evaluate the effectiveness of increasing the weights of certain terms based on structural information. We consider specifically terms derived from either a method name or method call. We hypothesized that a method's name often contains terms that describe its behavior or purpose. Unless the method is recursive, these terms may not be repeated elsewhere in the document, and would receive less weight than may be desired. Also, method calls denote the use of one functionality (implemented by the callee) to realize another (implemented by the caller). The terms derived from these calls could also be important in describing the functionality of a method. We considered increasing the weight of method call terms as a possible lightweight alternative to adding call

graph analysis. Multiple tools (SNIAFL [Zhao et al., 2006], Dora [Hill et al., 2007], LSICG [Shao et al., 2012]) have successfully integrated call graph analysis into TR based FLTs.

Because LDA uses a term count weighting scheme, we multiplied term occurrences by some scaling factor. We evaluate the use of 1, 2, 4, and 8 as scaling factors. When referring to a specific configuration, we use the notation $C(n,c)$, where n denotes the name scaling factor, and c the call scaling factor. We allow n and c to vary independently, producing sixteen possible configurations. Configuration $C(1,1)$ is standard unweighted LDA.

3.1.2 Subject systems

We evaluate FLT effectiveness on five subject systems — ArgoUML[1], Eclipse[2], JabRef[3], jEdit[4], and muCommander[5]. These systems have freely available source code, represent a large range of application domains and sizes, and are similar to systems produced in industry.

ArgoUML is a UML diagramming tool, and Eclipse is an IDE. JabRef is a bibliography management tool, and jEdit is a text editor designed for programming. muCommander is a cross-platform file browser. Table 3.1 specifies which versions of the software systems we are using. It also lists lines of code (both source and comment), method count and number of features we query for each system.

3.1.3 Benchmarks and gold sets

The feature queries we used to evaluate the FLT configurations are drawn from issues posted to the issue tracking system or bug repository associated with each subject system. These issue reports detail a "bug" (i.e., an unwanted feature), or request additional functionality (i.e., a

[1] `http://argouml.tigris.org`
[2] `http://www.eclipse.org`
[3] `http://jabref.sourceforge.net`
[4] `http://www.jedit.org`
[5] `http://www.mucommander.com`

Table 3.1: Subject Systems

System	Version	SLOC	CLOC	Methods	Features
ArgoUML	0.22	117,649	104,037	11,348	91
Eclipse	3.0	1,255,149	704,092	126,744	45
JabRef	2.6	74,350	25,927	5,323	39
jEdit	4.3	98,460	42,589	6,550	150
muCommander	0.8.5	76,649	68,367	8,811	92
Total		1,622,257	945,012	158,776	417

new feature). We used a set of benchmarks created by other researchers and made available to the research community[6]. These benchmarks contain sets of methods that were changed in order to resolve an issue posted on the issue tracking system associated with the subject system. The required methods were extracted from Subversion diffs. These sets have been termed "gold sets" [Dit et al., 2011; Poshyvanyk et al., 2007] and are assumed to be the set of methods implementing the featured described in the issue report.

Descriptive statistics about the gold sets are listed in Table 3.2.

Table 3.2: Numbers of methods in the gold sets.

System	Min	Median	Mean	Max	StdDev	Total
ArgoUML	1	3	7.70	72	13.11	701
Eclipse	1	2	2.87	22	3.34	129
JabRef	1	4	7.18	33	8.78	280
jEdit	1	3	4.99	41	5.58	748
muCommander	1	3	7.80	104	16.06	718

The benchmarks provide gold sets for 417 features across the five subject systems. The queries for the features are automatically generated from concatenating the associated issue re-

[6] http://www.cs.wm.edu/semeru/data/benchmarks

port's title and description [Dit et al., 2011]. The automatic generation of queries is adequate to model a query that could be provided if the user had no other knowledge of the system.

3.1.4 Effectiveness measure

Evaluating the performance of a TR-based FLT using traditional measures of recall and precision is unhelpful. Because the FLT can and does produce every method in the system in its list of ranked documents, recall is always 1, and precision is always $1/k$, where k is the number of documents in the corpus. Instead, Poshyvanyk et al. define an effectiveness measure [2007] that can be used for TR-based FLTs. This effectiveness measure is the rank of the first relevant document in the list produced by the FLT. It quantifies the number of source code entities a developer would have to search before reaching a relevant method.

The mean reciprocal rank (MRR) of the FLT is given by the average of the reciprocal of the effectiveness measure given some sample set of queries [Voorhees, 1999]:

$$\text{MRR} = \frac{1}{|Q|} \sum_{i=1}^{|Q|} \frac{1}{r_i} \tag{3.1}$$

where Q is the set of queries and r_i is the rank of the first relevant method for some query Q_i. A higher MRR implies a more effective FLT.

3.1.5 Methodology

We use a text extractor and preprocessor implemented in Python v2.6 using an open source Java 1.5 grammar and ANTLR v3. The tool extracts documents from methods and treats inner methods as distinct methods. The text of inner method (e.g., a method inside an anonymous class) is only attributed to that method, and not the containing one. Comments within a method, as well as block comments immediately preceding a method, are all considered text of the method.

During text extraction, terms that are produced by certain grammar rules are duplicated according to the current structural term weighting configuration. We use the term weighting configurations described in Section 3.1.1.

The preprocessor follows the steps depicted in Figure 2.1. Identifiers from `java.lang` are filtered out. Remaining tokens are split on camel case, underscores, and non-letter symbols. The original token is retained. The terms are all normalized to lower case and then filtered by an English stop word list [Fox, 1992] and Java keyword list. Terms shorter than three characters are also removed. A Porter stemmer[7] is then applied to every term.

The weighted corpus is then provided as input to Mallet, which we use to generate and query LDA models. For the configuration parameters for the LDA model generation, we use the suggested values from Biggers et al. [2012]. That is, $\alpha = 1.0$, K varying on system size, and $\beta = 0.5$ for small systems, varying inversely proportional to K. For inferencing, Mallet uses a CGS algorithm that requires a parameter σ, the number of sweeps to make over the corpus. We set $\sigma = 1000$ to balance accuracy and speed. The queries are automatically generated from issue reports as described in Section 3.1.3. The classifier we use is Hellinger distance.

3.2 Research Question and Hypotheses

Our case study addresses the following research question:

- *RQ: Does structural term weighting affect the accuracy of an LDA-based FLT?*

The independent variables in our study are the scaling factors applied to terms derived from a method's name or method calls. We consider the scaling factors 1, 2, 4, and 8.

[7] `http://tartarus.org/~martin/PorterStemmer/python.txt`

When formulating hypotheses, we do not presuppose the directionality of the difference between any two configurations. Therefore, all our hypotheses are two-sided. We formulate null hypotheses to evaluate if using any configuration has a significantly different result compared to using any other. For example:

$H_0 : C(2,4) = C(8,4)$

> Configuration $C(2,4)$ **does not significantly affect** the accuracy of the LDA-based FLT compared to configuration $C(8,4)$.

The remaining 119 null hypotheses are of the same form. If, after testing the null hypotheses, we find we can reject it with a high confidence ($\alpha = 0.05$), we accept an alternative two-sided hypothesis. This hypothesis states that using a configuration does have a significantly different result compared to using another configuration. For example:

$H_A : C(4,1) \neq C(8,2)$

> Configuration $C(4,1)$ **does significantly affect** the accuracy of the LDA-based FLT compared to configuration $C(8,2)$.

The remaining 119 alternative hypotheses are of the same form.

3.3 Data Collection & Analysis

For each benchmark and configuration pairing, we produce a list of effectiveness measures corresponding to the queries in the benchmark. The size of this list is equal to the number of queries in the benchmark for the system being tested. For example, the benchmark for ArgoUML contains 91 features, so for each of the sixteen configurations a list of 91 effectiveness measures is produced. These ranking lists are used in the following analyses.

We first produce descriptive statistics of the data sets for each system-configuration pair including min, first quartile, median, third quartile, and max. This provides a high-level view of the data sets. We illustrate these statistics using boxplots.

We report the Mean Reciprocal Rank, as described in Section 3.1.4, for each configuration over all queries for each benchmark. These values range from 0 to 1, with larger values indicating better results. We report the results for each system in a table, and for performance overall as a bar chart.

We then perform a Friedman test, the non-parametric analog of the (parametric) one-way repeated measures ANOVA. We use the Friedman test to indicate if, for a particular benchmark, there is any statistically significant effect of using one configuration over another. If the Friedman test reports a significant effect, we then use a post-hoc pairwise Wilcoxon signed-rank test with Holm p-value correction. This test allows us to determine which configuration pairings result in statistically significant differing results. We report the results of the Friedman test as a table.

Finally, we analyze the results of two queries qualitatively. We report the query documents and method documents as generated by Mallet as tables and list the source code of the relevant methods.

3.4 Results

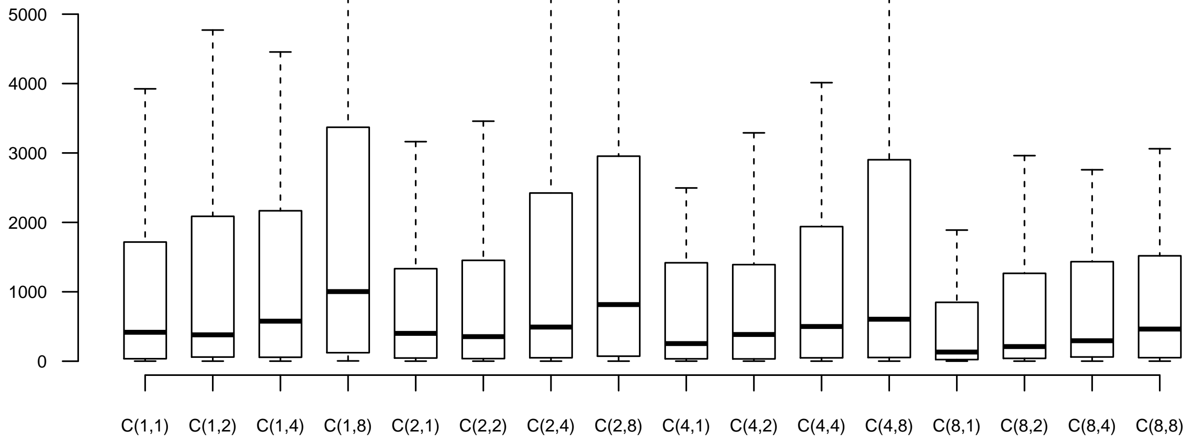
In this section we report the results of using sixteen different structural term weighting configurations (see Section 3.1.1) on the effectiveness measures of five benchmarks (see Section 3.1.3) when using an LDA-based FLT.

3.4.1 Descriptive

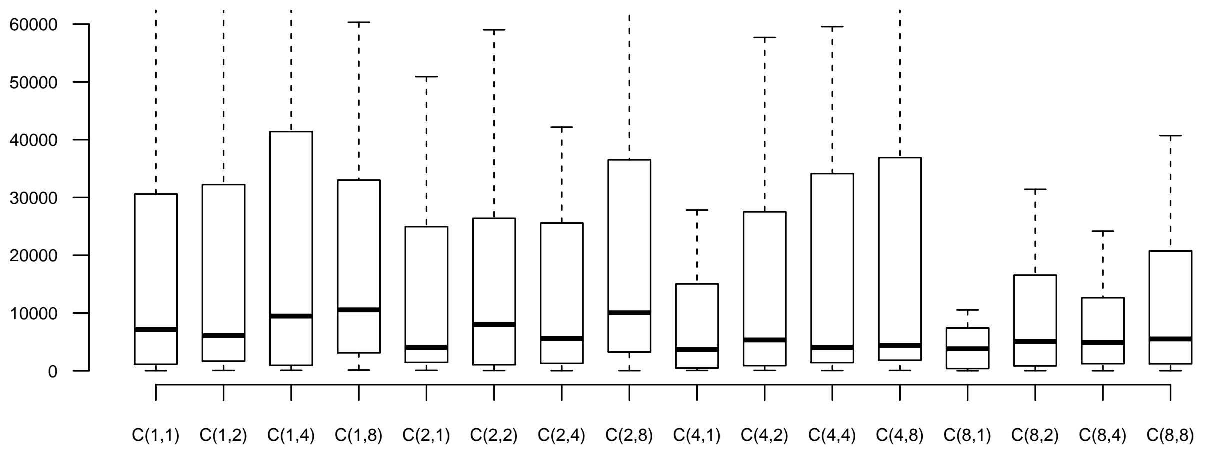
Figure 3.1 illustrates the descriptive statistics of the effectiveness measures for each benchmark. Note that Figure 3.1f is a boxplot for the combined set of 417 features.

We note the general trend of better performance (i.e., lower rank) when increasing the method name multiplier while holding the method call multiplier constant. For example, for JabRef with $C(1,8)$ we obtain a median of 448 and with $C(4,8)$ a median of 75. Similarly, we note decreased performance when increasing the method call multiplier while keeping the method name multiplier constant. For example, for ArgoUML with $C(1,1)$ we obtain a median of 417 and with $C(1,8)$ a median of 1003.

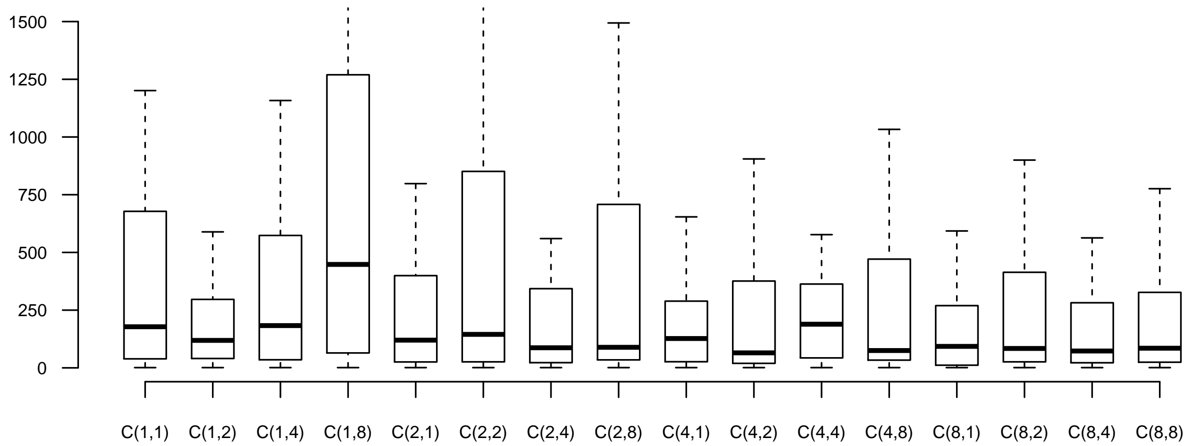
Figure 3.1: Boxplots of Effectiveness Measures



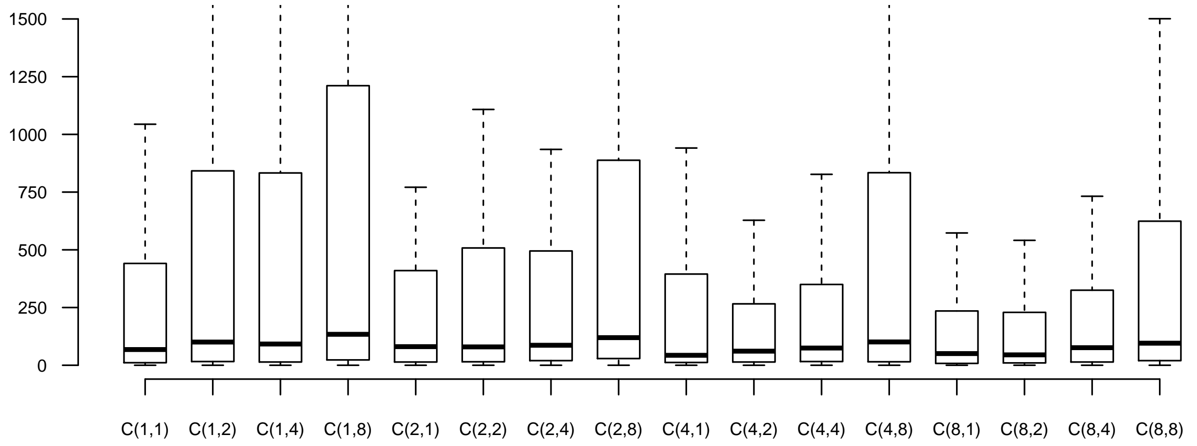
(a) ArgoUML



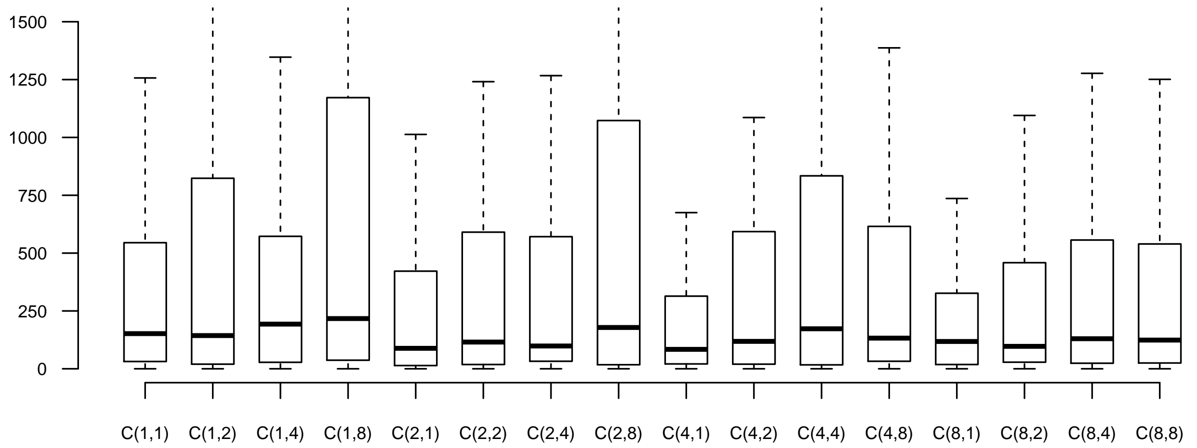
(b) Eclipse



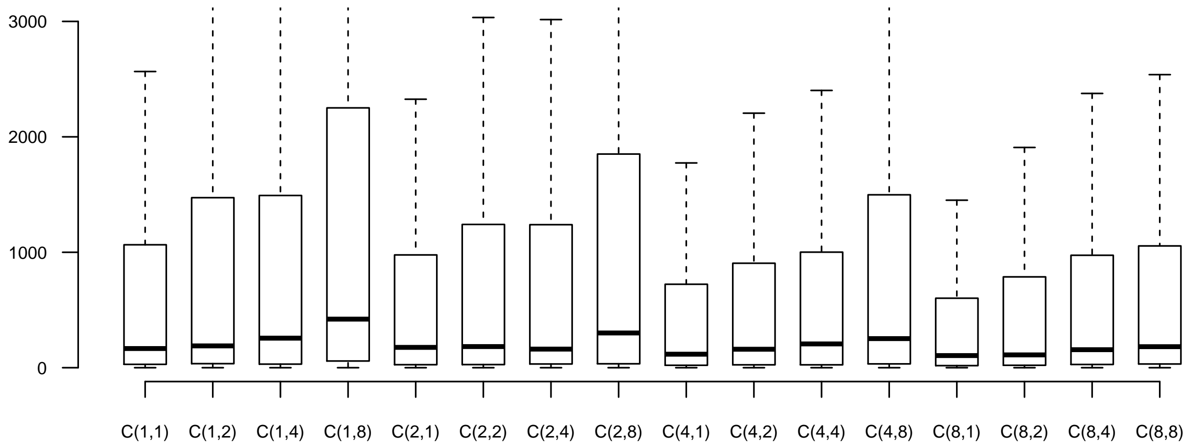
(c) JabRef



(d) jEdit



(e) muCommander



(f) All

3.4.2 Mean reciprocal rank

We report the mean reciprocal rank (MRR) as described in Section 3.1.4 for each configuration and subject system in Table 3.3.

Figure 3.1 illustrates the MRR over all 417 features. This chart shows decreasing performance (i.e., lower MRR) when increasing the method call multiplier, and increasing performance when increasing the method name multiplier.

Table 3.3: Mean Reciprocal Rank for each Configuration and System

	Argo	Eclipse	JabRef	jEdit	muCom
$C(1,1)$	0.088	0.003	0.080	0.128	0.078
$C(1,2)$	0.034	0.001	0.093	0.094	0.116
$C(1,4)$	0.034	0.001	0.086	0.087	0.047
$C(1,8)$	0.018	0.001	0.041	0.058	0.060
$C(2,1)$	0.058	0.001	0.049	0.096	0.098
$C(2,2)$	0.054	0.001	0.078	0.116	0.089
$C(2,4)$	0.050	0.002	0.087	0.069	0.058
$C(2,8)$	0.044	0.001	0.084	0.059	0.068
$C(4,1)$	0.065	0.002	0.086	0.094	0.090
$C(4,2)$	0.039	0.002	0.110	0.109	0.131
$C(4,4)$	0.037	0.001	0.088	0.087	0.071
$C(4,8)$	0.063	0.001	0.075	0.085	0.109
$C(8,1)$	0.081	0.004	0.127	0.133	0.100
$C(8,2)$	0.065	0.003	0.088	0.125	0.079
$C(8,4)$	0.036	0.002	0.095	0.095	0.070
$C(8,8)$	0.051	0.003	0.102	0.077	0.082

3.4.3 Statistical analysis

We conducted a Friedman test for each of the five subject system to determine if structural term weighting has a significant effect when using an LDA-based FLT on that system. The chi-squared and p-values for each system are listed in Table 3.4. The test revealed significant effects for all five systems, so we performed a post-hoc Wilcoxon signed-rank test with Holm correction for the data set of each system.

Figure 3.1: Mean Reciprocal Rank for each Configuration over all Queries

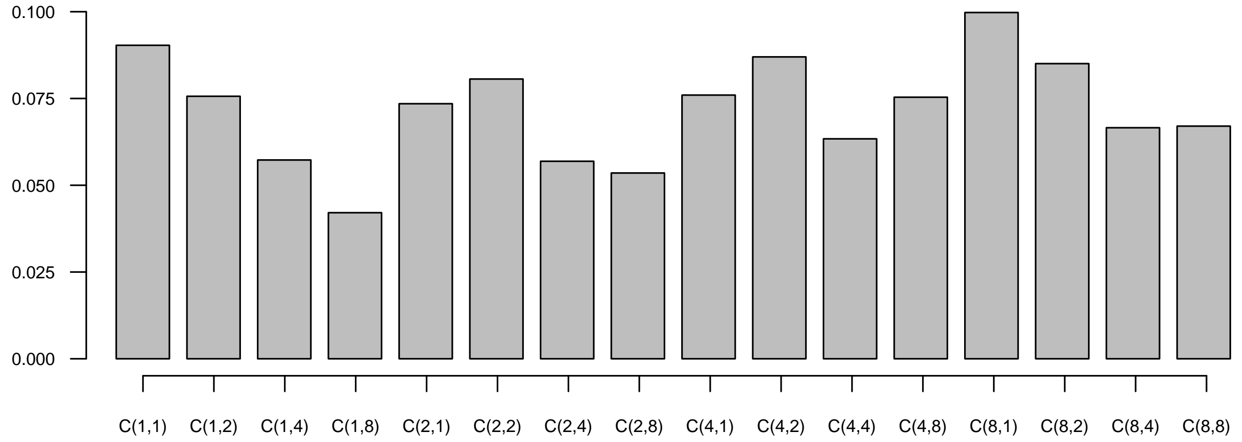


Table 3.4: Friedman Test Results

System	χ^2	df	p-value
ArgoUML	106.316	15	< 0.01
Eclipse	57.178	15	< 0.01
JabRef	31.570	15	< 0.01
jEdit	154.809	15	< 0.01
muCommander	47.754	15	< 0.01

The post-hoc Wilcoxon signed-rank tests reported many configuration pairs with statistically significant differences for each system ($p < 0.05$). There were 24 such pairs for ArgoUML, 5 for Eclipse, 3 for JabRef, 43 for jEdit, and 4 for muCommander.

When comparing with unweighted LDA (i.e., $C(1,1)$), only ArgoUML, and jEdit exhibited configurations with statistically significant differences. For ArgoUML, $C(1,8)$ ($p < 0.001$) and $C(8,1)$ ($p < 0.04$) are signficantly different, whereas for jEdit, $C(1,2)$ ($p < 0.03$), $C(1,8)$ ($p < 0.001$), and $C(2,8)$ ($p < 0.02$) are significantly different.

We also combined the data sets for all five systems and conducted a Friedman test over all 417 query results, revealing a significant effect ($\chi^2(15) = 332.3, p < 0.001$).

Table 3.5: Query document for ArgoUML feature 3911

Query Document
incorrect remov popup menu action argument incorrect remov popup menu action argument select action properti panel field argument rightclick remov latter wrong accord terminolog page argouml argo uml delet model

Post-hoc tests using Wilcoxon signed-rank test with Holm correction revealed 58 configuration pairings with statistically significant differences ($p < 0.05$). In particular, the post-hoc tests revealed three configurations with statistically significant differences to $C(1,1)$: $C(1,8)$ ($p < 0.001$), $C(2,8)$ ($p < 0.001$), and $C(8,1)$ ($p < 0.001$).

3.4.4 Qualitative analysis

Our statistical analysis showed a significant effect from structural term weighting. In the next few paragraphs we present qualitative analysis of some example features.

Consider the query document for ArgoUML feature 3911 as displayed in Table 3.5. The correct method to return for this query is the constructor of `ActionRemoveArgument`. The source for this method is displayed in Figure 3.2. This constructor is a very short piece of code, and the terms in the method name are repeated in the comment block directly above the method. As is listed in Table 3.6, the document for this method produced using $C(1,1)$ is largely composed of method name terms already. The repetition of those terms in $C(8,1)$ do not harm the accuracy of the FLT, as both these configurations return this method as their highest ranked possibility. However, $C(1,8)$ duplicates the term *local*, which is not a term describing the feature of interest. $C(1,8)$ does not report the correct method until rank 225.

Now consider the query document for jEdit feature 1931333 as displayed in Table 3.7. The correct method to return for this query is `CompleteWord.keyTyped`. The source for this method is listed in Figure 3.3. For this query, $C(8,1)$ performs much better than $C(1,1)$ reporting

Figure 3.2: The source code for ArgoUML `ActionRemoveArgument` class constructor

```
/**
 * Constructor for ActionRemoveArgument.
 */
protected ActionRemoveArgument() {
    super(Translator.localize("menu.popup.delete"));
}
```

Table 3.6: Method documents for ArgoUML `ActionRemoveArgument` class constructor

Config	Rank	Method document
$C(1,1)$	1	actionremoveargu action remov argument translat local constructor actionremoveargu action remov argument menu popup delet
$C(8,1)$	1	actionremoveargu action remov argument actionremoveargu action remov argument actionremoveargu action remov argument actionremoveargu action remov argument actionremoveargu action remov argument actionremoveargu action remov argument actionremoveargu action remov argument actionremoveargu action remov argument translat local constructor actionremoveargu action remov argument menu popup delet
$C(1,8)$	225	actionremoveargu action remov argument translat local local local local local local local local constructor actionremoveargu action remov argument menu popup delet

Table 3.7: Query document for jEdit feature 1931333

Query Document
pre append text use autocomplet select complet autocomplet popup appen text reproduc creat buffer type foo foobar type hit ctrl autocomplet autocomplet popup display altern foo foobar type select altern complet foobar instead foo

Table 3.8: Method documents for jEdit method `CompleteWord.keyTyped(keyEvent)`

Config	Rank	Method document
$C(1,1)$	94	keytyp key type getkeychar key char isdigit digit index index index index index getcandid candid getsiz size setselectedindex set select index index doselectedcomplet select complet consum dispos isletterordigit letter digit nowordsep word sep indexof index doselectedcomplet select complet textarea text userinput user input consum dispos textarea text userinput user input consum resetword reset word word keyev key event keytyp key type medhod fall handl foo insert foobar
$C(8,1)$	1	keytyp key type keytyp key type keytyp key type keytyp key type keytyp key type keytyp key type keytyp key type keytyp key type getkeychar key char isdigit digit index index index index index getcandid candid getsiz size setselectedindex set select index index doselectedcomplet select complet consum dispos isletterordigit letter digit nowordsep word sep indexof index doselectedcomplet select complet textarea text userinput user input consum dispos textarea text userinput user input consum resetword reset word word keyev key event keytyp key type medhod fall handl foo insert foobar

this method as its first result as opposed to result 94. The method documents for these configurations are listed in Table 3.8. $C(8,1)$ includes the term *type* more frequently in its document, which may have contributed to the increased performance for this query, as the query includes *type* three times in its relatively small document.

3.5 Discussion

The analysis above suggests some notable conclusions. By combining the descriptive statistics and mean reciprocal rank data with the results of the post-hoc tests, we can observe that some configurations have a statistically significant effect when compared to unweighted LDA (i.e., $C(1,1)$).

Figure 3.3: The source code for the jEdit method `CompleteWord.keyTyped(keyEvent)`

```
protected void keyTyped(KeyEvent e)
{
    char ch = e.getKeyChar();
    if(Character.isDigit(ch))
    {
        int index = ch - '0';
        if(index == 0)
            index = 9;
        else
            index--;
        if(index < getCandidates().getSize())
        {
            setSelectedIndex(index);
            if(doSelectedCompletion())
            {
                e.consume();
                dispose();

            }
            return;
        }
        else
            /* fall through */;
    }

    // \t handled above
    if(ch != '\b' && ch != '\t')
    {
        /* eg, foo<C+b>, will insert foobar, */
        if(!Character.isLetterOrDigit(ch) && noWordSep.indexOf(ch) == -1)
        {
            doSelectedCompletion();
            textArea.userInput(ch);
            e.consume();
            dispose();
            return;
        }

        textArea.userInput(ch);
        e.consume();
        resetWords(word + ch);
    }
}
```

There are configurations that reported better performance than $C(1,1)$. For ArgoUML, the post-hoc Wilcoxon signed rank test showed a statistically significant difference in using $C(1,1)$ and $C(8,1)$. This result was also present when considering the combined data set of all 417 features. The descriptive statistics and mean reciprocal rank values show that in these instances $C(8,1)$ performs better than $C(1,1)$. Although not statistically significant, similar differences can be seen for the other systems as well.

There were also configurations that performed worse than $C(1,1)$. For example, $C(1,8)$ performed worse than $C(1,1)$ for ArgoUML, jEdit, and the full 417 feature dataset.

We also note that the boxplots suggest a diminished interquartile range, shown by the distance from box to whisker, when increasing the method name multiplier. This trend extends across all our studied systems, most strongly when using a method name multiplier of 8. For example, with Eclipse, $C(8,1)$ – $C(8,8)$ show substantially lower whiskers than configurations with lower method name multipliers.

Our results suggest that increasing the weight of terms originating from method names can improve accuracy, but increasing the weight of terms originating from method calls can decrease accuracy. Given the results, we would suggest using a multiplier of 8 for method name terms, while not multiplying method call terms.

3.6 Threats to Validity

In this section we describe and address some limitations that may affect the validity of our findings and our ability to generalize them.

Threats to conclusion validity concern inaccuracies in our conclusions about the relationships in our observations. We did not assume any particular distribution of the effectiveness mea-

sures. We used non-parametric statistical tests and used a p-value adjustment method to account for family-wise error rate.

Threats to construct validity concern measurements accurately reflecting the concepts of interest. A possible threat to construct validity is our benchmarks. Errors in our gold sets could result in inaccurate effectiveness measures. However, these gold sets have been used in previous research [Dit et al., 2011; Dit, Revelle, Gethers, and Poshyvanyk, 2013; Revelle et al., 2010]. The gold sets were produced by other researchers, and are made publicly available online.

Threats to internal validity include possible errors in executing the study or defects in our tool chain could affect our results and conclusions. We throughly tested our toolchain and reviewed the data resulting from each phase of the case study to control these threats. The same tool chain was applied to each system in the study, so any errors are systematic and should not substantially affect our results.

The queries we used, composed of titles and descriptions from issue trackers, are another possible threat to internal validity. They may not accurately describe the features of interest. However, three of the five systems are primarily used by software developers, who are likely to describe a desired or faulty feature more accurately than a typical end-user.

Threats to external validity concern how widely our findings can be generalized. All five of our subject systems are written in Java, so we are unable to generalize to systems written in a different language. However, the subject systems represent a range of sizes and domains. They are similar to systems developed in industry.

CHAPTER 4

CONCLUSION & FUTURE WORK

The structured nature of source code has often been overlooked when determining term importance in TR based feature location. In this thesis we proposed and evaluated the use of a proposed structural term weighting technique for an LDA based FLT.

4.1 Summary of Findings

We measured the performance of an LDA based FLT using sixteen different configuration using the effectiveness measure [Dit et al., 2013; Poshyvanyk et al., 2007]. For the five systems we studied — ArgoUML, Eclipse, JabRef, jEdit, and muCommander — our results indicate a trend of increased performance by weighting method name terms, but decreased performance when weighting method call terms. A configuration multiplying method terms eight times had statistically significant performance gains in three of the five systems. Thus, we conclude that structural term weighting has a significant impact on the accuracy of a feature location task, assisting with program comprehension and software maintenance.

4.2 Future Work

In future work we plan to evaluate the use of additional multipliers for method name terms, as well as the effect of removing some classes of terms by using a multiplier of zero. We also plan to consider terms originating from other structural classes, such as comments and string literals. In addition, we plan to analyze characteristics of corpora and to use the analysis results to define metrics for predicting whether structural term weighting is likely to improve the accuracy of a TR-

based FLT. Finally, we plan to investigate combinations of structural term weighting and common term weighting schemes from the NLP domain (e.g., tf-idf).

REFERENCES

Andrieu, C., N. Freitas, A. Doucet, and M. Jordan (2003). An introduction to MCMC for machine learning. *Machine Learning 50*, 5–43.

Biggers, L., C. Bocovich, R. Capshaw, B. Eddy, L. Etzkorn, and N. Kraft (2012). Configuring latent Dirichlet allocation based feature location. Technical report.

Biggers, L. and N. Kraft (2012). A comparison of stemming algorithms for text retrieval based feature location. Technical report, Department of Computer Science, The University of Alabama.

Blei, D., A. Ng, and M. Jordan (2003). Latent Dirichlet allocation. *J. of Machine Learning Research 3*, 993–1022.

Deerwester, S., S. Dumais, G. Furnas, T. Landauer, and R. Harshman (1990). Indexing by latent semantic analysis. *Journal of the American Society of Information Science 41*, 391–407.

Dit, B., L. Guerrouj, D. Poshyvanyk, and G. Antoniol (2011). Can better identifier splitting techniques help feature location? In *Proc. of 19th IEEE Int'l Conf. on Program Comprehension*, pp. 11–20.

Dit, B., M. Revelle, M. Gethers, and D. Poshyvanyk (2013). Feature location in source code: A taxonomy and survey. *Journal of Software: Evolution and Process 25*, 53–95.

Eisenberg, A. D. and K. De Volder (2005). Dynamic feature traces: Finding features in unfamiliar code. In *Proceedings of the 21st IEEE International Conference on Software Maintenance*, pp. 337–346.

Fox, C. (1992). Lexical analysis and stoplists. In W. Frakes and R. Baeza-Yates (Eds.), *Information Retrieval: Data Structures and Algorithms*. Prentice-Hall.

Gay, G., S. Haiduc, A. Marcus, and T. Menzies (2009). On the use of relevance feedback in IR-based concept location. In *Proc. of the IEEE Int'l Conf. on Software Maintenance*.

Griffiths, T. and M. Steyvers (2004). Finding scientific topics. *Proc. of the Natl. Academy of Sciences 101*(Suppl. 1).

Hill, E., L. Pollock, and K. Vijay-Shanker (2007). Exploring the neighborhood with Dora to expedite software maintenance. In *Proc. Int'l Conf. on Automated Software Eng.*

Hill, E., S. Rao, and A. Kak (2012). On the use of stemming for concern location and bug localization in java. In *Proceedings of the 12th IEEE International Working Conference on Source Code Analysis and Manipulation*.

Liu, D., A. Marcus, D. Poshyvanyk, and V. Rajlich (2007). Feature location via information retrieval based filtering of a single scenario execution trace. In *Proc. of the 22nd Int'l Conf. on Automated Software Engineering*, pp. 234–243.

Lukins, S., N. Kraft, and L. Etzkorn (2008). Source code retrieval for bug localization using latent Dirichlet allocation. *Proc. of the 15th Working Conf. on Reverse Engineering*.

Lukins, S., N. Kraft, and L. Etzkorn (2010, September). Bug localization using latent dirichlet allocation. *Information and Software Technology 52*(9), 972–990.

Marcus, A. and T. Menzies (2010). Software is data too. In *Proc. of the FSE/SDP Wksp. on Future of Software Engineering Research*, pp. 229–232.

Marcus, A., A. Sergeyev, V. Rajlich, and J. Maletic (2004). An information retrieval approach to concept location in source code. In *Proc. of the 11th Working Conf. on Reverse Engineering*, pp. 214–223.

Poshyvanyk, D., Y. Gueheneuc, A. Marcus, G. Antoniol, and V. Rajlich (2007, June). Feature location using probabilistic ranking of methods based on execution scenarios and information retrieval. *IEEE Transactions on Software Engineering 33*(6), 420–432.

Rajlich, V. and N. Wilde (2002). The role of concepts in program comprehension. In *Proc. of the 10th IEEE Int'l Wksp. on Program Comprehension*, pp. 271–278.

Ratanotayanon, S., H. Choi, and S. Sim (2010). My repository runneth over: An empirical study on diversifying data sources to improve feature search. In *Proc. of the 18th IEEE Int'l Conf. on Program Comprehension*.

Revelle, M., B. Dit, and D. Poshyvanyk (2010). Using data fusion and web mining to support feature location in software. In *Proc. of 18th IEEE Int'l Conf. on Program Comprehension*, pp. 14–23.

Salton, G. (1989). *Automatic text processing: The transformation, analysis and retrieval of information by computer*. Addison-Wesley.

Salton, G. and C. Buckley (1988). Term-weighting approaches in automatic text retrieval. *Information Processing & Management 24*(5), 513âĂŞ–523.

Scanniello, G. and A. Marcus (2011). Clustering support for static concept location in source code. In *Proc. of the 19th IEEE Int'l Confön Program Comprehension*.

Shao, P., T. Atkison, N. Kraft, and R. Smith (2012). Combining lexical and structural information for static bug localization. *Int'l Journal of Computer Applications in Technology*.

Voorhees, E. M. (1999). The trec-8 question answering track report. In *In Proceedings of TREC-8*, pp. 77–82.

Wang, X., L. Zhang, T. Xie, J. Anvik, and J. Sun (2008). An approach to detecting duplicate bug reports using natural language and execution information. In *Proc. of the 30th Int'l Conf. on Software Engineering*, pp. 461–470.

Wilson, A. and P. Chew (2010). Term weighting schemes for latent dirichlet allocation. In *Human Language Technologies: The 2010 Annual Conferencerence of the North American Chapter of the Association for Computational Linguistics*, Volume 10, pp. 465–473.

Zhao, W., L. Zhang, Y. Liu, J. Sun, and F. Yang (2006). SNIAFL: Towards a static noninteractive approach to feature location. *ACM Trans. Softw. Eng. Methodol. 15*(2).