2011

The accuracy of information retrieval based bug localization techniques

Matthew D. Beard

THE ACCURACY OF INFORMATION RETRIEVAL BASED BUG LOCALIZATION TECHNIQUES

by

MATTHEW D. BEARD

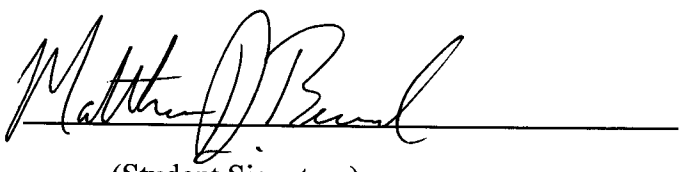
A THESIS

Submitted in partial fulfillment of the requirements
for the degree of Master of Science in Software Engineering in Computer Science
in
The Department of Computer Science
to
The School of Graduate Studies
of
The University of Alabama in Huntsville

HUNTSVILLE, ALABAMA

2011

In presenting this thesis in partial fulfillment of the requirements for a master's degree from The University of Alabama in Huntsville, I agree that the Library of this University shall make it freely available for inspection. I further agree that permission for extensive copying for scholarly purposes may be granted by my advisor or, in his/her absence, by the Chair of the Department or the Dean of the School of Graduate Studies. It is also understood that due recognition shall be given to me and to The University of Alabama in Huntsville in any scholarly use which may be made of any material in this thesis.



(Student Signature)

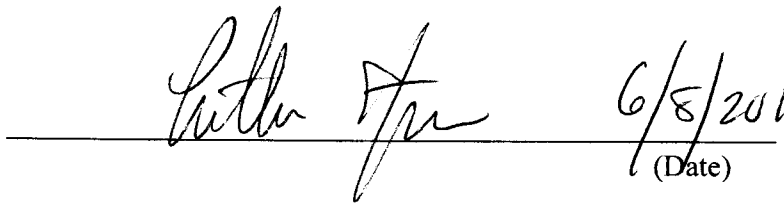
6/8/2011

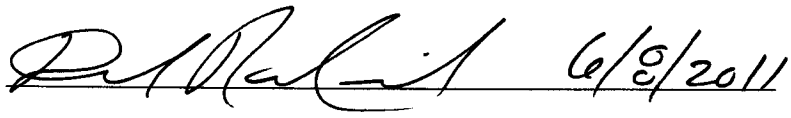
(Date)

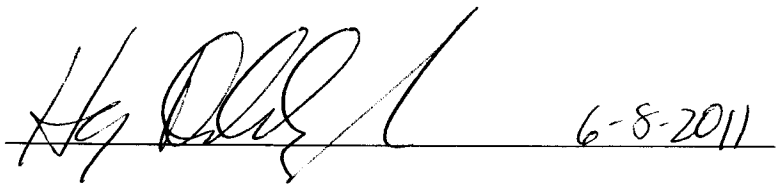
THESIS APPROVAL FORM

Submitted by Matthew D. Beard in partial fulfillment of the requirements for the degree of Master of Science in Software Engineering in Computer Science and accepted on behalf of the Faculty of the School of Graduate Studies by the thesis committee.

We, the undersigned members of the Graduate Faculty of The University of Alabama in Huntsville, certify that we have advised and/or supervised the candidate on the work described in this thesis. We further certify that we have reviewed the thesis manuscript and approve it in partial fulfillment of the requirements for the degree of Master of Science in Software Engineering in Computer Science.


6/8/2011 (Date)
Committee Chair


6/8/2011


6-8-2011

Heggere S. Ranganath 6-8-2011
Department Chair


6/8/11
College Dean

Rhonda Kay Hoede 8/5/11
Graduate Dean

ABSTRACT

The School of Graduate Studies

The University of Alabama in Huntsville

Degree Master of Science in Software Engineering

College/Dept. Science/Computer Science

Name of Candidate Matthew D. Beard

Title The Accuracy of Information Retrieval Based Bug Localization Techniques

Bug localization involves using information about a bug to locate affected code sections. Several automated bug localization techniques based on information retrieval (IR) models have been constructed in recent years. Latent semantic indexing and latent Dirichlet allocation are two such techniques.

The "gold standard" for measuring an IR technique's accuracy considers the technique's ability to locate a "first relevant method." However, the question remains – does finding this single method enable location of a complete set of bug related methods? Previous arguments assume this to be true; however, few analyses of this assumption have been performed. Our first task is to test this assumption.

To further measure IR accuracy, we analyze the relevancy of the IR model's "first method returned." We use various structural analysis techniques to extend relevant methods and determine accuracy and reliability of these assumptions. We also compare the accuracy of both IR techniques in this context.

Abstract Approval: Committee Chair _____ 6/8/2011
(Date)

Department Chair _____ 6-8-2011
(Date)

Graduate Dean _____ 8-5-11
(Date)

ACKNOWLEDGMENTS

I preface this section by saying how extremely blessed I am to have had the opportunity to write this thesis, to perform research under the conditions I have, and to merely further my education at the University of Alabama in Huntsville. I greatly appreciate the help and guidance of Dr. Letha Etzkorn and her willingness to employ me as a research assistant. Before meeting her, I had almost no idea as to what direction I wanted my higher education to take. She has helped shape my path and open numerous doors of opportunity which have enabled me to successfully complete my Master's and hopefully pursue my Ph.D. I appreciate her wisdom and guidance in this process – she is truly a great professor and advisor. I am also very grateful for my committee: Dr. Harry Delugach and Dr. Dan Rochowiak, whose insights have helped me to become a better researcher.

Also, thank you to Betty Nelson, Diane Cox, and Maryann Bierer for always being ready to aide me with whatever problems or questions I had about this process, and for always being willing to just chat.

I am extremely grateful to my parents for their continual guidance, reassurances, and words of wisdom throughout this process, my whole education, and, even more so, my entire life. Thank you for being such a blessing to me and for guiding me to Huntsville and helping me settle in here. Thank you to Kim for supporting and motivating me throughout this process, for putting up with my (occasional) complaining, and for being willing to proofread and sort through all of my "geek-speak."

Finally, and most importantly, thank you to God for the continual guidance and strength He provides, the blessings that only He can give, and the promise of hope for tomorrow, and eternity.

TABLE OF CONTENTS

Page

LIST OF FIGURES ..... **x**

LIST OF TABLES ..... **xi**

LIST OF ACRONYMS ..... **xii**

CHAPTER

1 INTRODUCTION **1**

2 BACKGROUND **4**

2.1 IR Models for Concept Location & Source Code Retrieval 4

2.2 Bug Localization 8

2.3 Call Graphs 9

2.4 Call Graphs and Structural Bug Localization 11

2.5 Directly Related Work 12

3 RESEARCH APPROACH: OUR STRUCTURAL BASED ANALYSIS OF IR ACCURACY **13**

3.1 IR Models 14

3.2 Relevant Methods 15

3.3 Structural Analysis 16

4 CASE STUDY RESULTS **18**

4.1 First Relevant Method 21

4.2 First Method Returned..... 24

4.3 LSI vs. LDA Accuracy..... 27

5 ISSUES AFFECTING OUR CASE STUDY 29

6 CONCLUSIONS AND FUTURE WORK..... 30

REFERENCES 32

LIST OF FIGURES

Figure		Page
2.1	Examples of call graphs	10
4.1	Success rate for locating partial sets of relevant methods using first relevant method (LDA and LSI) + structural techniques	22
4.2	Success rate for locating complete sets of relevant methods using first relevant method (LDA and LSI) + structural techniques	23
4.3	Accuracy of first method returned (FMR) by LSI + structural techniques (ST)	25
4.4	Accuracy of first method returned (FMR) by LDA + structural techniques (ST)	26
4.5	Rank of first relevant methods returned by LSI and LDA	28

LIST OF TABLES

Table Page

4.1 Finalized versions of queries for bugs in Rhino 1.5R5..... 20

LIST OF ACRONYMS

FRM	First Relevant Method
FMR	First Method Returned
IR	Information Retrieval
LDA	Latent Dirichlet Allocation
LSA	Latent Semantic Analysis
LSI	Latent Semantic Indexing
pLSI	Probabilistic Latent Semantic Indexing
RTC	Relational Topic-based Coupling
SVD	Singular Value Decomposition
VB	Variational Bayes

CHAPTER 1

INTRODUCTION

In software maintenance, a key task involves using the information about a bug to locate the sections of code in which the bug occurs. Once these sections are located, the developer can fix the coding error(s) that caused the bug. While simply looking through code and comments was once an adequate technique for bug localization, the size and complexity of software has grown exponentially in recent years, making automated techniques a near necessity for efficient, low-cost maintenance.

One of the more recent types of automated bug localization techniques involves the usage of information retrieval (IR) models to retrieve relevant information about the bug. Latent semantic indexing (LSI) is one of the primary IR techniques used in these analyses [18]. While LSI does provide useful results, another IR technique, latent Dirichlet allocation (LDA) [2], has shown promise as a more robust and thorough technique. LDA has previously been demonstrated to be a useful technique in performing various source code retrieval tasks, such as concept mining [15]. Furthermore, LDA has also shown promise as a useful technique specifically for bug localization [13].

When measuring any bug localization technique to determine accuracy and reliability, the commonly recognized "gold standard" for accuracy requires a technique to return a complete set of all code sections that are affected by the bug. In previous work, the use of IR techniques in bug localization has focused around a distinctly different "gold standard" for determining whether an IR based technique can be deemed accurate. Researchers in this

area base comparisons of their work by determining which technique yields results closest to this "gold standard." That standard is the ability of the IR technique to locate a "first relevant method" (FRM), that is, the first method returned that is actually relevant to the bug [18] [13] [12]. Though this existing gold standard certainly returns one method that is affected by the bug, the question still remains – does finding this single method actually allow one to find all parts of the code that are relevant to the bug? In previous research, the assumption has been made that once the first relevant method is located, structural characteristics of the source code can be used to lead one to any remaining code that must be fixed. However, to our knowledge, only one brief, superficial analysis of this assumption has been performed in the past [12].

A further area of interest in the analysis of results returned by IR techniques involves determining whether the "first method returned" (FMR) of a similarity query performed against an IR model is relevant to the bug. The first method returned is the very first method located by the query against the IR model. Since the first method returned may not always be directly relevant to a bug (i.e., containing faulty code), the question arises – can this technique be used to locate methods which are directly relevant to the bug? It is possible that, while not directly related, the IR technique has determined underlying semantic similarities between the first method returned and the bug itself. If this is true, then the use of structural characteristics in the source code to locate relevant methods that need to be fixed should be feasible.

In order to determine whether the previously described "gold standard" is a reliable assumption, in our research, we examine the reliability of the "first relevant method" gold standard, and separately we also examine the accuracy of the "first method returned"

assumption. To do this, we will start with each individual assumption, then analyze and examine the structure of the source code to determine whether the assumption plus the structural characteristics of the code do indeed lead to all (or part) of the sections of code that must be changed in order to fix the bug. One of the most relevant structural techniques for this case is call graphing, which maps the caller/callee relationships. Our structural analysis will focus on call graphs, as well as simple class and inheritance information, to determine whether the "first relevant method" and "first method returned" are reliable starting points for further structural analysis.

We are also interested in the accuracy of LSI and LDA in relation to their use in bug localization. Thus we examine the "first relevant method" and "first method returned" assumptions in relation to whether the "first relevant method" and "first method returned" were created using LSI or LDA.

CHAPTER 2

BACKGROUND

This section begins with a description of information retrieval techniques and their application to source code. This is followed with a discussion of existing bug localization techniques, including those that implement IR techniques. Structural techniques, specifically call graphs, are then examined.

2.1 IR Models for Concept Location & Source Code Retrieval

Some recent techniques of operating on and retrieving relevant information from source code use models of the code, as opposed to operating on the source itself. These techniques require source to be represented as a corpus, or collection of documents. Each document consists of a collection of terms, or words. Words within documents are often filtered to remove irrelevant, or "stop," words and stemmed using a stemming algorithm prior to their insertion into the corpus. The corpus itself is constructed from a variety of semantic information that is embedded in the code, depending on the application of the IR technique. Some corpora may contain words which consist only of identifiers, while others may also incorporate comments and other relevant information. The point of division for documents that make up the corpus is also determined by the application of the IR technique. Some divisions may be made according to characteristics of the source itself, such as by file, method, class, or project. Other applications of IR may divide source code by other criteria, such as by code clone class [22].

In order to retrieve user-requested results from an IR model, a query must be performed upon the corpus. User queries can be structured in the format of a document that is then entered into the corpus, or can be compared using a similarity measure, such as cosine similarity, after the IR technique is performed on the corpus. Results of a query are a ranked set of documents where a document's rank is given by the similarity between the document and the user query. The divisor used to separate documents will determine the specificity of results that are returned when a query is performed. While several IR models exist, two that have received a great deal of attention in the field of source code retrieval are latent semantic indexing and latent Dirichlet allocation.

LSI applies latent semantic analysis (LSA) to document indexing and retrieval [5]. A great deal of work has been performed in recent years using LSI to perform source code retrieval [14] [17] [18]. LSI is based on previous vector-based representations of the occurrences of words within documents, but extends the vector-based approach by using singular value decomposition (SVD). The original vector-based representations use vectors to represent terms in text documents and relationships between documents and terms in a collection as a term-document matrix. The matrix is structured to represent term vectors as rows, with each row giving a term's relationship to each document, and document vectors as columns, with each column giving a document's relationship to each term. Since each term's individual relation to each document is given, term-document matrices can end up being very extensive for large corpora. To reduce size, LSI employs SVD on the term-document matrix. Since dimensions in a vector space computed by SVD are ordered from most to least significant, the reduced representation will be the best possible for the dimensionality. A set of the top dimensions is then extracted (the count depending on the

application) and used for analysis. The rest (which may be tens, hundreds, thousands, or more) are discarded. This discarding process helps to eliminate random associations and noise that could produce errant or problematic results. The reduced term-document matrix can then be analyzed by associating documents together through calculating the cosine of the angle between the vectors corresponding to those documents. A user query can be performed by transforming the query into a document in the corpus and then analyzing the similarity of the query document with the rest of the corpus.

While LSI has been proven as an effective IR technique for a variety of applications, it is not without weakness. Since the results of LSI are expressed using a spatial representation that is numeric, the interpretation of the results can be difficult. Furthermore, the focus of LSI is to represent terms that exhibit *synonymy*, words that have similar meaning. LSI falls short in representing terms exhibiting *polysemy*, words that can have multiple meanings. To counter these and other weaknesses of LSI, probabilistic latent semantic indexing (pLSI) was introduced by Hofman. pLSI is a generative topic model that uses statistical techniques and models, the core being the aspect model [10]. In this technique, each term in a document is modeled as a multinomial random variable which can be interpreted as a topic, and each document is represented as a probability distribution on a fixed set of topics [2].

Studies comparing pLSI to LSI have shown that the former contains many advantages over the latter [2] [10]. However, pLSI also introduces several new issues. At its core, pLSI provides a basis for the probabilistic modeling of text; however, it does not provide a probabilistic model at the document level. Each document is only represented as a list of numbers, and there is no generative probabilistic model for those numbers. This means that the number of parameters in the model grows linearly as the corpus grows, which can lead

to problems with overfitting. In addition, pLSI cannot predict topic distributions for documents added to the collection after the initial estimation has been performed on the training set. Topics must be chosen from the topic distribution that was generated from the training set, so the performance of pLSI degrades when new documents are added.

To counter these weaknesses in the LSI family, Blei, Ng, and Jordan describe the latent Dirichlet allocation (LDA) generative probabilistic model, which extracts latent, or hidden, topics present in a document collection [2]. LDA models each item of a collection as a finite mixture over an underlying set of topics and each topic as an infinite mixture over an underlying set of topic probabilities. Each topic in the set is a probability distribution over the terms that make up the vocabulary of the corpus. The "theme" of a topic can then be determined by analyzing the terms that have the highest probability in relation to that topic.

All three of the IR techniques described here can handle synonymy well, but pLSI and LDA are both capable of handling polysemy much better than pure LSI. LDA also improves upon the shortcomings of pLSI by handling document level probabilistic modeling in a different way. The generative model in LDA is much more well-defined than pLSI's, so LDA can easily generalize to new documents. Finally, LDA overcomes the overfitting problem present in pLSI by not requiring new documents to fit into topic distributions that are present in one or more training documents.

Similarity computations with LDA can be performed using a variety of techniques. One such approach involves measuring the similarity between topic distributions which correspond to the query and to each of the documents in the corpus. Another approach models LDA as a probabilistic query to the topic model, in which the most relevant

documents maximize the query's conditional probability, given each document [21]. LDA similarity queries can also be performed using a variation of cosine similarity measure [19].

Although LDA has clear advantages over both LSI and pLSI, directly computing the probabilistic model for LDA is fractious, so it is necessary to use techniques for approximation. A variety of techniques exist for estimating LDA, including Gibbs sampling [21], expectation propagation [16], and variational methods [2], such as Variational Bayesian analysis [9].

2.2 Bug Localization

Techniques for bug localization may generally be classified as one of two types: static or dynamic. Dynamic bug localization techniques examine code at runtime, contrasting the behavior of correct and incorrect executions to isolate program elements that may contain bugs [11]. These techniques usually assume no prior knowledge of the semantics within the program code. On the other hand, static analysis techniques detect bugs by checking well-specified program models or the code itself [11]. In the context of this research, implementing IR techniques in automated bug localization must fall into the category of static techniques, as we are working with static source code. IR techniques such as we use normally could not be applied to code at runtime, although these kinds of techniques could be applied to static traces that were previously created at runtime.

IR-based static techniques aim to locate areas of source code that need to be modified in order to correct a bug. These techniques do not, and generally cannot, identify all locations where fixes must be implemented to correct a bug, but they can identify a relevant source entity or method where analysis can begin. Once a starting point is located, more in-depth

analysis can be performed to determine all affected areas. Since it is not necessary to locate all areas of the source code that are related to the bug, standard schemes of measuring the accuracy of IR techniques are not useful [8]. What in the past has been considered the best measure or "gold standard" of accuracy is to determine which IR-based static technique ranks the first relevant source entity the highest [18]. Since one relevant entity is all that is needed to begin examination, the ranking of that entity determines the amount of information a programmer must examine before reaching an entity that is relevant to the bug.

2.3 Call Graphs

Structural techniques for analyzing and navigating code involve an entirely different set of methodologies than those discussed so far. While semantic techniques focus on meanings and topics, structural techniques focus on the way a set of source code or a software project is structured syntactically, as well as the control and data flow represented therein. A variety of properties exist within code that can be analyzed to provide further understanding of the code's structure, including flow of control, flow of data, inheritance, control dependencies, and caller/callee relationships. In navigating through the layout of a program and determining the program's potential behavior at runtime, in-depth information regarding caller/callee relationships is necessary. To display such information in a format that is easily comprehendible, a call graph is used.

A call graph is a directed multigraph in which the nodes represent the methods in a program and the edges represent the potential calls between those methods [1]. Figure 2.1 illustrates two simple call graphs that involve functions *f* and *g*. In Figure 2.1(a), function *f*

calls function *g*. In Figure 2.1(b), function *f* calls *g* twice. Figure 2.1(c) depicts a more complex call graph that includes several methods, each of which is called multiple times. Though call graph visualizations appear to be very descriptive and well organized, such descriptions can become complex and difficult to read in the case of large software projects that have many methods and method calls. Because of this, call graphs are typically represented textually.

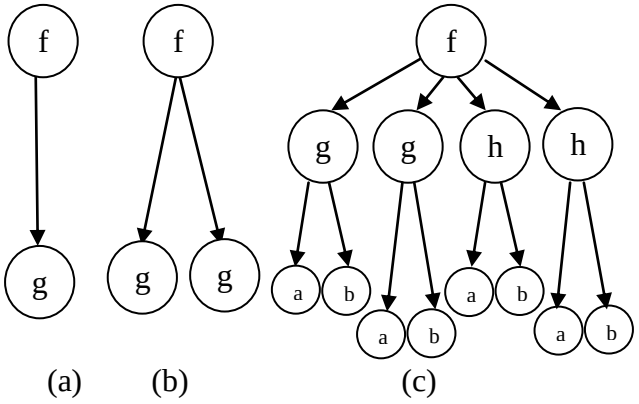


Figure 2.1: Examples of call graphs

Call graphs are usually created from one of two methods. One such method involves executing the program in which the caller/callee relationship is to be analyzed, and then monitoring the execution of the program to determine the methods that are called, as well as the frequency of calling and the identity of the caller. This dynamic method is extremely accurate for a single run of a program, as it captures the precise conditions and results that occur during that execution; however, this method cannot capture all possible structures of the caller/callee relationship. A second method involves using a parser to analyze the

software project without actually executing the program. This is a static call graph generation method, and is intended to capture and represent the behavior of every possible execution of a program. The number of calls to a function is always static when using static call graphs.

2.4 Call Graphs and Structural Bug Localization

The mapping of programming structure that results from creating a call graph of a program is an obvious place where one could begin in analyzing a program to determine whether a program has bugs. One such technique compares dynamically generated call graphs in correct and faulty executions and extracts the most discriminative sub-graphs which contrast the program flow between the two execution types. The differences in the extracted sub-graphs can pinpoint the bug and also provide information regarding the context of the bug [3].

Bug localization has also been performed using an edge-weighted adaptation of the standard call graph model. Such call graphs use edge-weights to denote counts of multiple calls to a single method. These compressed graphs can then be mined and the results of correct and faulty executions compared to locate the bug. The analysis of these call graphs combines search techniques by seeking errors that are structure affecting, using existing techniques, and call frequency affecting, using the edge-weighted call graphs [6].

In addition to the usage of call graphs, the use of other types of coupling information can improve the accuracy of bug localization. For example, recent work by Gethers and Poshyvanyk [7] uses information retrieval techniques in combination with structural analysis to measure coupling between classes. This new metric, Relational Topic based

Coupling (RTC), aims to improve on previous coupling measures, and can also assist in feature location.

2.5 Directly Related Work

As stated in 2.1, a number of analyses have been performed on source code in order to localize bugs using IR techniques. However, only a few, brief studies on this specific topic – the accuracy of IR based bug localization using structural techniques – have been performed. Specifically, in the past no researchers have thoroughly addressed the degree to which the commonly accepted "gold standard" of FRM actually enables location of the entire bug. A limited look at LDA's accuracy in this context is performed in [12]. As stated in 2.4, [7] considers coupling metrics in light of LDA, but structural techniques are not used to provide an accuracy measure of LDA, or of the completeness of the first relevant method assumption. To our knowledge, no other studies examine IR techniques in this context.

CHAPTER 3

RESEARCH APPROACH:

OUR STRUCTURAL BASED ANALYSIS OF IR ACCURACY

The focus of this research is to provide a clear picture of the accuracy of information retrieval based techniques for bug localization using two IR techniques, latent semantic indexing and latent Dirichlet allocation. Call graphs are then used as the analysis technique to determine each IR technique's accuracy. The goals of this analysis are as follows:

- Determine whether the "first relevant method" is a reliable place to begin IR-based bug localization to test the "gold standard" assumption.
- Determine whether the "first method returned" is a worthwhile place to begin IR-based bug localization.
- In addition, compare LSI vs. LDA's accuracy in the context of bug localization.

To perform this analysis, LDA and LSI models of the source code must both be built. The models are then queried using descriptions of selected bugs, returning ranked results of the queries. We use these results to derive two different sets of potentially relevant methods – the first relevant method and the first method returned. These two sets are compared against a set of methods mapped to corresponding bugs. The comparison is performed using information provided by (1) call graphs, (2) local classes, and (3) class inheritance.

3.1 IR Models

In order to construct the necessary IR models, a two-step process is required. First, the source code must be converted into a collection of documents that can then be understood by the selected IR technique. Second, the IR model is constructed, using either LSI or LDA.

Constructing the document collection: Semantic information must be extracted from the project as a whole. This process involves separating source code by the desired distinction (in our case, by methods), selecting individual pieces of semantic information for extraction (comments and identifiers), and then performing preprocessing on this information. The preprocessing phase involves removal of stop words (irrelevant, semantically insignificant words) and Porter stemming of words. This process is performed using the NetBeans plug-in outlined in [13]. This plug-in produces a single text file containing the corpus, consisting of documents separated by lines and words by spaces.

IR Model Construction and Querying: To construct and query the IR models, we use Gensim, a Python based framework for Vector Space Modeling [19]. This framework includes techniques for performing both latent semantic indexing and latent Dirichlet allocation.

Latent semantic indexing is performed by Gensim using the standard technique of singular value decomposition. To form the LSI model, Gensim requires a corpus as input (the output of the plug-in), as well as a dimensionality measure for clustering of documents. The output of this process is a model that can then be queried using standard cosine similarity. Queries are then converted to their corresponding dictionary representations and

run against the corpus. The results returned are a ranked set of document numbers that represent methods.

Latent Dirichlet allocation is performed by Gensim using an online variational Bayes (VB) algorithm [9]. To form the LDA model, Gensim requires the corpus, a dimensionality measure, and an iteration count. LDA itself is intractable for direct computation [2], so probabilistic approximation techniques must be employed to derive results. Since VB techniques perform probabilistic sampling, the model is re-sampled for a user-defined number of iterations in order to achieve optimal results. In this research, querying is performed using the same similarity technique as described for LSI, rendering results that are interpreted to correspond with ranked methods. The output of this process is a model formatted similarly to the output discussed previously in LSI.

3.2 Relevant Methods

From the results returned by the IR technique, we derive two sets of data: first relevant method returned and first method returned.

The first relevant method returned is the first method located by the query against the IR model that contains part of the bug. This method is sometimes the only method needed to fix a bug. In this case, structural techniques are not needed to verify the accuracy of the IR technique. However, it is often the case that multiple methods need to be fixed in order to completely resolve a bug. In this instance, structural analysis techniques can be used to determine whether this relevant method is a good starting point to find other methods that are in need of alteration to fix a bug.

The first method returned is the very first method located by the query against the IR model. This method may or may not be directly relevant to the bug in question. Our structural analysis attempts to determine whether these returned methods have any relation to the bug, regardless of whether they are a "guilty party" in the mapped bug.

3.3 Structural Analysis

To perform structural analysis, we use data gathered using STI's Understand for Java [23], as well as Doxygen [4].

Call Graph Analysis: Call graph results are derived using manual comparison of the IR returned method with methods that are callers/callees of that method. The call graph tree is traversed using caller and callee chains of length four (i.e., we look at methods that called or were called by the IR returned method four levels up or down the call graph). From a pre-study analysis, we determined that, because of the large amount of work involved in the manual comparisons to perform call graph tracing, as well as a decaying level relevancy between caller/callee relationships as chain length increases, we would impose a chain length limit of four on this analysis.

Local Class/Inheritance Analysis: Local class results are derived by looking at the class that the IR returned method is in. This class is then manually examined to determine whether it contains any other methods that are affected by the bug. If so, the returned method bears some relation to the bug in question. Inheritance results are derived in a similar way – the class of the returned method is analyzed to determine whether any of its parent classes contain bugs. All methods in the parent class are examined.

Our structural analysis methodology consists of employing all three techniques, beginning with call graphs, followed by local classes, and concluding with inheritance. We begin by looking for all possible relevant methods that can be located by traversing the call graph tree. After results from the call graph are exhausted, we survey the local class to locate any additional relevant methods. Once all local class methods have been viewed, we look at class inheritance to determine whether any parent classes of the IR returned method contain relevant methods.

CHAPTER 4

CASE STUDY RESULTS

To assess the results of the IR techniques, our case studies were performed on the Rhino software system, an open source implementation of JavaScript written in Java [20]. Version 1.5 release 5 (1.5R5) was chosen for analysis, as this release has a large number of documented bugs (35 total). Each of the bugs used in our case studies was mapped by hand to all respective affected methods. This means that, for the whole set of bugs, we know the methods that need to be altered to fix each bug – in essence, the "solution" to each problem. Therefore, we can compare the results of our IR techniques to a complete solution set of bug affected functions to determine accuracy and reliability. All bugs used in this study meet the following criteria:

- Bugs existed in source files of Rhino 1.5R5 that implemented core and compiler functionality.
- Bugs required method-level fixes.
- Bugs were fixed in the next release versions of Rhino (either 1.5R5.1 or 1.6R1)
- Bugs were categorized as *resolved* or *verified* and as *fixed*, so only bugs that could be verified and repaired are used.

Included bugs span a wide range of severity levels, from critical defects to program enhancements.

We used three different measures to determine various aspects of the accuracy and reliability of both IR techniques, LSI and LDA. The first two of these measures involve the

structural analysis techniques mentioned in 3.3. The third looks directly at the effectiveness of LSI and LDA with respect to one another.

For each of these studies, we use the methodology outlined in 3.1 to perform our IR analysis. The corpus for both IR techniques consisted of 1,780 documents, each representing a unique method. To determine the best dimensionality for LSI and LDA, as well as the best iteration count for LDA, we ran both techniques using a variety of settings. For LSI, it was determined that a dimensionality measure of 25 clusters was the most effective at returning the largest number of highly ranked methods. For LDA, a dimensionality measure of 75 topics along with a sampling count of 200 iterations was best.

Queries for each of the cases follow the same methodology used in [13]. These queries are formulated using information extracted from the bug title and description contained within bug reports in the Rhino bug repository. Each query contains keywords manually extracted from the bug title, relevant keywords found in the bug report summary, useful variants of keywords (e.g., adding *parse* in addition to *parser*), common abbreviations (e.g., *eol* for *end-of-line*), and the addition of any words that may be related to the bug (e.g., *day* in addition to *daylight savings time*).

Table 4.1: Finalized versions of queries for bugs in Rhino 1.5R5

Bug No.	Bug Query
58118	daylight savings time year offset timezone day
238699	context compile function instantiation exception
238823	context compile function null exception
239068	constructor functions initialized
244014	code complexity limits interpreter
244492	java script run time exception
245882	java importer constructor
249471	string index out exception native global js parse float bound char
252122	double error message
253323	decompiler parser parse
254778	label statement
254915	this name object with
255549	jvm ambiguity calling java methods argument constructor overload
255595	class context call default enter java run time thread
256318	scriptable object equivalent values
256339	stack interpreter
256389	compiler environs xml
256575	end line semicolon eol semi colon
256621	throw statement eol
256836	dynamic functions
256865	gcj byte code constants
257128	interpreter tail call java js_stack
257423	optimizer this name code
258144	runtime class generated jsc run main script
258183	catch condition rethrow exception
258207	exception name dont delete ecma script catch object obj
258417	array index out bounds exception regexp native reg exp stack size state data
258419	copy paste regexp native reg exp re data back track stack state
258958	lookup objects scriptable output stream prototypes parents
258959	scriptable input stream context class loader resolve
261278	strict mode missing var declarations
262447	scriptable object get property ids
263978	run line number negative execute error
266418	serialize regexp reg exp RE compile char set
274996	interpreters stack array index bounds exception java wrapped

4.1 First Relevant Method

The goal of this case study is to determine whether the "gold standard" assumption regarding the first relevant method (see Chapter 1) is reliable. For this study, we ran each of the queries listed in Table 1 against the corpus using LDA and LSI and used the ranked results to obtain the first relevant method returned. The first 500 returned methods (out of 1,780 total) were analyzed to determine whether a relevant method was present. Our sample size was limited to 500 because correlations between the query and methods deeper than this point generally become too low to be considered reasonable. Out of the 35 bug queries that were performed, LDA found a relevant method for 32, while LSI located 31 relevant methods.

With these results, call graph, class, and inheritance based analyses were then performed on the sets of relevant methods using the method data for each bug. To demonstrate the overall effectiveness of the first relevant method returned by IR techniques in conjunction with structural techniques, we combined the results from LDA and LSI for this analysis.

Our results show that FRMs, when combined with structural techniques, are very effective starting points for locating further relevant methods. As illustrated in Figure 4.1, the combination of IR based and structural analysis techniques allows location of at least a partially complete group of relevant methods for 95% (60/63) of the bugs. Results returned solely by the IR technique could only locate the complete bug affected method set 39% (24/63) of the time. The addition of combined call graph and local class analysis (inheritance did not provide useful additions in this case) on the FRM allows added relevant method information to be gathered for an additional 56% of the bug set.

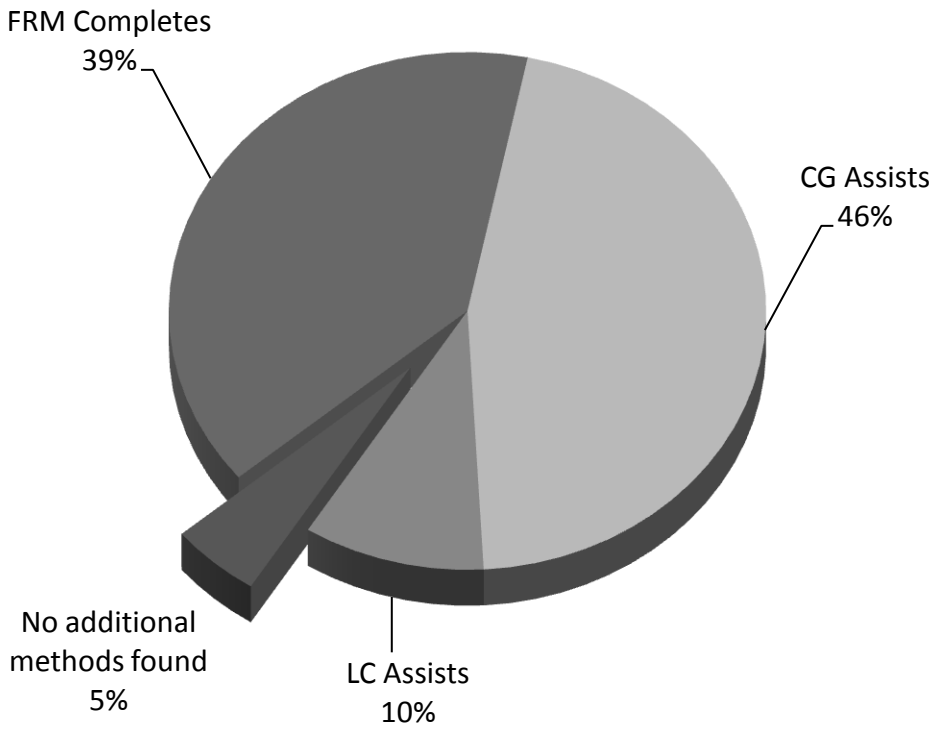


Figure 4.1: Success rate for locating partial sets of relevant methods using first relevant method (LDA and LSI) + structural techniques

When we examined the "gold standard" accuracy measure, which assumes that all relevant methods can be located using FRM, our results show a lower degree of success. As illustrated in Figure 4.2, the combined FRM, call graph, and local class information are able to assist in locating the complete set of all methods affected by a bug for 59% (37/63) of the bugs in our sample set. These results demonstrate that the FRM "gold standard" is effective, but only for slightly more than half of the bugs in this project.

The results of this analysis show that the FRM can be a good starting point for partial bug localization when combined with structural analysis techniques. However, in the context of our "gold standard," we can only conclude that this standard is capable of yielding a complete set of bug affected methods for approximately two-thirds of bugs in Rhino. To clarify, using FRM with the structural analysis techniques we employed in this study will generally allow us to locate *some* of the places in the code that must be changed in order to fix the bug, but not *all* the places that must be changed.

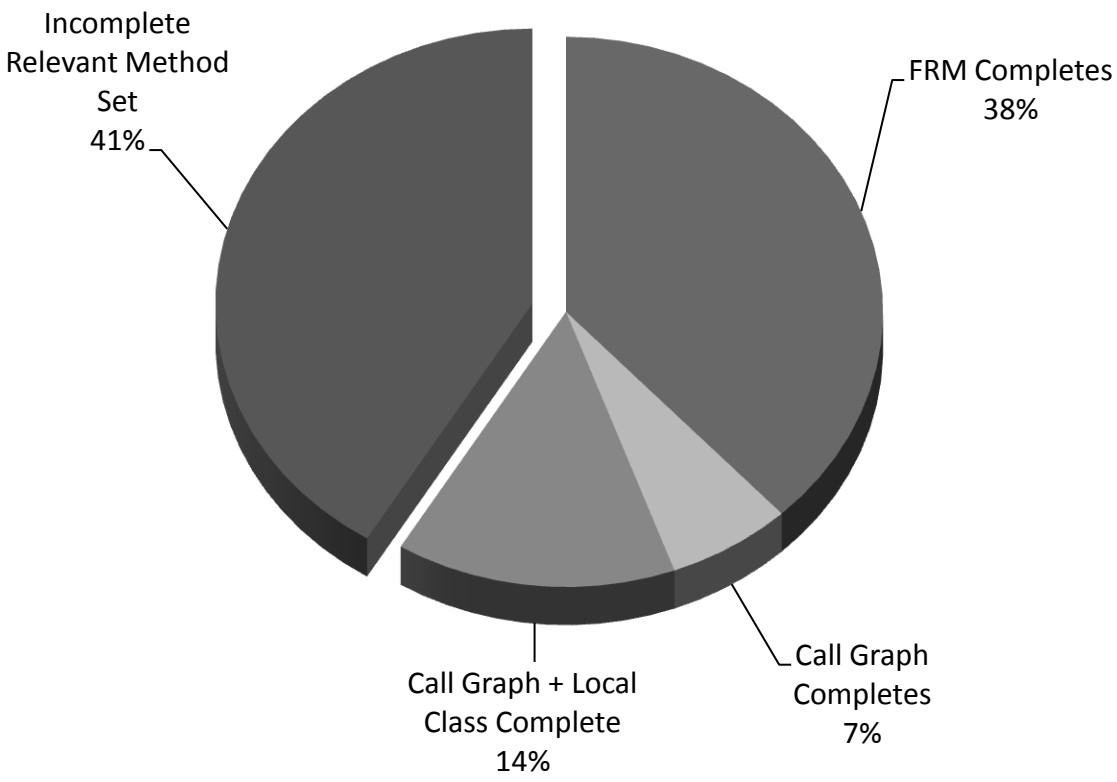


Figure 4.2: Success rate for locating complete sets of relevant methods using first relevant method (LDA and LSI) + structural techniques

4.2 First Method Returned

For the second study, we follow a process similar to that of the first study, involving running queries using both IR techniques. However, instead of looking through the returned results to find the FRM, we instead gather the method information for only the very first method returned, the method that, according to the IR technique, bears the highest similarity to the query. Because we gather the first method that is returned in this analysis, each bug is associated with a returned method, so our sample size is 35 bugs for both LDA and LSI.

We perform the same structural analysis techniques as discussed in Section 4.1 on this set of data. However, since this case study focuses on the accuracy of each IR technique's FMR, rather than the accuracy of the FRM "gold standard," we divide our results visualization to show the differences between results returned by LDA and LSI.

In order to measure accuracy, we use the data provided by structural analysis to determine whether the FMR can be used to reach methods related to the bug, regardless of whether the FMR itself is a directly relevant method (i.e., one that contains part of the bug). If the FMR does not contain the bug, but can still be used to reach all or part of the bug, then the FMR is indirectly relevant to the bug.

Our results show that the combination of structural techniques with both IR techniques provides significant extension to the FMR alone. As Figure 4.3 denotes, LSI's FMR was solely sufficient to locate relevant methods that correspond with 8% (3/35) of the sample set of bugs. However, the addition of all three structural techniques (CG, CG+LC, all) increases the number of bugs for which a relevant method can be located to 54% (19/35).

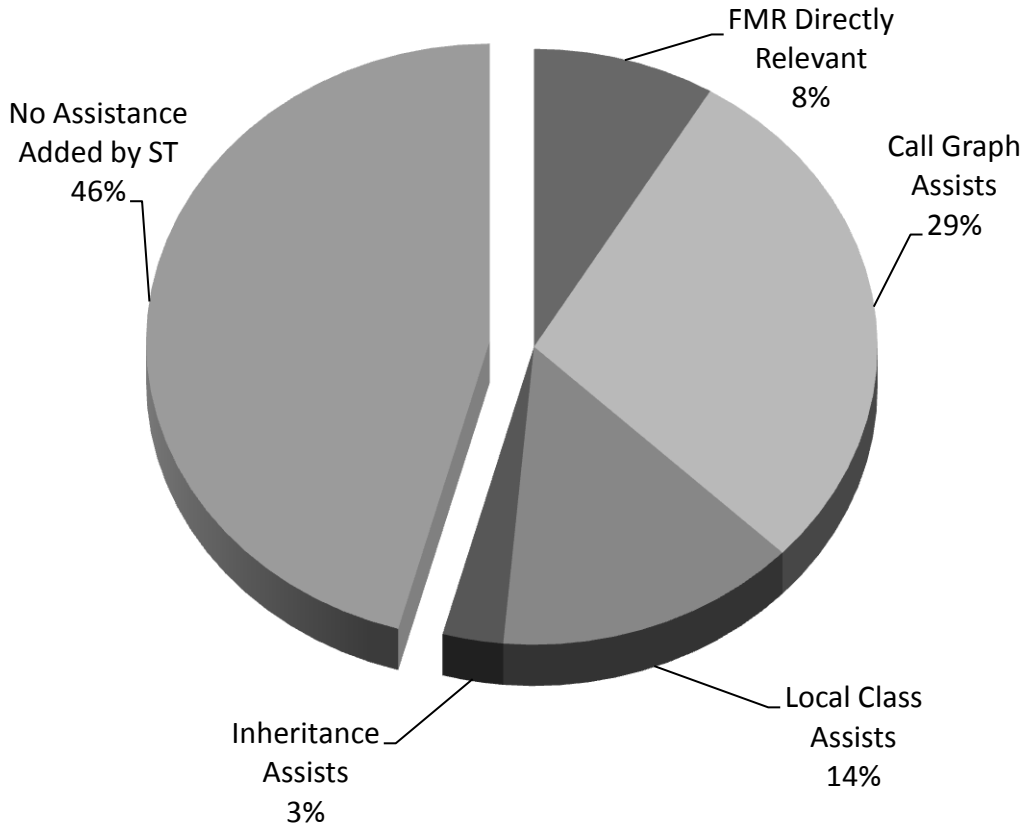


Figure 4.3: Accuracy of first method returned (FMR) by LSI + structural techniques (ST)

Results for LDA are similar. As Figure 4.4 denotes, LDA's FMR is solely sufficient to locate a relevant method for 3% (1/25) of the bug sample set. However, adding all three structural techniques increases the number of bugs for which a relevant method can be located to 63% (22/35).

Comparing the two techniques shows us that LSI's FMR, when considered alone, is accurate slightly more often than the FMR from LDA. However, FMRs returned by LDA seem to lend themselves better to extension by structural techniques. Specifically, the

addition of local class information to the structural analysis process for LDA returns more than twice as many relevant methods than the equivalent addition in the LSI structural analysis process (LDA 29% vs. LSI 14%). Such a result may indicate that LDA possesses certain advantages in examining and evaluating semantic information relating to the class level.

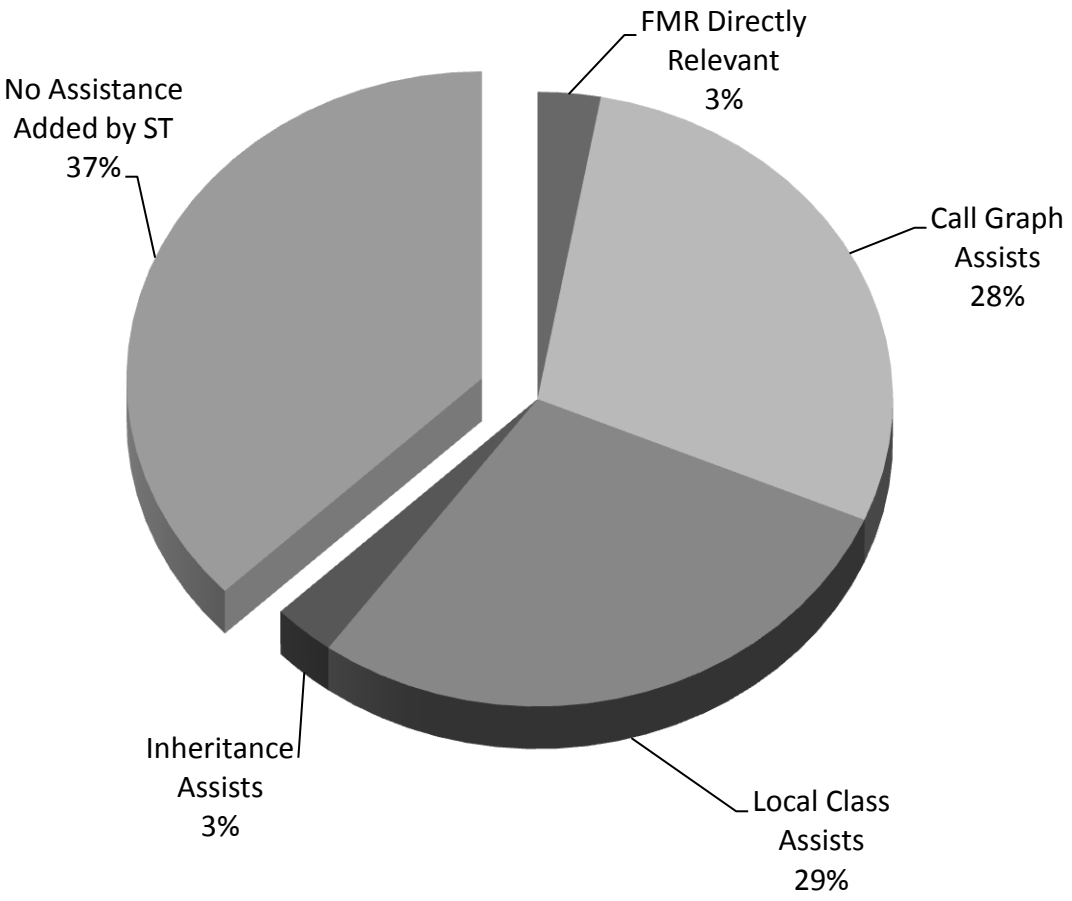


Figure 4.4: Accuracy of first method returned (FMR) by LDA + structural techniques (ST)

Combining IR and structural techniques also achieves some success in locating sets containing all methods affected by a bug. LSI plus structural techniques is capable of locating all affected methods for 25% (9/35) of the bug set. LDA performed similarly in this area, also locating all affected methods for 25% of the bug set.

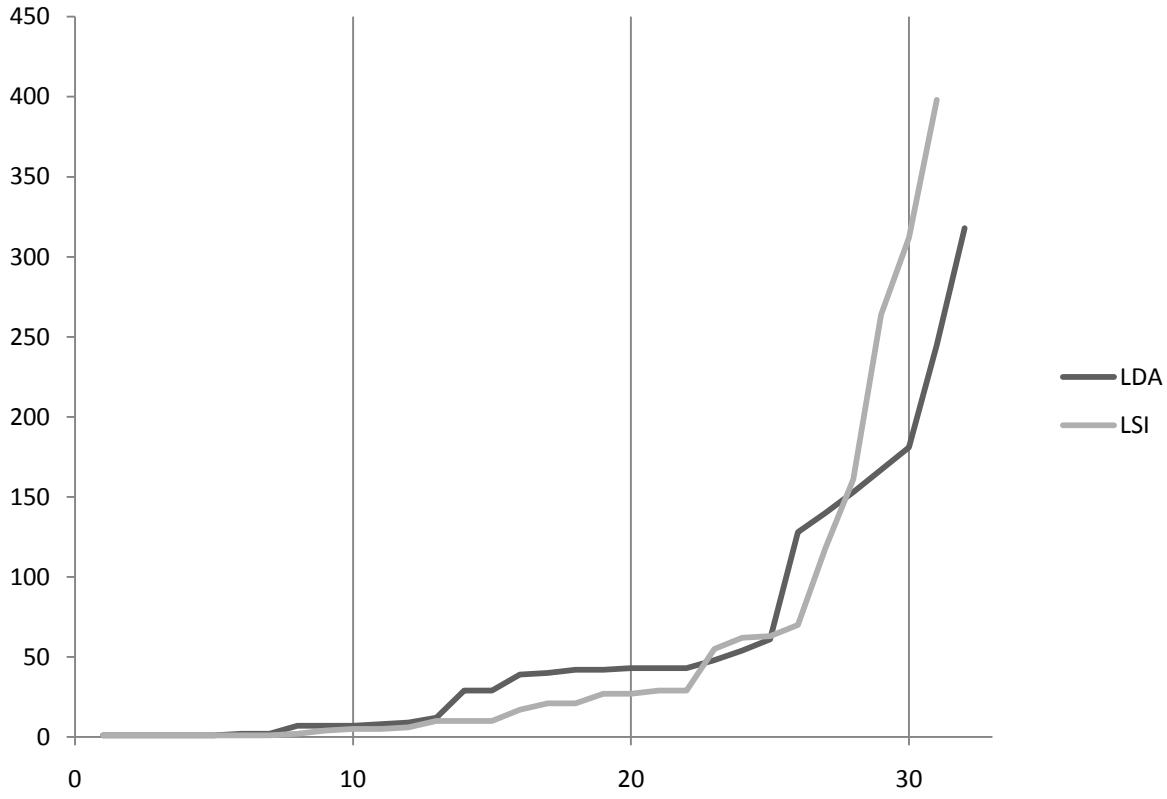
The results of this analysis show that FMR can be an effective starting point for locating methods affected by a bug.

4.3 LSI vs. LDA Accuracy

For our third study, we attempt to determine if any significant differences in the accuracy of either of these IR techniques exists in our results. To perform such an analysis, we look at the results from Section 4.1 regarding the FRM. In this section, we observe that LDA marginally outperforms LSI in locating FRMs. LDA was able to locate 32 FRMs by searching through the 500 methods that exhibit the greatest similarity to each of our bug queries. LSI locates 31 FRMs in this same process. However, while observation of the total number of FRMs located for our bug set is relevant, the rank of each FRM within the top 500 methods is also a relevant measure of accuracy. For instance, if one of our IR techniques returns the relevant method for a bug as the 10th most similar to the query, and the other technique returns the same relevant method as the 2nd most similar, then we can conclude that the second IR technique (LSI) is more accurate, as it ranked the relevant method higher with respect to our query.

Our results from this analysis show that the rankings of FRMs returned by LSI are slightly higher than those returned by LDA. Figure 4.5 charts the rank of all FRMs returned by each IR technique. We observe that the first 1/3 of FRMs are ranked nearly evenly by

both IR techniques, but LSI slightly outperforms LDA on the number of FRMs ranked 1 (7 vs. 5). A notable area of the results deals with FRMs with rankings of 10-50. For this ranking area, LSI's rankings of the FRMs are, on average, approximately 15 methods higher than comparable FRMs returned by LDA. We can conclude from these results that rankings between the two IR techniques are very close, with LSI showing slightly higher accuracy than LDA.



.

Figure 4.5: Rank of first relevant methods returned by LSI and LDA

CHAPTER 5

ISSUES AFFECTING OUR CASE STUDY

There a several issues with our analysis that could affect the quality of results returned. As stated in [13], the quality of results returned by any semantic-based technique is highly dependent on the amount and quality of semantic information found in both the source code, as well as queries of the source. Code with poorly named identifiers or sparse commenting is much more likely to yield poor semantic results. Bugs that are described incorrectly or poorly are also more likely to return methods with little to no correlation to the bug.

CHAPTER 6

CONCLUSIONS AND FUTURE WORK

The studies presented in this paper demonstrate that both the first relevant method and the first method returned can be used to assist in bug localization. However, we can conclude that the FRM "gold standard" is not a highly reliable assumption. As we have found, extending the FRM using structural techniques is only capable of locating a full set of relevant methods 60% of the time. Therefore, following the "gold standard" technique only produces the assumed results for slightly less than two-thirds of the bugs in the Rhino software. Even though this is the case, a comparison between the FRM and the FMR for bug localization shows that the FRM is a much more effective starting point for searching for relevant methods.

Our studies also demonstrate that LDA and LSI both have distinct advantages in our context. While LSI appears to be better at returning higher ranks for FRMs, LDA seems to provide better results that relate to underlying semantics, a useful feature in examining FMRs. Continued exploration into the accuracy of LSI and LDA with respect to bug localization is a definite area of interest.

In future studies, we plan to examine and incorporate the metric described in [7], as well as a variety of coupling metrics, to derive more information about the IR techniques. We also plan to survey various techniques of querying LDA corpora. While the similarity technique we use in the paper is acceptable, others [13] [21] use or suggest different techniques for querying that might demonstrate higher degrees of accuracy. A

comprehensive study of these query techniques is necessary to fully explore the true effectiveness of LDA in the context of concept location and bug localization. We also plan to incorporate a wider array of vector space modeling techniques to determine if other IR techniques, or combinations of techniques, might yield more accurate results.

REFERENCES

[1] D.F. Bacon and P.F. Sweeney, "Fast Static Analysis of C++ Virtual Function Calls," in *ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, San Jose, California, 1996, pp. 324-341.

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

[3] H. Cheng, D. Lo, Y. Zhou, X. Wang, and X. Yan, "Identifying Bug Signatures Using Discriminative Graph Mining," in *Pro. of the 18th Int'l. Symp. on Software Testing & Analysis (ISSTA)*, 2009, pp. 141-152.

[4] Doxygen. [Online]. http://www.doxygen.org/

[5] S.T. Dumais, "LSA and Information Retrieval: Getting Back to Basics," in *Handbook of Latent Semantic Analysis*: Lawrence Erlbaum Associates, 2007, pp. 293-321.

[6] F. Eichinger, K. Böhm, and M. Huber, "Mining Edge-Weighted Call Graphs to Localise Software Bugs," in *Proc. of the 2008 Euro. Conf. on Machine Learning and Knowledge Discovery in Databases*, 2008, pp. 333-348.

[7] M. Gethers and D. Poshyvanyk, "Using Relational Topic Models to Capture Coupling among Classes in Object-Oriented Software Systems," in *Proc. of 26th IEEE International Conference on Software Maintenance (ICSM 2010)*, Timişoara, Romania, 2010.

[8] D.A. Grossman and O. Frieder, *Information Retrieval: Algorithms and Heuristics*: Springer, 2004.

[9] M. Hoffman, D. Blei, and F. Bach, "Online Learning for Latent Dirichlet Allocation," in *Neural Information Processing Systems (NIPS 2010)*, Vancouver, 2010.

[10] T. Hofmann, "Probabilistic Latent Semantic Indexing," in *ACM SIGIR Int'l Conf. on Research and Development in Information Retrieval*, 1999, pp. 50-57.

[11] 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. 289-295, 2005.

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

[13] S.K. Lukins, N.A. Kraft, and L.H. Etzkorn, "Source Code Retrieval for Bug Localization using Latent Dirichlet Allocation," in *Proc. 15th Working Conf. Reverse Engineering (WCRE 2008)*, 2008, pp. 155-167.

[14] A. Marcus, A. Sergeyev, V. Rajlich, and J.I. Maletic, "An Information Retrieval Approach to Concept Location in Source Code," in *Proc. 11th Working Conf. Reverse Engineering (WCRE 2004)*, 2004, pp. 214-223.

[15] G. Maskeri, S. Sarkar, and K. Heafield, "Mining Business Topics in Source Code using Latent Dirichlet Allocation," in *Proc. 1st India Software Engineering Conference*, 2008, pp. 113-120.

[16] T.P. Minka and J. Lafferty, "Expection-Propagation for the Generative Aspect Model," in *Proc. 18th Conf. Uncertainty in Artifical Intelligence*, 2002, pp. 352-359.

[17] D. Poshyvanyk, Y.G. Gueheneuc, A. Marcus, G. Antoniol, and V. Rajlich, "Combining Probabilistic Ranking and Latent Semantic Indexing for Feature Location," in *IEEE Int'l Conf. Program Comprehension (ICPC 2006)*, 2006, pp. 137-148.

[18] D. Poshyvanyk, Y.G. Gueheneuc, A. Marcus, G. Antoniol, and V. Rajlich, "Feature Location Using Probabilistic Ranking of Methods Based on Execution Scenarios and Information Retrieval," *IEEE Trans. Softw. Eng.*, vol. 33, no. 6, pp. 420-432, June 2007.

[19] R. Rehurek and P. Sojka, "Software Framework for Topic Modelling with Large Corpora," in *Language Resources and Evaluation (LREC 2010)*, 2010.

[20] Rhino. [Online]. http://www.mozilla.org/rhino/

[21] M. Steyvers and T. Griffiths, "Probabilistic Topic Models," in *Latent Semantic Analysis: A Road to Meaning*, T. Landauer et al., Eds.

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

[23] Understand for Java. [Online]. http://www.scitools.com/