2008 15th Working Conference on Reverse Engineering

Source Code Retrieval for Bug Localization using Latent Dirichlet Allocation

Stacy K. Lukins
Univ. of Alabama in Huntsville
slukins@cs.uah.edu

Nicholas A. Kraft
Univ. of Alabama
nkraft@cs.ua.edu

Letha H. Etzkorn
Univ. of Alabama in Huntsville
letzkorn@cs.uah.edu

Abstract

In bug localization, a developer uses information about a bug to locate the portion of the source code to modify to correct the bug. Developers expend considerable effort performing this task. Some recent static techniques for automatic bug localization have been built around modern information retrieval (IR) models such as latent semantic indexing (LSI); however, latent Dirichlet allocation (LDA), a modular and extensible IR model, has significant advantages over both LSI and probabilistic LSI (pLSI). In this paper we present an LDA-based static technique for automating bug localization. We describe the implementation of our technique and three case studies that measure its effectiveness. For two of the case studies we directly compare our results to those from similar studies performed using LSI. The results demonstrate our LDA-based technique performs at least as well as the LSI-based techniques for all bugs and performs better, often significantly so, than the LSI-based techniques for most bugs.

1. Introduction

Software aging, documentation deficiency, and developer mobility can make software difficult to understand. Because understanding is involved in fifty to ninety percent of the software maintenance effort [26], these factors can slow software maintenance and as a result increase maintenance costs. To help curb these costs, many recent research efforts have focused on (partially or fully) automating software maintenance tasks.

Bug localization is a software maintenance task in which a developer uses information about a bug present in a software system to locate the portion of the source code that must be modified to correct the bug. Due to the size and complexity of modern software, effectively automating this task can reduce maintenance costs by reducing developer effort.

Techniques for automating bug localization take as input information about a subject software system and produce as output a list of entities such as classes, methods, or statements. Static bug localization techniques gather information from the source code (or a model of the code), whereas dynamic techniques gather information from execution traces of the system. Static techniques have some advantages over dynamic techniques: static techniques do not require a working subject software system; thus, they can be applied at any stage of the software development or maintenance processes. Also, unlike most dynamic techniques, static techniques do not require a test case that triggers the bug.

Some recent static techniques for automating bug localization have been built around modern information retrieval (IR) models such as latent semantic indexing (LSI) [21][22][28] and its probabilistic extension pLSI [11]. In this paper we describe a static technique for automating bug localization that is based on another IR model, latent Dirichlet allocation (LDA), whose properties, which include modularity and extensibility, provide advantages over both LSI and pLSI. LDA has previously been applied to mining concepts from source code [17][23] but not to automating bug localization. We provide three case studies that examine whether the advantages of LDA extend to source code retrieval for bug localization. Our results demonstrate the promise of our LDA-based technique for automating bug localization.

2. Background

We begin this section by describing the application of information retrieval (IR) to source code (source code retrieval). We next compare and contrast three information retrieval models: latent semantic indexing (LSI), probabilistic latent semantic indexing (pLSI), and latent Dirichlet allocation (LDA). Finally, we discuss static techniques for bug localization.

1095-1350/08 $25.00 © 2008 IEEE
DOI 10.1109/WCRE.2008.33

2.1. Source Code Retrieval

Some recent source code retrieval techniques operate on models of the source code that are constructed from semantic information embedded in the code, including identifiers and comments, and serve as the “documents”' that are provided as input to an IR technique. An individual document is constructed from an element of the source code such as a package, file, class, or method. The element used to partition the documents determines the granularity of the results returned by the IR technique in response to a user query. The results are a ranked list of documents; the rank of a document is calculated as the similarity between the document and the user query.

2.2. Information Retrieval Models

Latent semantic indexing (LSI) is the application of latent semantic analysis (LSA) to document indexing and retrieval [4] and is the focus of much recent work on source code retrieval [22][28][29]. LSI is based on the vector space model, an algebraic model that represents text documents as vectors of terms, and represents the relationships among the terms and documents in a collection as a term-document co-occurrence matrix where a row in the matrix is a vector that corresponds to a term and gives the relation to each document and a column in the matrix is a vector that corresponds to a document and gives the relation to each term. LSI uses singular value decomposition (SVD) to reduce the co-occurrence matrix; this reduction shrinks the search space and eliminates noise in the document representation while retaining the semantic meanings of the actual documents. The reduced co-occurrence matrix encodes a concept space, and similarity between two documents in the concept space is shown by calculating the cosine of the angle between their vectors [3]. To compare a user query to the documents, the user query is transformed into a document in the concept space before comparing it to the other documents.

The results returned by LSI can be difficult to interpret, because they are expressed using a numeric spatial representation. In addition, while LSI represents *synonymy*, terms with similar meaning, it does not do well when representing *polysemy*, terms with multiple meanings. To address these (and other) shortcomings of LSI, Hofmann introduced probabilistic latent semantic indexing (pLSI), a generative topic model with a strong foundation in statistics [11]. In pLSI each term in a document is modeled as a mixture over a set of multinomial random variables that can be interpreted as topics and each document is modeled as a probability distribution on a fixed set of topics [2]. Studies comparing LSI and pLSI have shown that pLSI has significant advantages over LSI [2][11].

While pLSI provides improvements over LSI, it also introduces new problems. The number of parameters in the pLSI model grows linearly with the number of documents in the collection; thus, pLSI is susceptible to overfitting [8]. In addition, pLSI is not able to predict appropriate topic distributions for new documents, because it is a generative model of the documents in the collection on which it was estimated and hence must choose from the topic distributions generated for those training documents. Thus, pLSI does not perform well when predicting new documents [2][34]. Girolami and Kaban have shown that these deficiencies can be resolved by considering pLSI within the framework of latent Dirichlet allocation (LDA) [8].

Latent Dirichlet allocation (LDA) is a probabilistic and fully generative topic model that is used to extract the latent, or hidden, topics present in a collection of documents and to model each document as a finite mixture over the set of topics [2][8]. Each topic in this set is a probability distribution over the set of terms that make up the vocabulary of the document collection. In LDA, similarity between a document d_i and a query Q is computed as the conditional probability of the query given the document:

$$Sim(Q, d_i) = P(Q \mid d_i) = \prod_{q_k \in Q} P(q_k \mid d_i)$$

where q_k is the k^{th} word in the query. Thus, a document is relevant to a query if it has a high probability of generating the words in the query [33].

Like LSI and pLSI, LDA represents synonymy; however, pLSI and LDA both differ from LSI in their ability to better represent polysemy and to produce immediately interpretable results. In the results returned by LDA, the most likely terms in each topic—the topics with the highest probability—can be examined to determine the likely meaning of the topic. Table 1 lists the set of terms and associated probabilities that constitute a topic, Topic 0, which was automatically generated by an LDA analysis of Mozilla [25]. The set of terms comprises the ten terms with the highest probability of belonging to Topic 0. One can immediately interpret the results and determine that Topic 0 is related to printing a page of data. This is an improvement over LSI; though LSI has been used to extract and label topics [15], it cannot do so directly or in isolation.

LDA retains the advantages of pLSI over LSI and also provides improvements over pLSI. However,

Table 1: Topic extracted from Mozilla

Topic 0	
set	0.474086
str	0.267139
print	0.145522
setup	0.017282
prt	0.011658
page	0.009856
preview	0.007720
engine	0.005232
pm	0.004879
footer	0.003997

LDA is intractable for direct computation, so approximation techniques are required [2][34]. Such techniques include variational methods [2], expectation propagation [24], and Gibbs sampling [9][33], which is a Markov chain Monte Carlo method for estimation.

2.3. Bug Localization

Techniques for automating bug localization differ in how they gather information about the subject software system and in how they analyze that information. Static techniques gather information from the source code directly or from a model of the code.

IR-based static techniques aim to identify the elements of the software system that need to be modified to correct a bug. Such techniques do not attempt to identify every element of the software system that must be fixed. Instead, they aim to identify a starting point from which correction of the bug can be undertaken. Many bug fixes do not relate directly to the bug itself, but rather are the result of the impact of the bug on the other elements of the software system. These additional fixes are considered to be a concern of impact analysis.

Because we are not looking for all elements of the software system relevant to a bug, precision and recall—measures commonly used to determine the effectiveness of IR techniques [10]—are not as useful when applied to bug localization. The most common measure used to determine the accuracy of IR-based static techniques is the rank of the first relevant source element returned by the technique. This indicates the number of elements that the programmer must examine (if following the rankings) before reaching an element that actually needs to be corrected.

3. Previous Work

LSI has been applied to a number of applications of source code retrieval such as concept and feature location [21][22][28][29] and clustering [15]. In several of these studies, the LSI-based approaches are compared to methods utilizing other static techniques for source code retrieval, such as search on dependency graphs and regular expression pattern matching utilities, and show improvement over these other techniques [21][22].

The earlier studies on LSI suggest that combining multiple techniques for source code retrieval is likely to perform better than any one technique alone [21][22]. In fact, in most later studies, LSI was used in conjunction with another method. For example, the study in [15] uses traditional clustering techniques in combination with LSI to derive what the authors refer to as *semantic clusters*. In addition, the studies in [28][29] find that LSI performs much better when combined with a dynamic approach.

Several studies have examined the use of LSI for bug localization [18][28]-[30]. The largest number of bugs are examined in [29], which presents an approach to the bug localization problem using LSI combined with a dynamic technique called Scenario-based Probabilistic Ranking (SPR). The authors present the results of the combined approach as well as the results of the techniques used separately. Because this paper's focus is static techniques for bug localization, we are specifically interested in the LSI results.

In these studies, based on the bug description, the user formulated a query for LSI search and two execution scenarios for SPR, one which exercises the bug and one which does not. Execution of the queries resulted in a list of methods, ranked by similarity to the query. The rank of the first truly relevant method, a method actually fixed as a result of the bug, was presented and used to measure effectiveness.

Results using LSI alone were varied for the bugs analyzed. These results are presented in Section 5.2 and 5.3 and compared to the approach presented in this paper. The combined approach performed well for bugs analyzed, with the first relevant method listed in the top six results returned for each bug. The authors conclude that LSI alone is not an effective method for bug localization [29]. For both LSI and the combined approach, so few bugs in each version of the software system were examined—at most three in any one version of the software systems—that it remains unclear how the method would perform across all bugs in a software system.

To our knowledge, LDA has not previously been applied to the bug localization task. However, it has been applied to source code retrieval in the context of topic mining. Linstead et al. conducted an experiment using LDA to mine concepts from source code [17]. The authors sought to demonstrate the effectiveness of LDA in automatically extracting concepts, in the form of topics, from source code. The study involved the analysis of 1,555 Java projects from SourceForge and Apache. Results indicated the LDA approach produced functional and easily interpretable topics. Maskeri *et al*. used LDA to automatically mine business topics from the source code of the Apache and Petstore software systems [23]. Their analysis was performed at the file level of granularity, and topics were labeled manually. Results indicated the approach was able to successfully discover some, but not all, of the domain topics of the system.

4. LDA-Based Approach to Bug Localization

To perform LDA-based bug localization on a given version of a software system, first an LDA model of the source code is built. Then, the created model is queried as often as necessary to localize bugs existing in that version. The following sections describe the tasks involved in creating and querying an LDA model for a given software build.

4.1. Construct an LDA model

Two steps are necessary to construct an LDA model of a software system: (1) build a document collection from the source code, and (2) perform an LDA analysis on the document collection. These steps are detailed below and illustrated in Figure 1.

Step 1: Build a document collection from the source code. Extract semantic information, such as comments and identifiers, from each source code element at the desired level of granularity (e.g., package, class, method). Preprocess the semantic information as desired before writing it to the document collection, e.g., perform stemming and remove stop words. Store the preprocessed data extracted from each source element as a separate document in the document collection.

To automate Step 1, we created two NetBeans plug-ins: One to analyze Java code and one to analyze C++. Since the experiments in this paper are at the method level of granularity, each document represents one method of the source code. We chose to extract string-literals in addition to comments and identifiers, as they are found in error messages and likely to be included in a bug description. Multi-word identifiers are split into separate words based on common coding practices, e.g., an identifier *printFile* would be split into the terms *print* and *file*. In addition, stop words, including programming language specific keywords, are removed. Each remaining word is stemmed using a Porter stemming algorithm [27] before being included in the document collection.

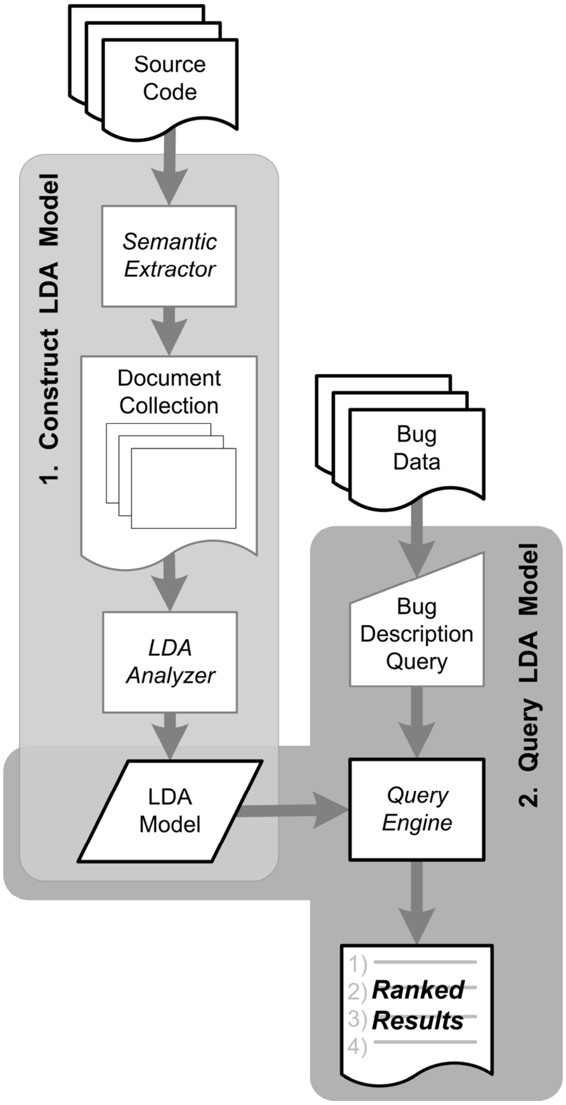

Figure 1: LDA-based approach to bug localization

Execution of either plug-in results in a single file containing the entire document collection for the source files being analyzed. The first line in the file contains the number of documents in the collection and each successive line contains a single document, that is, the list of words extracted from a single method. This format is consistent with the input format required by the tool used in Step 2.

Step 2: Generate an LDA model. Perform an LDA analysis, as described in Section 2.2, on the document collection generated in Step 1. To perform the LDA analysis in Step 2, we used an open-source software tool for LDA analysis called GibbsLDA++ [7]. GibbsLDA++ uses Gibbs sampling to estimate topics from the document collection as well as estimate the word-topic and topic-document probability distributions.

Input into the LDA tool consists of the document collection file generated by Step 1 of the process. Before an LDA analysis can be performed on the document collection using the tool, the following parameters must be set.

- The number of topics.
- The number of iterations for the Gibbs sampling process.
- α, a hyperparameter of LDA, determines the amount of smoothing applied to the topic distributions per document [33].
- β, a hyperparameter of LDA, determines the amount of smoothing applied to the word distributions per topic [33].

The LDA analysis results in the following two probability distributions which, along with the topics themselves, comprise the LDA model.

- *The word-topic probability distribution (*φ*).*
- *The topic-document distribution (*θ*).*

The word-topic distribution contains, for every word in the vocabulary of the document collection and every topic in the model, the probability a given word belongs to a given topic. Similarly, the topic-document distribution contains, for every topic in the model and every document in the collection, the probability that a given topic belongs to a given document.

GibbsLDA++ also outputs a list of topics with the top *n* words in the topic, i.e., the *n* words that have the highest probability of belonging to that topic, where *n* is a parameter that may be set for each analysis. At this point, a static LDA model of the source code has been constructed. This model can then be queried for each bug discovered.

4.2. Query the LDA Model

Now the user may query the LDA model previously generated. Terms in the query should be preprocessed in the same manner as the source code, e.g., stop words removed and stemming performed. Each query results in a list of source code elements ranked by similarity to the query (most similar elements ranked highest). This step is not specific to the task of bug localization; the user may query the model for other purposes as well, e.g., to find candidate software components for reuse or to find components that likely need to be modified as a result of a change request.

We formulated source code queries manually by utilizing information about bugs we extracted from the bug title and description entered into the software's bug repository by the person initially reporting the bug. Details regarding the formation of queries for each case study are discussed in the description for each study.

We created a tool to calculate the similarity between an issued query and each document in the document collection. Our tool returns a list of results to the user ranked by the similarity measure; these represent the elements of the source code that likely need modification to correct a given bug. The user may query an existing LDA model as often as desired.

5. Case Studies in Bug Localization

To assess the viability of an LDA-based approach to bug localization, we performed case studies on three different software systems. The goal of the case studies was to measure how well an LDA-based system can predict the methods that likely need modification to correct a given bug.

All three case studies use the approach outlined in Section 4 to perform bug localization. For the case studies, the number of topics used was K=100. In addition, we used standard parameter values, α=0.05 (50/K) and β=0.01, that have been recommended in the literature [34].

To determine the accuracy of the predictions for each bug, the LDA query results were compared to relevant elements for the bug, i.e., the actual elements fixed by developers to correct the bug. These relevant source code elements were determined by examining the software patch for each bug posted in the software's bug repository.

5.1. Rhino Case Study

For the first case study, we analyzed 35 bugs in version 1.5 release 5 (1.5R5) of the software system Rhino [31], an open source implementation of JavaScript written in Java. The analysis of Rhino allows us to answer an important question. Given a software system, will the LDA-based technique perform well across all bugs in the system or only a few? If the technique performs well for only a few select bugs, it would not be a practical approach to employ. However, this study indicates the LDA-based approach returns a relevant method in the top ten for 77% of the bugs, in the top five for 63% of the bugs, and as the top result for 23% of the bugs.

All bugs that met the following criteria were extracted from Rhino's bug repository.

- Bugs existed in source files of version 1.5R5 implementing the core and compiler functionality of the software.
- Bugs required a method-level fix.

- Bugs were fixed in either version 1.5R5.1 (3 bugs) or v1.6R1 (32 bugs).
- Bugs were categorized as *resolved* or *verified* and as *fixed,* so only valid bugs that in fact had been fixed were returned.

Bugs at all levels of severity were included in the case study, from enhancements to critical defects.

For this case study, the first query for each bug was formed by manually extracting keywords from the bug title. The first query proved sufficient for over one-half (20/35) of the Rhino bugs. A second query was formed for the remainder of the bugs by manually adding to the query keywords from the summary of the initial bug report. At the same time, variants of words that seemed useful were added (e.g., add *parse* in addition to *parser*) in addition to any common abbreviations (e.g., *eol* for *end-of-line*). The second query proved effective for eleven bugs. The final three bugs required a further refinement of the query. For the third query for these bugs, we added any additional words related to the bug (e.g., adding *day* for *daylight savings time*).

Table 2 lists the bug number and title from Rhino's bug repository for each bug examined along with the query we formulated and ran against the LDA model for the software.

Table 2: Rhino bugs analyzed (LDA query words in bold)

Bug #	Bug Title [*Added Query Words*]
58118	ECMA Compliance: **daylight savings time** wrong prior to **year** 1 [*day offset timezone*]
238699	**Context**.**compileFunction** throws **InstantiationException**
238823	**Context**.**compileFunction** *throws* **Null**Pointer **exception**
239068	Scope of **constructor functions** is not **initialized**
244014	Removal of **code complexity limits** in the **interpreter**
244492	**JavaScript**Exception to extend **Runtime**Exception and common **exception** base
245882	**JavaImporter constructor**
249471	**String index out** of range **exception** [*parse bound float global js native char*]
252122	**Double** expansion of **error message**
253323	Assignment to variable '**decompiler**' has no effect in **Parser** [*parse*]
254778	Rhino treats **label** as separated **statement**
254915	Broken "**this**" for **name**() calls (CVS tip regression) [*object with*]
255549	**JVM**-dependent resolution of **ambiguity** when **calling Java methods** [*argument constructor overload*]
255595	Factory **class** for **Context** creation [*call default enter java runtime thread*]
256318	NOT_FOUND and **ScriptableObject**.**equivalentValues**
256339	**Stack**less **interpreter**
256389	Proper **CompilerEnvirons**.is**Xml**Available()
256575	Mistreatment of **end**-of-**line** and **semi**/**colon** [*eol*]
256621	**throw statement**: **eol** should not be allowed
256836	**Dynamic** scope and nested **functions**
256865	Compatibility with **gcj**: changing **ByteCode**.<**constants**> to be int
257128	**Interpreter** and **tail call** elimination [*java js_stack*]
257423	**Optimizer** regression: **this**.**name** += expression generates wrong **code**
258144	Add option to set **Runtime Class** in classes **generated** by **jsc** [*run main script*]
258183	**catch** (e if **condition**) does not **rethrow** of original **exception**
258207	**Exception name** should be **DontDelete** [*delete catch ecma obj object script*]
258417	java.long.**ArrayIndexOut**Of**BoundsException** in org.mozilla.javascript.**regexp**.**NativeRegExp** [*stack size state data*]
258419	**copy paste** bug in org.mozilla.javascript.**regexp**.**NativeRegExp** [*RE data back stack state track*]
258958	**Lookup** of excluded **objects** in **ScriptableOutputStream** doesn't traverse **prototypes**/**parents**
258959	**ScriptableInputStream** doesn't use **Context**'s application **ClassLoader** to **resolve** classes
261278	**Strict mode**
262447	NullPointerException in **ScriptableObject**.**getPropertyIds**
263978	cannot **run** xalan example with Rhino 1.5 release 5 [*line number negative execute error*]
266418	Can not **serialize reg**ular **exp**ressions [*regexp RE compile char set*]
274996	Exceptions with multiple **interpreters** on **stack** may lead to **ArrayIndex**OutOf**BoundsException** [*java wrapped*]

Of the methods fixed by developers as part of the software patch for each bug, the ranks of those methods that ranked highest in the list of methods returned by our query are summarized in Figure 2.

It is interesting to note that a few of the bug titles and descriptions mentioned at least one of the methods that needed to be fixed by name, which sometimes did and sometimes did not help the results. For example, consider bug #238823, which mentions *Context.compile* (as well as a few other methods) in the bug description. *Context.compile*, which was the

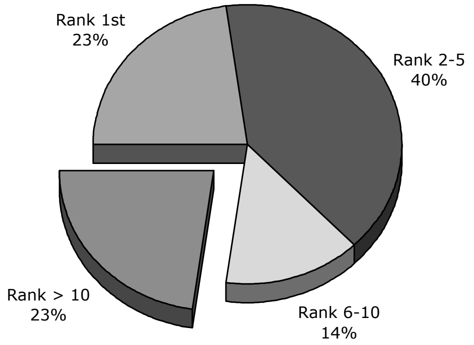

Figure 2: Rank of first relevant method returned for Rhino bugs

highest ranking method, was ranked 4th in the results. Bug #262447 has the title "*NullPointerException in ScriptableObject.getPropertyIds*," which is indeed the method that was fixed; however, the method is ranked 11th in the results. This is consistent with the fact that LDA attempts to capture the meaning of a document beyond just a set of terms.

Other times, the bug title and/or description mentions the name of a method that is ultimately **not** changed due to the bug. For example, for bug #238699, whose title is "*Context.compileFunction throws InstantiationException*," the LDA approach correctly predicts *Codegen.compile* (ranked 1st), not *Context.compileFunction* (ranked 27th), to be the function needing modification.

Query formulation appears to play a role in the quality of the results. For example, the title for bug #261278 is "*Strict mode*." Querying the LDA model with simply the words "*strict mode*" results in *ScriptRuntime.setName* being ranked 12th. However, looking at the bug's description, "*It would be nice to have a strict mode in Rhino that would at least throw an exception for missing var declarations*," and modifying the query to be "*strict mode missing var declarations*" results in the same method being ranked 1st. Therefore, care must be taken to form the queries.

Other queries performed quite well despite very few clues in the bug title and description about methods that may need to be fixed. For example, bug #258207 has a description of "*According ECMAScript standard, section 12.4, the exception name in the catch object should have DontDelete property*." Running the query "*exception name dontdelete delete catch ecma obj object script*" results in *ScriptRuntime.newCatchScope* being returned in 5th position. Bug #256621's title is "*throw statement: eol should not be allowed*" and the query "*throw statement eol*" results in the relevant method *Parser.statementHelper* being ranked 1st.

Overall, the LDA-based approach performed quite well. As illustrated in Figure 2, for 77% (27/35) of the bugs analyzed the first relevant method was returned in the top ten results and for 63% (22/35) of the bugs the first relevant method was returned in the top five results. These results show the use of LDA in bug localization to be a very promising approach.

Table 3: Eclipse bugs analyzed (LDA/LSI query words in bold)

Software Version	Bug No.	Bug Title [*Added query words*]
2.1.3	5138	**Double-click-drag** to **select** multiple words doesn't work [***mouse up down release text offset document position***]
2.0.0	31779	**UnifiedTree** should ensure **file**/**folder** exists [***node system location***]
3.0.2	74149	The **search** words after "" will be ignored [***query quoted token***].

5.2. Eclipse Case Study

The second case study involves the application of our LDA-based technique to the Eclipse software system, an open source IDE written in Java [5]. The analysis of Eclipse allows us to evaluate the accuracy of our approach when used in a large software system. This same system was analyzed in [29] so it allows a direct comparison of LDA to LSI for bug localization.

We ran the same queries as in [29] on the Eclipse LDA model, and the accuracy of the LDA predictions equaled or exceeded that of the LSI predictions. For each bug examined, Table 3 lists the software version in which the bug occurred, the bug number and bug title taken from the Eclipse bug repository, and the query used in both our LDA analysis and the earlier LSI analysis from [29]. Table 4 presents the results of the LDA query—the name of the first relevant method returned by the query and its LDA rank—along with the LSI results for comparison. Note that the *TextDoubleClickStrategyConnector.mouseUp* method is referred to as *TextViewer.mouseUp* in [29], but the method belongs to an inner class of *TextViewer* called *TextDoubleClickStrategyConnector* and we have used the inner class name.

These results show the LDA-based approach performed the same as LSI for bug #31779 and outperformed LSI for the other two. This case study suggests LDA performs at least as well as LSI for bug localization and indicates the LDA-based technique scales well to larger software systems. The Mozilla case study below further validates these findings, showing an even greater improvement over LSI.

Table 4: Comparison of LDA to LSI over Eclipse

Bug No.	First Relevant Method Returned by LDA and LSI [29] along with its LDA/LSI ranks
5138	TextDoubleClickStrategyConnector.mouseUp LDA Rank: **2** JavaStringDoubleClickSelector.doubleClicked LSI Rank: **7**
31779	UnifiedTree.createChildNodeFromFileSystem LDARank: **2** LSI Rank: **2**
74149	QueryBuilder.tokenizeUserQuery LDA Rank: **1** LSI Rank: **5**

5.3. Mozilla Case Study

For our third case study, we applied the LDA-based bug localization technique to five bugs in Mozilla, an open source internet application suite written in C++. Mozilla, like Eclipse, is a large software system; thus it provides an additional test point for determining the performance of LDA for bug localization in larger systems. This same system was also analyzed in [29] so it allows another comparison of LDA to LSI for bug localization.

For the LDA technique, we first ran the same queries as the LSI study. Bug #209430 resulted in a vast improvement over LSI using the same query, as shown in Table 6. For the other bugs, a second query was formed by adding keywords from the bug title and summary for each bug as well as removing any words that did not seem relevant. This second query was effective for two of the bugs. We issued a third query for the remaining two bugs by adding any additional words related to the bugs. For example, for bug #216154, the bug title and summary discusses a problem with clicking an anchor, so the word *link* seemed an obvious keyword to use in addition to the words *anchor* and *target* from the bug title.

For each bug, Table 5 lists the version of the software in which the bug existed, the bug title and number taken from the Mozilla bug repository, the LDA query used in this study, and the LSI query used in [29]. (Note the bug titles are not the same in the LSI paper, which listed an excerpt from the longer bug description rather than the bug title.)

For bug #225243, the study in [29] indicates this bug existed in version 1.6 of the software. However, examination of the Mozilla 1.6 code revealed the software was patched prior to the release of 1.6, and the bug actually existed in version 1.6 Alpha (1.6a), a release candidate of 1.6. This paper uses the proper version of the software (1.6a) for that bug.

The LDA results are presented in Table 6, with the LSI Rank included for comparison. The LDA rank of the first relevant method returned by the LDA query for each bug is in the top five for three of the bugs and in the top ten for all of the bugs, which is a significant improvement over LSI.

6. Threats to Validity

There are several issues that could affect the validity of our results. Like other semantic-based techniques, the LDA approach used depends in part on the quality of the semantic information present in the source code. In addition, the LDA approach is likely sensitive to the quality of the information present in the bug titles and descriptions, which depends on the ability of the bug submitter to accurately describe the problem. If the bug description does not truly reflect the situation, then it may be difficult to accurately summarize the bug in a query. However, the study in [13] examined 200,000 bug titles and found that “95% of noun phrases referred to visible software entities, physical devices or user actions.” This suggests that information in bug titles and summaries truly is useful

Table 5: Mozilla bugs analyzed with corresponding LDA/LSI queries

Software Version	Bug No.	Bug Title	LDA Query	LSI Query [29]
1.6	182192	Remove quotes (“) from collected addresses	collect collected sender recipient email name names address book addressbook	collect collected sender recipient email name names address addresses addressbook
1.5.1	209430	Ctrl+Delete and Ctrl+Backspace delete words in the wrong direction	delete deleted word action	delete deleted word action
1.5.1	216154	Anchors in e-mails are broken (Clicking Anchor doesn’t go to target in e-mail)	mailbox uri url pop msgurl service anchor target link	mailbox uri url msg pop3 msgurl service
1.6a	225243	[ps] Page appears reversed (mirrored) when printed	print page post script postscript image	print page orientation portrait landscape postscript postscriptobj
1.5.1	231474	attachments mix contents	attach mailattachcount msgattach parsemailmsgst	attachment encoding content mime

Table 6: Comparison of LDA to LSI over Mozilla

Bug No.	First Relevant Method Returned by LDA and LSI [29] with LDA/LSI Ranks
182192	nsAbAddressCollector::CollectAddress LDA Rank: **3** LSI Rank: **37**
209430	nsPlaintextEditor::DeleteSelection LDA Rank: **9** LSI Rank: **49**
216154	nsMailboxService::NewURI LDA Rank: **4** LSI Rank: **76**
225243	nsMailboxService::NewURI LDA Rank: **9** LSI Rank: **24**
231474	Root::MimeObject_parse_begin LDA Rank: **4** LSI Rank: **18**

in localizing bugs.

The quality and usefulness of the queries may vary with the skill of the debugger; this could impact the accuracy of the results. A less experienced debugger may have more trouble formulating a query that would lead to good results. However, in many of the bugs analyzed in the completed case studies, good results were obtained with the first query issued, using terms taken directly from the bug titles and descriptions, suggesting very little knowledge is required to formulate useful queries.

We did not have available an LSI-based bug localization tool; therefore, we compared our results to the published results of previous researchers [29]. These published results represented the largest LSI study for bug localization previously performed; therefore, our comparison over Mozilla and Eclipse was the largest possible that allowed us to compare our technique to LSI results. The fact that our LDA-based technique performed as well as LSI for one bug and better than LSI for all other bugs analyzed in those systems shows our technique to be very promising. In our other study over Rhino, we showed how our LDA approach was able to scale to a much larger number of bugs.

7. Conclusions and Future Work

The case studies presented in this paper show LDA can successfully be applied to source code retrieval for the purpose of bug localization. Furthermore, the results suggest an LDA-based approach is more effective than approaches using LSI alone for this task. Our case studies examined the same bugs in Mozilla and Eclipse as [29]. The LSI analysis resulted in only three out of the eight total bugs analyzed (37.5%) with the first relevant method ranked in the top ten results returned. For the LDA approach, the user query resulted in all eight (100%) of the bugs ranked in the top ten. Ultimately, the results of the case studies we presented demonstrate that our LDA-based technique performs at least as well as the LSI-based techniques in one case (for one bug) and performs better, often significantly so, than the LSI-based technique in all other cases.

In addition, previous studies of LSI-based bug localization have analyzed at most three bugs in any one version of a software system. Our analysis of Rhino allowed us to see how our LDA approach performed on 35 bugs in a single version of Rhino, which was all bugs in that version that met the given criteria. While it is unknown how LSI performs across all bugs (or even a majority of the bugs) in a software system, our Rhino study shows the LDA-based approach returns a relevant method in the top ten for 77% of the bugs analyzed.

In future studies, we plan to examine more bugs in larger software systems, such as Eclipse and Mozilla, to determine how the approach works across a large number of bugs in those systems. We plan as well to investigate the bugs for which the LDA technique did not perform well in order to improve the technique in those cases.

8. References

[1] G. Antoniol and Y.G. Guéhéneuc, "Feature Identification: A Novel Approach and a Case Study", In *Proc. 21st IEEE Int. Conf. on Software Maintenance*, Budapest, Hungary, September 2005, pp. 357-366.

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

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

[4] S.T. Dumais, "LSA and Information Retrieval: Getting Back to Basics," *Handbook of Latent Semantic Analysis*, T. Landauer, D. McNamara, S. Dennis, W. Kintsch (eds), Lawrence Erlbaum Associates, 2007, pp. 293-321.

[5] The Eclipse Home Page, http://www.eclipse.org.

[6] C. Gall (Stein), S. Lukins, L. Etzkorn, S. Gholston, P. Farrington, D. Utley, J. Fortune, and S. Virani, "Semantic Software Metrics Computed from Natural Language Design Specifications," *IET Software*, 28(10), February 2008, pp. 17-26.

[7] GibbsLDA++. http://gibbslda.sourceforge.net/.

[8] M. Girolami, and A. Kabán, "On an Equivalence between PLSI and LDA", In *Proc. 22nd Annu. ACM SIGIR*

Int. Conf. on Research and Development in Information Retrieval, Toronto, Ontario, Canada, July 2003, pp. 433-434.

[9] T.L. Griffiths and M. Steyvers, "Finding Scientific Topics", In *Proc. Nat. Academy of Sciences*, 101(1), April 2004, pp. 5228-5235.

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

[11] T. Hofmann, "Probabilistic Latent Semantic Indexing", In *Proc. 22nd Annu. ACM SIGIR Int. Conf. on Research and Development in Information Retrieval,* Berkeley, CA, USA, August 1999, pp. 50-57.

[12] ISO/IEC 14764:2006, "Information Technology – Software and Systems Engineering - Software Engineering – Software Life Cycle Processes – Maintenance," International Organization for Standardization, Geneva, Switzerland, 2006.

[13] A.J. Ko, B.A. Myers and D.H. Chau, "A Linguistic Analysis of How People Describe Software Problems," In *Proc. of IEEE Conf. on Visual Language and Human-Centric Computing (VL/HCC)*, 2006, pp. 127-134

[14] A. Kuhn, S. Ducasse and T. Gîrba, "Enriching Reverse Engineering with Semantic Clustering", In *Proc. 12th Working Conf. on Reverse Engineering*, Pittsburg, PA, November 2005, pp. 133-142.

[15] A. Kuhn, S. Ducasse, and T. Gîrba, "Semantic Clustering: Identifying Topics in Source Code", *Information Software and Technology*, 49(3), Butterworth-Heinemann, Newton, MA, USA, March 2007, pp. 230-243.

[16] D.J. Lawrie, H. Field, and D. Brinkley, "Leveraged Quality Assessment using Information Retrieval Techniques", In *Proc. 14th IEEE Int. Conf. on Program Comprehension,* Athens, Greece, June 2006, pp. 149-158.

[17] E. Linstead, P. Rigor, S. Bajracharya, C. Lopes, and P. Baldi, "Mining Concepts From Code with Probabilistic Topic Models", In *Proc. 22nd IEEE/ACM Int. Conf. on Automated Software Engineering*, Atlanta, Georgia, USA, November 2007, pp. 461-464.

[18] D. Liu, A. Marcus, D. Poshyvanyk, and V. Rajlich, "Feature Location via Information Retrieval Based Filtering of a Single Scenario Execution Trace", In *Proc. 22nd IEEE/ACM Int. Conf. on Automated Software Engineering*, Atlanta, Georgia, November 2007, pp. 234-243.

[19] J.I. Maletic and A. Marcus, "Supporting Program Comprehension using Semantic and Structural Information", In *Proc. 23rd Int. Conf. on Software Engineering*, Toronto, Ontario, Canada, May 2001, pp. 103-112.

[20] A. Marcus, A. De Lucia, J.H. Hayes, D. Poshyvanyk, "Working Session: Information Retrieval Based Approaches in Software Evolution", In *Proc. 22nd IEEE Int. Conf. on Software Maintenance*, Philadelphia, PA, USA, September 2006, pp. 197-209.

[21] A. Marcus, V. Rajlich, J. Buchta, M. Petrenko, and A. Sergeyev, "Static Techniques for Concept Location in Object-Oriented Code", In *Proc. 13th Int. Workshop on Program Comprehension*, St. Louis, MO, USA, 2005, pp. 33-42.

[22] A. Marcus, A. Sergeyev, V. Rajlich, J.I. Maletic, "An Information Retrieval Approach to Concept Location in Source Code", In *Proc. 11th Working Conference on Reverse Engineering*, Delft, The Netherlands, November 2004, pp. 214-223.

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

[24] T.P. Minka and J. Lafferty, "Expectation-Propagation for the Generative Aspect Model", In *Proc. 18th Conf. on Uncertainty in Artificial Intelligence*, Edmonton, Alberta, Canada, April 2002, pp. 352-359.

[25] The Mozilla Home Page, https://www.mozilla.org.

[26] H.A. Müller, J.H. Jahnke, D.B. Smith, M.-A. Storey, S.R. Tilley, and K. Wong, "Reverse Engineering: A Roadmap," In *Proc. of Future of Software Engineering*, Limerick, Ireland, June 2000, pp. 47-60.

[27] The Porter Stemming Algorithm, http://tartarus.org/~martin/PorterStemmer/.

[28] D. Poshyvanyk, Y.G. Guéhéneuc, A. Marcus, G. Antoniol, and V. Rajlich, "Combining Probabilistic Ranking and Latent Semantic Indexing for Feature Location", In *Proc. 14th IEEE Int. Conf. on Program Comprehension*, Athens, Greece, June 2006, pp. 137-148.

[29] D. Poshyvanyk, Y.G. Guéhéneuc, A. Marcus, G. Antoniol, and V. Rajlich, "Feature Location Using Probabilistic Ranking of Methods Based on Execution Scenarios and Information Retrieval", *IEEE Trans. Softw. Eng.*, 33(6), June 2007, pp. 420-432.

[30] D. Poshyvanyk and A. Marcus, "Combining Formal Concept Analysis with Information Retrieval for Concept Location in Source Code", In *Proc. 15th IEEE Int. Conf. on Program Comprehension*, Banff, Alberta, Canada, June 2007, pp. 37-48.

[31] Rhino. http://www.mozilla.org/rhino/.

[32] D. Rousidis and C. Tjortjis, "Clustering Data Retrieved from Java Source Code to Support Software Maintenance: A Case Study", In *Proc. 9th European Conf. on Software Maintenance and Reengineering*, Manchester, UK, March 2005, pp. 276-279.

[33] M. Steyvers and T. Griffiths, "Probabilistic Topic Models", *Handbook of Latent Semantic Analysis*, T. Landauer, D. McNamara, S. Dennis, W. Kintsch (eds), Lawrence Erlbaum Associates, 2007.

[34] X. Wei and B. Croft, "LDA-Based Document Models for Ad-hoc Retrieval", In *Proc. 29th Annu. Int. ACM SIGIR Conf. on Research & Development on Information Retrieval*, Seattle, WA, USA, 2006, pp. 178-185.