Information and Software Technology 52 (2010) 972–990

Contents lists available at ScienceDirect

Information and Software Technology

journal homepage: www.elsevier.com/locate/infsof

Bug localization using latent Dirichlet allocation

Stacy K. Lukins[a,*], Nicholas A. Kraft[b,1], Letha H. Etzkorn[a,2]

[a] Computer Science Department, University of Alabama in Huntsville, Huntsville, AL 35899, USA
[b] Department of Computer Science, University of Alabama, Tuscaloosa, AL 35487-0290, USA

ARTICLE INFO

Article history:
Received 13 June 2009
Received in revised form 9 April 2010
Accepted 11 April 2010
Available online 22 April 2010

Keywords:
Bug localization
Program comprehension
Latent Dirichlet allocation
Information retrieval

ABSTRACT

Context: Some recent static techniques for automatic bug localization have been built around modern information retrieval (IR) models such as latent semantic indexing (LSI). Latent Dirichlet allocation (LDA) is a generative statistical model that has significant advantages, in modularity and extensibility, over both LSI and probabilistic LSI (pLSI). Moreover, LDA has been shown effective in topic model based information retrieval. In this paper, we present a static LDA-based technique for automatic bug localization and evaluate its effectiveness.
Objective: We evaluate the accuracy and scalability of the LDA-based technique and investigate whether it is suitable for use with open-source software systems of varying size, including those developed using agile methods.
Method: We present five case studies designed to determine the accuracy and scalability of the LDA-based technique, as well as its relationships to software system size and to source code stability. The studies examine over 300 bugs across more than 25 iterations of three software systems.
Results: The results of the studies show that the LDA-based technique maintains sufficient accuracy across all bugs in a single iteration of a software system and is scalable to a large number of bugs across multiple revisions of two software systems. The results of the studies also indicate that the accuracy of the LDA-based technique is not affected by the size of the subject software system or by the stability of its source code base.
Conclusion: We conclude that an effective static technique for automatic bug localization can be built around LDA. We also conclude that there is no significant relationship between the accuracy of the LDA-based technique and the size of the subject software system or the stability of its source code base. Thus, the LDA-based technique is widely applicable.

1. Introduction

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. In particular, a developer formulates a query using information in a bug report and uses the query to search the software system in order to determine which parts of the code must be changed to "fix" 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. This list indicates elements of the source code that likely need modification to correct the bug in question. 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, e.g., static techniques do not require a working subject software system. Thus, static techniques can be applied at any stage of the software development or maintenance processes. Additionally, unlike most dynamic techniques, static techniques do not require one or more test cases that trigger 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) [1,27] and its probabilistic extension, pLSI [13]. 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 shown to be a promising

[*] Corresponding author. Tel.: +1 256 824 6088, +1 256 726 8620; fax: +1 256 824 6039.
E-mail addresses: slukins@cs.uah.edu (S.K. Lukins), nkraft@cs.ua.edu (N.A. Kraft), etzkorl@uah.edu (L.H. Etzkorn).
[1] Tel.: +1 256 348 4740; fax: +1 256 348 0219.
[2] Tel.: +1 256 824 6291; fax: +1 256 824 6039.

0950-5849/$ - see front matter © 2010 Elsevier B.V. All rights reserved.
doi:10.1016/j.infsof.2010.04.002

technique for a variety of source code retrieval tasks, such as mining concepts from source code [17,21]. In addition, our previous work [19] showed LDA to be a viable technique for bug localization and provided an initial assessment of the accuracy of an LDA-based approach to source code retrieval for bug localization. This paper briefly reports the results of our previous study [19] and expands it significantly by measuring the accuracy of LDA-based bug localization when applied to a total of 322 bugs across multiple iterations of two software systems. This study is much larger than previous LSI-based bug localization studies [27–29] and illustrates the scalability of our LDA-based technique.

In addition to examining the scalability of our LDA-based bug localization approach, we also examine whether our approach is adversely affected by the stability of the software. IR-based software engineering techniques using LSI or LDA operate on informal semantic tokens (comments and identifiers) present in the code. This dependency could present a problem when applying these IR-based techniques to changing software. For example, design-level changes such as class name changes could potentially affect the accuracy of IR-based software engineering techniques. Also, the study in [9] found that newly added code in a changing software system is sparsely commented; thus, newly added classes, as well as classes with many internal changes, likely have less semantic information available to be used by IR-based software engineering techniques. Thus we argue that changeable or unstable code very often could result in poor or missing semantic tokens, which therefore could affect the results of our approach. For this reason we examine the relationship between our approach and code instability.

In the past, various stability metrics have been shown to accurately reflect the stability of software across multiple development iterations [1,16]. In this paper, we compare these metrics to the results of our approach to determine whether or not the instability of the software results in poorer operation of our LDA-based technique.

In summary, our primary goal is to thoroughly explore our static LDA-based approach to bug localization, and to examine its advantages, disadvantages, and boundaries. To do this, our paper:

- ○ Discusses the advantages and disadvantages of an LDA-based approach compared to other information retrieval approaches.
- ○ Demonstrates the improved accuracy of our LDA-based bug localization technique compared to a previous LSI-based bug localization technique over the same data examined in the earlier LSI-based study.
- ○ Illustrates the scalability of our LDA-based bug localization technique to large software packages.
- ○ Determines whether the accuracy of our LDA-based bug localization technique, as one example of an IR-based bug localization technique, is sensitive to software stability.

The remainder of this paper is organized as follows: Section 2 is related work. Section 3 describes our research approach. Section 4 is results, and Section 5 presents conclusions.

2. Related work

This related work section is in two parts, IR-based source code retrieval and software metrics. First, we describe IR-based source code retrieval techniques.

2.1. IR models for source code retrieval

Some recent source code retrieval techniques operate on models of source code rather than directly on source code. Techniques that form the models accept as input a corpus, or document collection, that represents the source code being analyzed. The corpus itself is constructed from semantic information embedded in the code, including identifiers and comments, which is extracted and collected into units that serve as the documents. An individual document is constructed from an element of the source code such as a package, file, class, or method. Queries result in a ranked list of documents, where the rank of a document is determined by the similarity between the document and the user query. The most similar documents are returned first. Two such information retrieval models receiving attention of late are latent semantic indexing (LSI) and latent Dirichlet allocation (LDA).

LSI is the application of latent semantic analysis (LSA) to document indexing and retrieval [22] and is the focus of much recent work on source code retrieval [22,27,28]. 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. LSI uses singular value decomposition (SVD) to reduce the co-occurrence matrix. Similarity between two documents in the concept space is measured by calculating the cosine of the angle between their vectors [27]. To compare a user query to the documents, the user query is first transformed into a document in the concept space. Note that LSI requires the use of a dimensionality reduction parameter that must be tuned for each document collection [14]. LSI has previously been shown to be superior to the vector space model itself in that the vector space model does not handle synonymy or polysemy [6]. Also, LSI has been shown to work better in various software engineering environments than does the vector space model [20,28].

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 as 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 [13]. 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 [4]. Studies comparing LSI and pLSI have shown that pLSI has significant advantages over LSI [4,13].

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 [11]. 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 performance suffers when predicting new documents [4,35]. Girolami and Kaban have shown that these deficiencies can be resolved by considering pLSI within the framework of latent Dirichlet allocation (LDA) [11].

Latent Dirichlet allocation (LDA) is a probabilistic and fully generative topic model 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 [4,11]. 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:

$$Sim(Q, d_i) = P(Q|d_i) = \Pi_{q_k \in Q} P(q_k|d_i) \tag{2.1}$$

where q_k is the kth 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 [13].

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 terms 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 [23]. The set of terms comprises the 10 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. The human related reasoning for this conclusion might go as follows: I have two terms related to printing (“print” and “prt”), three terms related to a page (“page,” “footer,” and “preview”), and two terms related to settings (“set”, the stemmed version of “settings”, and “setup”). Probably most humans would stop at this point and decide the topic is printing a page (print settings for a page). However, going further and examining the probabilities, a human would note that “print,” “set,” and “str” have the highest probabilities. The word “str” could be considered related to text data—otherwise it is not particularly related to the other words in the list. Thus, “print” and “set” are the obvious high probability choices that go with the other words in the topic, reinforcing our conclusion that the topic likely represents print settings for a page.

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 while simultaneously providing improvements over pLSI. However, LDA is intractable for direct computation, so approximation techniques are required [4,35]. Gibbs sampling, a specific form of Markov Chain Monte Carlo (MCMC), can be used to approximate an LDA model by directly estimating the assignment of words to topics given the observed words in a document collection (corpus) [12].

2.1.1. Choosing the number of topics in LDA modeling

Much discussion has occurred in the literature regarding the selection of the number of topics to use when building an LDA model, with common values ranging from 50 to 300 topics. One hundred topics is most commonly used, having been found to be successful for document collections of up to around 20,000 documents in multiple cases [17] and as many as 113,316 [17]. Three hundred topics have also been found useful in a number of studies with corpora having around 30,000–40,000 documents [12,33]. While few studies have been performed on collections of greater than 50,000 documents, it is generally accepted in LDA language modeling that more topics are necessary for larger document collections. The largest corpus analyzed using LDA for retrieval purposes contained 242,918 documents [35], and it was determined experimentally that 800 topics yielded the highest precision when querying the model.

Many have experimented with both manual and automatic techniques for choosing an optimal value [12]. When applied to source code, the study in [21] ultimately found that automated techniques to determine number of topics are often inadequate: “... in a number of cases, the number of topics is better supplied by domain experts and architects of the system.”

The difficulty with choosing the appropriate number of topics in LDA could be considered similar to the choice of a dimensionality reduction parameter in LSI—in both cases the size and vocabulary of the document collection could affect behavior.

2.2. Source code stability

Stability measures attempt to quantify the amount of evolutionary change (or lack of it) from one iteration of a software system to the next. Stability is considered a desirable quality of software design; unstable software tends to have lower reliability and higher maintenance costs [16].

Alshayeb and Li proposed a stability metric [1] (which we refer to as SDI_{inh}), which was a redefinition of the original *SDI* metric proposed in [16]. Both the original *SDI* metric and the refined SDI_{inh} metric were designed to compute the percentage of change from the design in iteration t (D_t) to the design in iteration $t+1$ (D_{t+1}) of a software system and are defined as follows. Given the following measures:

a the number of classes whose names were modified from D_t to $D_t + 1$,
b the number of new classes added to $D_t + 1$,
c the number of classes removed from D_t,
d the number of classes whose inheritance hierarchies were modified from D_t to $D_t + 1$, and
m the total number of classes in D_t

where a, b, c, d, and m are all positive integers. *SDI* and SDI_{inh} are non-negative values.

$$SDI = \frac{(a+b+c)}{m} \times 100 \tag{2.2}$$

$$SDI_{inh} = \frac{(a+b+c+d)}{m} \times 100 \tag{2.3}$$

Olague et al. [25] further refined Alshayeb and Li’s *SDI* metric. Olague et al’s SDI_e metric used different measures in order to allow the metric to be easier to calculate automatically. In addition, SDI_e used entropy to dampen the potential effect on the overall system stability values that may be caused by large changes in a single measure. SDI_e is defined as follows [25]:

1. *Newly created*: Classes that were added to iteration $t+1$ of the software.
2. *Removed*: Classes removed from iteration t of the software.
3. *Changed*: Classes with internal changes from iteration t to $t+1$.
4. *Unchanged*: Classes that remained the same from iteration t to iteration $t+1$.

$$SDI_e = -\sum_{i=1}^{j} \frac{C-i}{N} \log_2 \frac{C_i}{N} \tag{2.4}$$

where j is the number of categories, i is an integer value $1, 2, \ldots, j$ which represents one of the categories listed above, each C_i represents the number of classes that belong to category i, and N is the total number of classes in iteration $t+1$ of the software.

To determine the number of classes that were modified from one iteration of a software system to the next (classes changed

Table 1
Top 10 terms in Topic 0 extracted from Mozilla.

Topic 0									
Set	Str	Print	Setup	Prt	Page	Preview	Engine	Pm	Footer
0.4740	0.2671	0.1455	0.0173	0.0117	0.0099	0.0077	0.0052	0.0049	0.00440

category), Olague et al. calculated class metrics on all classes in the software system using a commercial metrics collection tool. For each class, if any of the 200 class metric values changed from one version of the software to the next, then the class was considered to have changed. As Roden [31] has observed, several of the metrics produced by the commercial tool were arguably inappropriate to use as change or stability measures. Thus, for analyses in this paper we employ a simpler version of the SDI_e metric, which we will call $SDI_{e,ck}$. In this metric we use only the Chidamber and Kemerer (CK) suite of object-oriented class metrics (see next section for a discussion of these metrics) [5]. The ideal metrics that could be used for a metrics-based stability metric such as this have not yet been determined—this is a topic of future research. However, the Chidamber and Kemerer metrics are well known and supported by many metrics tools, so they represent a reasonable initial version of a metrics-based stability metric.

2.3. Chidamber and Kemerer object-oriented metrics suite

The Chidamber and Kemerer (CK) suite [5] of OO metrics is defined as follows: (we collected these metrics using Understand for Java™ [34]):

- *WMC* (*Weighted Methods Per Class*): Sum of the complexities of the methods of a class. Assuming unit complexity per method, a count of the number of local methods in a class.
- *DIT* (*Depth of Inheritance Tree*): The max depth of a class in the inheritance hierarchy.
- *NOC* (*Number of Children*): A count of the number of immediate subclasses of a class.
- *CBO* (*Coupling Between Object Classes*): The number of other classes to which a class is coupled, where coupling is the extent to which two classes interact.
- *RFC* (*Response for a Class*): A count of the number of local and inherited class methods (as calculated by the Understand for Java™ tool. Note that the original Chidamber and Kemerer suite defined this as all the local methods plus all methods called by local methods—the Understand for Java™ definition is somewhat different from the original.
- *LCOM* (*Lack of Cohesion in Methods*): A measure of the extent to which the methods of a class act upon the same instance variables of the class. Measured by Understand for Java™ as percent of Lack of Cohesion, but the exact definition used by the tool is unclear.

One or more of the Chidamber and Kemerer metrics would normally change values if the underlying code changed. However, several of these metrics can also be used as indicators of code complexity. The WMC metric is a measure of the internal complexity of an individual class [5]. Cohesion metrics (such as LCOM) have also been shown to be related to complexity of an individual class [32]. Several of the other metrics, DIT, NOC, CBO, and RFC, have been shown to evaluate the external complexity of a class, that is, the complexity of the relations between multiple classes [32].

3. Research approach

3.1. LDA-based bug localization approach

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 Fig. 1.

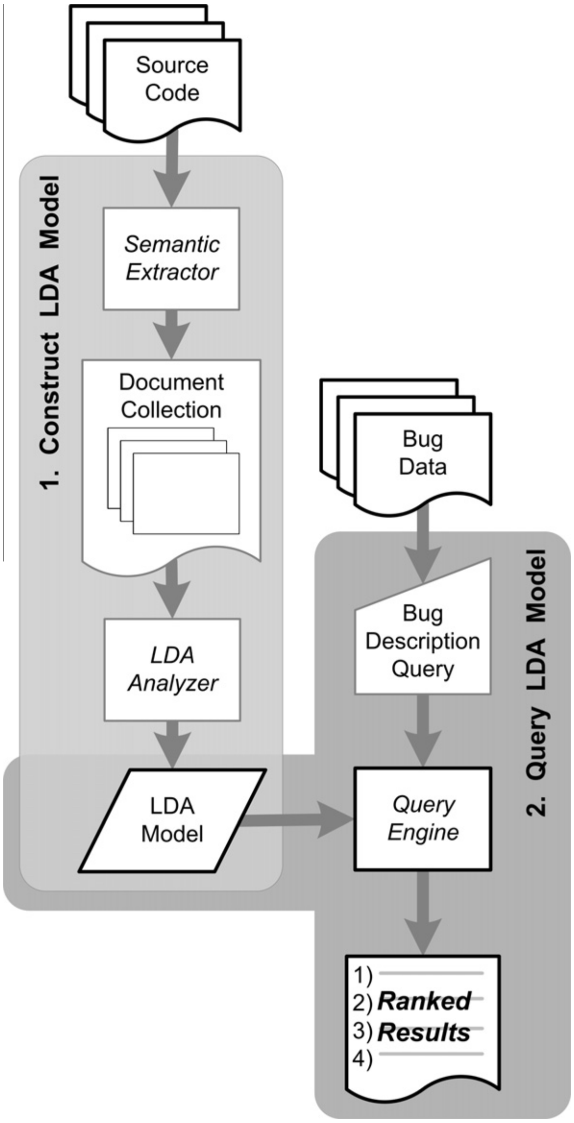

Fig. 1. LDA-based approach to bug localization.

3.1.1. 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 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 collection.

For this paper, we created all documents at the method level of granularity, that is, each document represents one method from 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*. Each word is stemmed using a Porter stemming algorithm [26].

3.1.2. Step 2: Generate an LDA model

To perform the LDA analysis in Step 2, we used an open-source software tool for LDA analysis called GibbsLDA++ [10]. GibbsLDA++ uses Gibbs sampling to estimate topics from the document collection as well as estimate the word–topic and topic–document probability distributions.

Before an LDA analysis can be performed on the document collection using the tool, the following parameters must be set:

1. The number of topics.
2. The number of iterations for the Gibbs sampling process.
3. α, a hyperparameter of LDA, determines the amount of smoothing applied to the topic distributions per document [33].
4. 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 that, along with the topics themselves, constitute the LDA model: (1) the word–topic probability distribution (φ) and (2) 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.

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. Each query results in a list of source code elements ranked by similarity to the query (most similar elements returned first).

Our query building methodology is based originally on the methodology presented by Poshyvanyk et al. [28]. However, we have made our query-building steps more formal and specific, and thus more repeatable than that of Poshyvanyk et al. In Poshyvanyk et al.'s case studies on matching bug reports to code (Case Studies 2 and 3 in their paper [28]), the authors started from simple queries consisting of only a few terms from the bug description. This resulted in formulating at least one extra query each time and at most three extra queries. After each query step, they reformulated their query to be more specific to their search. In each step, selecting the initial queries and refining them was based on the developer's experience.

In our study, 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. The following process was used to create queries:

1. Form an initial query by manually extracting keywords from the bug title. Words not related to the bug domain (e.g., non-useful words such as "doesn't work") were ignored.
2. Form a second query by manually adding keywords from the summary of the initial bug report to the first query.
3. Form a third query by adding words related to the bug and/or removing words from query 1 or 2 that were less applicable to the bug domain (e.g., *error*). Words added were limited to the following:
   a. Common abbreviations (e.g., *eol* for *end-of-line*) or whole words when an abbreviation was used (e.g., *management* for *mgmt*).
   b. Variants of words already in the query (e.g., *parse* for *parser*).
   c. Synonyms of words in the query (e.g., *quit* for *kill* [process]).
   d. Sub-words of words in the query (e.g., *name* for *rename*).

Each step results in a single query, and at most three queries were formed for each bug in the case studies. That ensured we did not refine repeatedly until a good result was achieved. The only deviation from this process occurred in Case Study 1 (see below). Our query mechanism relies on the experience of the developer, as was true in Poshyvanyk et al. [28], in that the selection of keywords and addition of words more or less applicable to the bug domain is based on developer experience. However, our process is more mechanical than that employed by Poshyvanyk et al. because the kinds of words that could be added are limited as described above.

3.2. Case studies

To determine the effectiveness and accuracy of the LDA-based approach to bug localization and to explore the relationship between source code stability and LDA-based bug localization, we designed four different case studies. In all our case studies, we define the accuracy of our approach for a query based on a single bug in terms of the rank (number of methods down from the first method returned) of the first relevant method, actually related to the bug, returned by our approach. This is further discussed in Section 3.2.2 below. Over many bugs, we define the accuracy of our approach as the average rank of the first relevant method returned for each of the bugs examined.

Case Study 1 examined whether the accuracy of LDA-based bug localization is better than LSI, over the same data used in previous LSI studies [28]. In Case Study 1, our goal was to compare the static LDA-based technique to the static LSI-based technique only. We did not compare to vector spaces since that has been largely shown previously (see Section 2.1). Our focus was also not on comparing the static LDA-based technique to dynamic techniques or to combined static and dynamic techniques (see Section 4.1.3).

Case Study 2 examined the accuracy of the LDA-based bug localization technique over all the bugs available in one complete software system. This complete analysis allows us to answer whether the LDA-based technique performs well across all bugs in a system, or only a few bugs. This is an indication of the practicality of this technique. Case Study 3 expands the determination of the accuracy of the LDA-based technique to a much larger number of bugs, over multiple iterations of two software systems. This is a much larger number of bugs than was examined in Case Study 2, and thus gives an indication of the scalability of the technique. However, although Case Study 3 examines more bugs than Case Study 2, it does not examine all bugs (whereas Case Study 2 does examine all bugs in one version of a package). Thus Case Study 2 and Case Study 3 are complementary. Finally, in Case Studies 4 and 5 we examine our LDA-based bug localization vs. software size, vs. source code stability, and vs. source code complexity.

3.2.1. Data examined in the case studies

We analyzed three open-source software systems in our case studies: Mozilla [23], Rhino [30], and Eclipse [8]. It was necessary to choose versions of the software for which bug data, such as bug descriptions and software patches, were available. Software versions analyzed in each study are listed in Table 2.

For our Case Study 1, we chose five bugs to analyze from Mozilla and three bugs to analyze from Eclipse because of their use in a previously published LSI-based analysis performed by Poshyvanyk et al. [28]. In this paper, we analyze these same bugs using our LDA-based analysis, and compare our results to Poshyvanyk et al.'s earlier work using LSI. We use the same software versions used in the previous LSI study.

For our Case Study 2, we examined Rhino. We chose Rhino because of the detailed bug information available as well as its mod-

Table 2
Software used in case studies.

Case study	Software	Versions analyzed
1	Mozilla	1.5.1, 1.6, 1.6a
	Eclipse	2.0, 2.1.3, 3.0.2
2	Rhino	1.5R5
3, 4, 5	Rhino	1.4R3, 1.5R1, 1.5R2, 1.5R3, 1.5R4, 1.5R5, 1.6R1, 1.6R2, 1.6R3, 1.6R4, 1.6R6, 1.6R7
	Eclipse	3.0, 3.0.1, 3.0.2, 3.1, 3.1.1, 3.1.2, 3.2, 3.2.1, 3.2.2, 3.3, 3.3.1, 3.3.2, 3.4

erate size, which made it possible to examine all bugs in the software.

Rhino is an open source implementation of JavaScript written in Java [30]. A total of 106 bugs across 12 versions of Rhino were analyzed. Version 1.6R5 of Rhino was not analyzed because it resulted from a re-licensing of the product and contains the same code as the previous version (1.6R4). Table 3 lists summary statistics across all versions of Rhino studied.

For our Case Study 3, our goal was to examine the scalability of our approach by analyzing a very large number of bugs across multiple iterations of two software systems. For this we analyzed large numbers of bugs from both Rhino and Eclipse. We analyzed the 106 bugs in Rhino and another 216 bugs in Eclipse. Eclipse is an open source IDE written in Java [8]. Eclipse has over 200,000 methods in the latest version analyzed. In total, we analyzed 13 versions of Eclipse and 216 bugs. Table 3 lists summary statistics for the iterations of Rhino and Eclipse studied in Case Studies 3–5.

3.2.2. Design of the case studies

For all case studies, comments, identifiers, and string-literals were extracted from the subject software systems and used to create the document collections for each system. All case studies were performed at the method level of granularity. Minimal preprocessing was performed: all words were stemmed before being added to the collection. We performed some initial simple tests prior to beginning our case studies. In these tests, stop word removal was not found to substantially improve the results. Thus, for the case studies we did not remove stop words.

Since 50 and 100 topics were found in the literature to be successful for moderate-sized document collections, we ran initial experiments on Rhino 1.5R5 using both 50 and 100 topics. One hundred (100) topics were found to yield better results (in terms of the rank of first relevant method returned for individual queries, see previous discussion of the accuracy measure); thus, we used 100 topics for all further analyses involving Rhino. In addition, we chose 100 topics for our initial analysis of Eclipse and Mozilla (Case Study 1), which worked well in the study. However, our expanded study of Eclipse suggested 100 topics were too few for that system. Recognizing that Eclipse, with over 200,000 methods, results in a much larger document collection than Rhino, we experimented with both 250 and 500 topics on two iterations of Eclipse. Five hundred topics resulted in better performance, and we continued to use 500 topics for the remaining iterations of Eclipse. For completeness, we analyzed Mozilla using 500 topics as well. Because the size of Mozilla is between the size of Rhino and the size of Eclipse (the Mozilla versions we analyzed have around 60,000 methods), we ran an additional analysis of Mozilla using 200 topics. We used parameter values, $\alpha = 50/K$ and $\beta = 0.01$, that have been recommended in the literature [35].

To assess the effectiveness of the technique, it was necessary to determine relevant elements for each bug analyzed, i.e., the actual elements fixed by developers to correct the bug. These relevant source code elements were determined by manually examining the software patch for each bug posted in the software's bug repository. Lines of code in the patch were compared to the source code files for the appropriate iteration of the software, and the name of the method(s) modified in the software patch was noted. Antoniol et al. [2] classified issues reported into bugs and non-bugs, where a "bug" referred to a corrective maintenance request and other requests ("non-bugs") referred to perfective and adaptive maintenance, refactoring, discussions, requests for help, etc. For Case Study 1, we simply employed the same bugs that were previously examined in Poshyvanyk et al. [28]. For the other case studies, all Eclipse reports examined meet the categorization of "bugs" according to Antoniol et al. [2]. All Rhino reports were examined, which included both "bugs" and "non-bugs" as categorized by Antoniol et al. [2]. The Rhino bug mappings employed are not identical to those of Eaddy et al. [7]; rather, they are the same employed by Olague et al. [24].

In all the case studies, 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 – are not useful when applied to bug localization. To determine the accuracy of the predictions for each bug, the LDA query results were compared to the relevant elements for the bug. The rank of the first relevant method returned by each query was recorded. This indicates the number of entities the programmer must examine (if following the rankings) before reaching an entity that actually needs to be corrected. For each software iteration analyzed in Case Studies 3–5, we also calculate the average of the ranks of the first relevant method returned for all bugs.

Table 3
Rhino and Eclipse summary statistics.

Rhino summary statistics					Eclipse summary statistics				
Software version	LOC	Number of classes	Number of methods (documents in corpus)	Number of unique words in corpus	Software version	LOC	Number of classes	Number of methods (documents in corpus)	Number of unique words in corpus
1.4R3	17,700	95	1173	1908	3.0	1128,036	11,596	115,677	11,342
1.5R1	27,358	126	1575	2269	3.0.1	1493,325	12,365	115,842	11,371
1.5R2	29,848	179	2125	2490	3.0.2	1309,758	12,320	115,875	11,375
1.5R3	30,235	178	2144	2508	3.1	1676,107	14,440	136,559	13,153
1.5R4	31,870	198	2312	2593	3.1.1	1598,231	14,449	136,745	13,168
1.5R5	33,609	201	2380	2591	3.1.2	1595,131	14,449	136,776	13,180
1.6R1	35,516	211	2861	2729	3.2	1892,983	17,219	158,857	14,165
1.6R2	50,853	211	2868	2743	3.2.1	1901,272	17,273	159,240	14,462
1.6R3	51,030	212	2899	2764	3.2.2	1902,632	17,280	159,307	14,475
1.6R4	51,044	212	2900	2764	3.3.	2087,131	18,895	176,646	15,178
1.6R6	58,299	260	3587	2881	3.3.1	2089,808	18,903	176,783	15,199
1.6R7	58,315	260	3587	2880	3.3.2	2090,576	18,905	176,814	15,210
					3.4	2345,450	21,294	203,315	16,608

Note that comparing the techniques by using the rank of the first related method is also how Poshyvanyk et al. compared their LSI-based technique to other techniques. As Poshyvanyk et al. previously discussed [28], the goal of these search techniques is to reduce developers' effort when searching through software for bugs. The situation is analogous to that of searching the list of results provided on a Google search. The number of links on a search page that must be examined before finding relevant information is a good measure of effort.

Once a developer has found the first relevant method, many other related methods can be found through their coding links with that method. Thus, at that point there may be no need to proceed further down the original search list in order to fix the bug. For this reason, the first relevant method is the important search criterion—searching for other methods is less useful.

Stability metrics *SDI*, SDI_{inh}, and our new $SDI_{e,ck}$ as well as the CK metrics were calculated on multiple iterations of the Rhino and Eclipse software. *SDI* and SDI_{inh} were calculated manually by comparing class names from one iteration of the software to the next iteration and counting any changes, including additions, deletions, inheritance changes, and name changes. The $SDI_{e,ck}$ metric was calculated by using the added and deleted class information previously gathered for *SDI* and SDI_{inh} and by automatically comparing CK metric values from one iteration of the software to the next to determine changed classes. CK metric values were averaged for each software iteration.

3.2.2.1. Case Study 1. For this study, the LDA-based technique was applied to three bugs in Eclipse and five bugs in Mozilla. These bugs were chosen because they are the same bugs analyzed in [28] using LSI, thus this study allows a direct comparison of LDA to LSI for bug localization. Our query formation process for this case study deviated from the basic method used elsewhere, because our initial query was taken from the LSI queries used in [28] rather than the bug title. That is, our first query was exactly the same as the original LSI query. Therefore, step two of query formation consisted of adding words related to the bug from both the bug title and summary that were not included in the LSI query as well as removing words from the LSI query that did not come from the bug title or bug summary.

Note that information retrieval methods tend to be query-dependent. That is, a query that would work well when using one information retrieval method would not always work well with another information retrieval method. In our case, a query that is targeted toward LSI would usually be a somewhat different query than one that is targeted toward LDA; since the techniques themselves differ in how they build the search space, one would expect to form queries differently for LDA than one would for LSI. Documents are represented differently in LDA and LSI, both going beyond merely term-matching. Because the documents themselves, and the relationship between the words in the document are represented differently by the two techniques, it is reasonable to expect queries would perform differently as well.

However, for completeness in this case study, which compares LDA to earlier published LSI results [28], we first ran our LDA analysis using the same queries that were originally used by Poshyvanyk et al. [28] See Case Study 1 Hypothesis 1 below. These queries are presumably more tuned toward LSI since they were chosen for the LSI-based analysis [28]. Then we performed a second test, using our queries that were targeted toward LDA on our LDA-based approach and comparing them to the earlier study which used queries targeted toward LSI on their LSI-based approach. See Case Study 1 Hypothesis 2 below.

3.2.2.1.1. Case Study 1 Hypothesis 1. This test examines whether LDA performs better than LSI over the same corpus previously published by Poshyvanyk et al. [28] and using the same queries previously published by Poshyvanyk et al. [28].

H_0: LDA does not perform better than LSI over the same corpus previously used by Poshyvanyk et al. [28], using the same queries previously published by Poshyvanyk et al. [28].
H_1: Does perform better than LSI over the same corpus previously used by Poshyvanyk et al. [28], using the same queries previously published by Poshyvanyk et al. [28].

3.2.2.1.2. Case Study 1 Hypothesis 2. This test examines whether LDA performs better than LSI over the same corpus previously published by Poshyvanyk et al. [28]; however, in this test the LDA queries are formulated such that they are more appropriate for our LDA approach, and the LSI queries were previously formulated by Poshyvanyk et al. [28] so that they were appropriate for their LSI-based approach.

H_0: LDA does not perform better than LSI over the same corpus previously used by Poshyvanyk et al. [28] when using LDA-appropriate queries for the LDA-based approach compared to LSI-appropriate queries for the LSI-based approach.
H_1: LDA does perform better than LSI over the same corpus previously used by Poshyvanyk et al. [28]] when using LDA-appropriate queries for the LDA-based approach compared to LSI-appropriate queries for the LSI-based approach.

To clarify, our procedure for Case Study 1, based on the Goal Question Metric paradigm [3], is as follows:

Goal: To examine whether the accuracy of LDA-based bug localization is better than LSI, over the same data used in previous LSI studies [28].
Question 1 Related to Goal: Case Study 1 Hypothesis 1.
Question 2 Related to Goal: Case Study 1 Hypothesis 2.
Metric for Questions 1 and 2: Rank of first relevant method returned for individual queries.

3.2.2.2. Case Study 2. For the second case study, we analyzed 35 bugs in version 1.5 release 5 (1.5R5) of the software system Rhino [30]. This formed a complete set (according to certain criteria, see below) of the Rhino bugs that were available. Bugs at all levels of severity were included in the case study, from enhancements to critical defects. All bugs that met the following criteria were extracted from Rhino's bug repository.

1. Bugs existed in source files of version 1.5R5 implementing the core and compiler functionality of the software.
2. Bugs required a method-level fix.
3. Bugs were fixed in either version 1.5R5.1 (3 bugs) or v1.6R1 (32 bugs).
4. Bugs were categorized as *resolved* or *verified* and as *fixed*, so only valid bugs that in fact had been fixed were returned.

3.2.2.2.1. Case Study 2 Hypothesis 1. This test examines whether LDA is sufficiently accurate over all bugs in a single software system. Here "sufficient accuracy" is defined as first relevant method is returned in the top 10 results.

H_0: The LDA-based bug localization technique does not possess sufficient accuracy over all the bugs available in one complete software system.
H_1: The LDA-based bug localization technique does possess sufficient accuracy over all the bugs available in one complete software system.

To clarify, our procedure for Case Study 2, based on the Goal Question Metric paradigm [3], is as follows:

Goal: To examine whether LDA is sufficiently accurate over all bugs in a single software system.
Question Related to Goal: Case Study 2 Hypothesis 1.
Metric for Question: Percentage of bug queries with first relevant method in top 10 results.

3.2.2.3. *Case Study 3.* This case study examined scalability of the bug localization technique, involving the analysis of over 300 bugs in multiple iterations of two software systems: 106 bugs are examined across 12 versions of Rhino and 216 bugs are analyzed across 13 versions of Eclipse. This study was performed at the method level. The goal for Rhino was to analyze all possible bugs in all versions. The goal for Eclipse was to analyze a manageable subset of bugs; we aimed for about 15–20 bugs per version analyzed. Bugs in Eclipse were chosen randomly from the set of appropriate bugs. The exact bugs chosen are given in [18].

Since the number of methods in this study was so very large, and is different for each system, it is difficult to define "sufficient accuracy" the same way for both systems, in terms of a small number of results. For example, 10 methods in version 1.6R7 of Rhino represent 0.3% of all methods in that version of Rhino, whereas 100 methods in Eclipse version 3.4 represent 0.05% of all methods in that version of Eclipse. For this reason we report results in terms of percentage of bugs with the first relevant method in a specified range, and we also report average rank returned across all bugs queried.

For Rhino, which is moderate sized, we define "sufficient accuracy" as before, in terms of the top 10 results returned. However, for Eclipse, we define "sufficient accuracy" in terms of the top 1000 results returned. Although 1000 methods are likely more than a developer would like to search through, it does reduce the search space considerably. For example, lacking a search tool such as ours, presumably most developers would focus their searches on individual parts of the software rather than performing a search through all parts of the software, so it is unlikely that a developer would simply search blindly through thousands of methods. However, by using our tool the developer could look for methods high on the search list that corresponded to areas of the code that the developer thought might relate to the problem. Thus this does reduce the amount of code a maintainer must search through in order to find a "broken" method.

3.2.2.3.1. *Case Study 3 Hypothesis 1.* This test examines whether LDA is sufficiently accurate over all bugs in the given software system.

H_0: The LDA-based bug localization technique does not possess sufficient accuracy over all the bugs available in the given software system.
H_1: The LDA-based bug localization technique does possess sufficient accuracy over all the bugs available in the given software system.

To clarify, our procedure for each system in Case Study 3, based on the Goal Question Metric paradigm [3], is as follows:

Goal: To examine whether LDA is sufficiently accurate over all bugs in the given software system (Rhino or Eclipse).
Question Related to Goal: Case Study 3 Hypothesis 1.
Metric for Question: Percentage of bug queries with first relevant method in top 10 results (Rhino); percentage of bug queries with first relevant method in top 1000 results (Eclipse).

3.2.2.4. *Case Study 4.* Before examining the impact of factors such as stability and complexity on the accuracy of bug localization, we must consider whether the size of the software itself affects our approach. As the software grows, there are more methods and classes in which to search for data relevant to any queries. Therefore, does the technique become less effective as the size of the software increases? This case study examines whether the accuracy of LDA-based bug localization decreases as software size increases.

To assess the impact of the software size on accuracy, the following hypothesis is examined:

3.2.2.4.1. *Case Study 4 Hypothesis 1.* This test examines whether there is an association between the size of a software system and the accuracy of LDA-based bug localization:

H_0: A relationship *does not* exist between the software size measures (Lines of Code, Number of Classes, and Number of Methods) and the accuracy of LDA-based bug localization (average rank) over all classes in the software.
H_1: A relationship *does* exist between the size measures (Lines of Code, Number of Classes, and Number of Methods) and the accuracy of LDA-based bug localization (average rank) over all classes in the software.

For all tests, an α value of 0.05 is used.

The same software versions, bugs, and queries used in Case Study 3 were used in this study. Size measures were calculated using the software metrics tool Understand for Java [34]. The results were correlated using Spearman's rank correlation because the data were not normally distributed. The Spearman's rank correlation coefficient (r_s) is a number ranging from -1 to +1. Values at either end of the extreme indicate high negative (close to -1) or positive (close to +1) correlations between two variables. A value close to zero indicates no correlation exists between the measured data. Non-normality was determined using the Kolmogorov–Smirnov test for normality.

To clarify, our procedure for each system in Case Study 4, based on the Goal Question Metric paradigm [3], is as follows:

Goal 1: To assess the impact of software size on the accuracy of bug localization.
Question related to Goal 1: Case Study 4 Hypothesis 1.
Metric for Question 1: A correlation between the average of the ranks of the first relevant method returned for each bug in each software iteration, and the Lines of Code, Number of Classes, and Number of Methods size metrics.

3.2.2.5. *Case Study 5.* In this case study, we examine the impact of stability and complexity on the accuracy of bug localization. Separately, we examine the impact of complexity on accuracy of bug localization. Accuracy is measured by the average of the ranks of the first relevant method returned for each bug in each software iteration.

To assess the impact of stability on the accuracy of bug localization, the following hypothesis is examined:

3.2.2.5.1. *Case Study 5 Hypothesis 1.* This test examines whether there is an association between the stability of source code and the accuracy of LDA-based bug localization:

H_0: A relationship *does not* exist between the design stability measures SDI, SDI_{inh}, and $SDI_{e,ck}$ and the accuracy of LDA-based bug localization (average rank) over all classes in the software.
H_1: A relationship *does* exist between the stability measures SDI, SDI_{inh}, and $SDI_{e,ck}$ and the accuracy of LDA-based bug localization (average rank) over all classes in the software.

To assess the impact of complexity on the accuracy of bug localization, the following hypothesis is examined:

3.2.2.5.2. Case Study 5 Hypothesis 2. This test examines whether there is an association between the CK metrics (measures of complexity) and the accuracy of LDA-based bug localization:

H_0: A relationship *does not* exist between the CK metrics and the accuracy of LDA-based bug localization (average rank) over all classes in the software.
H_1: A relationship *does* exist between the CK metrics and the accuracy of LDA-based bug localization (average rank) over all classes in the software.

For all tests, an α value of 0.05 is used.

The same software versions, bugs, and queries used in Case Study 3 were used in this study. Stability was calculated as outlined previously; the results were correlated using Spearman's rank correlation because the data were not normally distributed.

To clarify, our procedure for each system in Case Study 5, based on the Goal Question Metric paradigm [3], is as follows:

Goal 1: To assess the impact of stability on the accuracy of bug localization.
Question 1 Related to Goal 1: Case Study 5 Hypothesis 1.
Metric 1 for Question 1: A correlation between the average of the ranks of the first relevant method returned for each bug in each software iteration, and the SDI, SDI_{inh}, and $SDI_{e,ck}$ stability metrics.
Goal 2: To assess the impact of complexity on the accuracy of bug localization.
Question 2 Related to Goal 2: Case Study 5 Hypothesis 2.
Metric 2 for Question 2: A correlation between the average of the ranks of the first relevant methods returned for each bug in each software iteration, and the *CK* metrics.

4. Results of the case studies

4.1. Case Study 1: LDA vs. LSI for bug localization

4.1.1. Eclipse results

We ran the same queries for LDA as previously published for LSI [28] on the Eclipse LDA model. For each bug examined, Table 4 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 published LSI analysis [28]. Table 5 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. For bug fixes that involved multiple methods, such as bug #5138, the first relevant method returned may have been different for LSI and LDA. Note that the *TextDoubleClickStrategyConnector.mouseUp* method is referred to as *TextViewer.mouseUp* in [28], but the method belongs to an inner class of *TextViewer* called *TextDoubleClickStrategyConnector* and we have used the inner class name.

Table 4
Eclipse bugs analyzed with LDA/LSI query.

Software version	Bug no.	Bug title	LDA/LSI query [28] NOTE: same query used as in Poshyvanyk et al. [28]
2.1.3	5138	Double-click-drag to select multiple words does not work	Double click drag select mouse up down release text offset document position
2.0.0	31779	UnifiedTree should ensure file/folder exists	Unified tree file folder node system location
3.0.2	74149	The search words after “” will be ignored	Search query quoted token

These results show the LDA-based approach performed as well as or better than LSI when 100 topics were used and outperformed LSI for all bugs when 500 topics were used. Increasing the number of topics only slightly impacted the results for these bugs, since the 100 topic model had already performed quite well. The only difference was for bug #31779, where the LDA rank went from two (100 topics) to one (500 topics). Overall, this case study suggests LDA performs at least as well as LSI for bug localization. Note that since we used the original LSI-targeted queries from Poshyvanyk et al. [28], targeting the queries more closely toward an LDA-based analysis would only be expected to improve results. For two out of three bugs, using the same query as used for the LSI study in Poshyvanyk et al. [28], the rank for the query using the LDA-based analysis was higher than the rank for that query using the LSI-based analysis. For this reason we reject the null hypothesis (see Case Study 1 Hypothesis 1) and accept the alternate hypothesis that LDA does perform better than LSI over the same corpus previously used by Poshyvanyk et al. [28], using the same queries previously published by Poshyvanyk et al. [28].

4.1.2. Mozilla results

Each bug analyzed in Mozilla is listed in Table 6 with 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 [28]. (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 #209430, only the initial LSI query was used. A second query was formed for the other four bugs, which proved sufficient for two of the bugs. For #182192, the word *addressbook* was split into *address* and *book*. For bug #225243, *portrait*, *landscape*, and *orientation* are not in the bug title or summary, so are removed, and the word *image* from the bug summary is added. In addition, the word *postscript* is split into *post* and *script*. A third query was issued for the remaining two bugs. For bug #216154, the bug title and summary discuss a problem with clicking an anchor, so in addition to adding the words *anchor* and *target* from the bug title in step 2 of the query formation process, we added the word *link*. For bug #231474, the vocabulary of the software was examined to determine words related to multiple email attachments (discussed in the bug summary), such as *mailattachcount*. This last query demonstrates a characteristic of Mozilla that likely affects the query results: many words in the vocabulary were coded without using conventions that would allow multi-word identifiers to be easily split. Therefore, the vocabulary of Mozilla is very large. For example, though Mozilla has many fewer methods than Eclipse (60,000 vs. 200,000), it has a larger vocabulary. The LDA results for all of the queries are presented in Table 7, with the LSI rank included for comparison.

Note that for bug #225243, the study in [28] 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.

For the 100 topic model, three bugs (#182192, #209430, and #216154) resulted in a vast improvement over LSI using the same query, one bug ranked slightly worse (#225243) and one performed much worse than LSI (#231474). It is not surprising that the results were not as good in some cases using the original LSI queries; since the techniques themselves differ in how they build the search space, one would expect to form queries differently

Table 5
Comparison of LDA to LSI over Eclipse. *Note*: same queries were used as in Poshyvanyk et al. [28].

Bug no.	First relevant method	LDA rank (100 topics)	LDA rank (500 topics)	LSI rank
5138	TextDoubleClickStrategyConnector.mouseUp(LDA)/JavaStringDoubleClickSelector.doubleClicked (LSI)	2	2	7
31779	UnifiedTree.createChildNodeFromFileSystem (both LDA and LSI)	2	1	2
74149	QueryBuilder.tokenizeUserQuery (both LDA and LSI)	1	1	5

Table 6
Mozilla bugs analyzed with corresponding LDA/LSI queries.

Software version	Bug no.	Bug title	LDA query	LSI query [28]
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 7
Comparison of LDA to LSI over Mozilla.

Bug no.	First relevant method	Rank using LSI-targeted query [28] on LDA system (100 topics)	Rank using LDA targeted query for LDA-based system (100 topics)	Rank using LSI-targeted query [28] on LDA system (200 topics)	Rank using LDA targeted query for LDA-based system (200 topics)	Rank using LSI-targeted query [28] on LDA system (500 topics)	Rank using LDA targeted query for LDA-based system (500 topics)	Original LSI rank [28] using LSI-targeted Query on LSI-based system
182192	nsAbAddressCollector::CollectAddress	9	3	7	4	1	1	37
209430	nsPlaintextEditor::DeleteSelection (same query used, LSI and LDA)	9	9	1	1	1	1	49
216154	nsMailboxService::NewURI	9	4	4	1	1	1	76
225243	nsPostScriptObj::begin_page	37	9	57	19	288	29	24
231474	Root::MimeObject_parse_begin	251	4	1620	4422	13,097	53,428	18

for LDA than one would for LSI. Documents are represented differently in LDA and LSI, both going beyond merely term-matching. Because the documents themselves, and the relationship between the words in the document are represented differently by the two techniques, it is reasonable to expect queries would perform differently as well.

Using 100 topics, the LDA rank of the first relevant method returned by the newly formed LDA query for each bug is in the top five for three of the bugs and in the top 10 for all of the bugs, which is a considerable improvement over LSI. Using 500 topics, the ranks for three of the bugs improved even more, with the first relevant method returned first for both the LSI query and the updated LDA query. Localization results for the other two bugs were worse in the 500-topic LDA model than both the 100 topic model and the LSI model. Most notable is the very poor performance of bug #231474. We issued a fourth query for bug #231474 on the 500 topic model: *attach mail email message msg mime parse encode*. This query resulted in the first relevant method returned with a rank of 1185 – a vast improvement over the other 500-topic result for that bug, but still not impressive. It is likely that 500 topics are too many for this version of Mozilla. For this reason, we generated Mozilla models using 200 topics as well.

Using 200 topics, the results for two bugs, #209430 and #216154, improved over both the LSI results and the results using 100 topics in LDA. Two other bugs, #182192 and #225243 showed improved results over LSI but had slightly worse performance than the 100-topic LDA model: bug #182192 had the first relevant method ranked third using 100 topics and ranked fourth using 200 topics, and bug #225243 had the first relevant method ranked ninth for 100 topics and 19th for 200 topics. The final bug, #231474, resulted in performance worse than that of both LSI and LDA using 100 topics, but better than LDA using 500 topics.

Recall that IR systems are very dependent on query type, and normally a query for an LSI-based system should be formulated differently from a query for an LDA-based system (see previous discussion). Thus normally one would not expect to use an LSI-based query on an LDA-based system. For this reason, we consider this test primarily for the sake of completeness, and we show in gray the results from queries on the LDA-based system using the query from the original LSI-based study. Note, that for three out of five of the LSI-based queries used on an LDA-based system, the results are still better using the LDA-based system than they were on the same LSI-based query used on the original LSI-based system. However, since two out of five of the original LSI-based queries do not work better when applied to the LDA-based system (100 topic model), we feel we are unable to reject the null hypothesis for Case Study 1 Hypothesis 1. However, since three out of five of the original LSI-based queries do work better when applied to the LDA-based system, and since a fourth LSI-based query does not perform considerably worse (100 topic model) on an LDA-based system than on an LSI-based system (compared to how much better the first three queries work on an LDA-based system), we feel that it is more appropriate to consider the LDA-based system and the LSI-based system equivalent for LSI-based queries than to consider one better than the other.

For the LDA 100 topic model, for all five queries, when using an LDA-based query on an LDA system and comparing it to an LSI-based query on an LSI-based system, the LDA system performs better than does the LSI-based system. For this reason, for Case Study 1 Hypothesis 2, for the 100 topic model we reject the null hypothesis and accept the alternate hypothesis that LDA does perform better than LSI over the same corpus previously used by Poshyvanyk et al. [28] when using queries on LDA appropriate for the LDA-based approach compared to queries on LSI appropriate for the LSI-based approach.

For the LDA 200 topic model, for four out of five queries, when using an LDA-based query on an LDA system and comparing it to an LSI-based query on an LSI-based system, the LDA system performs better than does the LSI-based system. For this reason, for Case Study 1 Hypothesis 2, for the 200 topic model we also reject the null hypothesis and accept the alternate hypothesis that LDA does perform better than LSI over the same corpus previously used by Poshyvanyk et al. [28] when using queries on LDA appropriate for the LDA-based approach compared to queries on LSI appropriate for the LSI-based approach. However, we note that the fifth query has a much worse result on the LDA model than does the LSI query on the LSI model, so this is not as strong a result as for the LDA 100 topic model.

For the LDA 500 topic model, three out of five queries are notably better for an LDA-based query on an LDA-based system compared to an LSI-based query on an LSI-based system. Another query is very slightly worse. However, the fourth query is very much worse for LDA than for LSI. Thus we do not reject the null hypothesis for Case Study 1 Hypothesis 2 for the 500 topic model.

4.1.3. Discussion of results

This case study presented an analysis of five bugs in Mozilla and three bugs in Eclipse that had previously been analyzed using LSI. The results of the study show the performance of LDA for all bugs was at least as good as the LSI results. Both the use of 100 topics and 500 topics in Eclipse resulted in good performance for the three bugs analyzed, with the 500 topic models performing only slightly better. Using 100 topics proved the most effective overall for Mozilla, with all bugs having the first relevant method returned in the top 10 results. Although the performance was better for some of the bugs using 200 and 500 topics, the performance was much worse for other bugs.

For these bugs, the technique performed better in Eclipse than Mozilla, with all Eclipse bugs resulting in the first method returned with a rank of either one or two. The performance in Mozilla may be slightly worse due to the number of identifiers in Mozilla that cannot be easily split into sub-words. This results in a much larger vocabulary, which likely affects the results.

The purpose of our paper was to examine the utility of our static LDA-based bug localization technique. As we discussed in Section 1, there are two primary kinds of bug localization techniques, static techniques which examine the source code of a system, and dynamic techniques which gather information from execution traces. In this paper, we did not employ dynamic techniques, rather we focused solely on a new static technique. In the previous work by Poshyvanyk et al. [28], the results of a static LSI-based technique were reported separately, and then were combined with a dynamic technique (SPR) in a system called PROMESIR. It was not our purpose in this paper to compare our static LDA-based technique to a combined system such as PROMESIR. However, for completeness, we report the comparison in Table 8.

The combined static plus dynamic PROMESIR results over the three Eclipse bugs were comparable to those of our static LDA-based techniques. Over Mozilla, PROMESIR appears somewhat better—marginally so when compared to the 100 topic LDA model (the biggest difference is for bug number 209430). However, when PROMESIR is compared to the 500 topic static LDA model, the static LDA model is better on three but considerably worse on two. Note that all the approaches were better than SPR except for the last Mozilla bug on the 200 topic and 500 topic model.

However, the combined (static plus dynamic) PROMESIR is not a particularly good comparison to our static LDA-based technique taken alone, since the combined approach employs more and varied input data than does a static approach. An interesting future research topic would be to examine a system that employed the static LDA-based technique, instead of the static LSI-based technique, together with the dynamic SPR in new combined approach such as that of PROMESIR.

4.2. Case Study 2: Completeness of LDA-based bug localization using Rhino

Table 9 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 at both the class and method level. All words used in the final LDA query are in bold and consist of words used from the bug title (resulting from step 1 of the query formation process outlined in Section 3.2), words added from the bug summary {in braces} (resulting from step 2), and words added that did not come from either the bug title or summary (in parentheses) (following step 3).

4.2.1. Method-level results

Of the methods fixed by developers as part of the software patch for each bug, the ranks of those methods returned highest in the list of methods returned by our query are summarized in Fig. 2. At the method level, the first query proved sufficient (the first relevant method was returned in the top 10 results) for over one-half (19/35) of the Rhino bugs. The second query proved

Table 8
Comparison of LDA to LSI to PROMESIR.

Package	Bug no.	Rank using LDA targeted query for LDA-based system (100 topics)	Rank using LDA targeted query for LDA-based system (200 topics)	Rank using LDA targeted query for LDA-based system (500 topics)	Original LSI rank [28] using LSI-targeted query on LSI-based system	SPR	PROMESIR
Eclipse	5138	2	N/A	2	7	268	1
	31779	2	N/A	1	2	170	1
	74149	1	N/A	1	5	456	3
Mozilla	182192	3	4	1	37	94	2
	209430	9	1	1	49	509	1
	216154	4	1	1	76	332	6
	225243	9	19	29	24	472	6
	231474	4	4422	53428	18	494	1

Table 9
Rhino bugs analyzed (LDA query words in bold).

Bug no.	Bug title {query words added from bug summary} (additional words added)
58118	ECMA Compliance: **daylight savings time** wrong prior to **year** 1 **{offset timezone} (day)**
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 {native global js parse float bound 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 **semicolon (eol semi colon)**
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 {ecma script catch object} (obj)**
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 track stack state}**
258958	**Lookup** of excluded **objects** in **ScriptableOutputStream** doesn't traverse **prototypes/parents**
258959	**ScriptableInputStream** does not use **Context**'s application **ClassLoader** to **resolve** classes
261278	**Strict mode {missing var declarations}**
262447	NullPointerException in **ScriptableObject.getPropertyIds**
263978	Cannot **run** xalan example with Rhino 1.5 release 5 **{line number negative execute error}**
266418	Cannot **serialize** regular expressions **{regexp reg exp RE compile char set}**
274996	Exceptions with multiple **interpreters** on **stack** may lead to **ArrayIndex**OutOf**BoundsException {java wrapped}**

effective (the first relevant method was returned in the top 10 results) for 12 bugs. The final four bugs required a further refinement of the query.

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 highest-ranking method, was ranked fourth 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 mention 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 first), not Context.compileFunction (ranked 27th), to be the function needing modification.

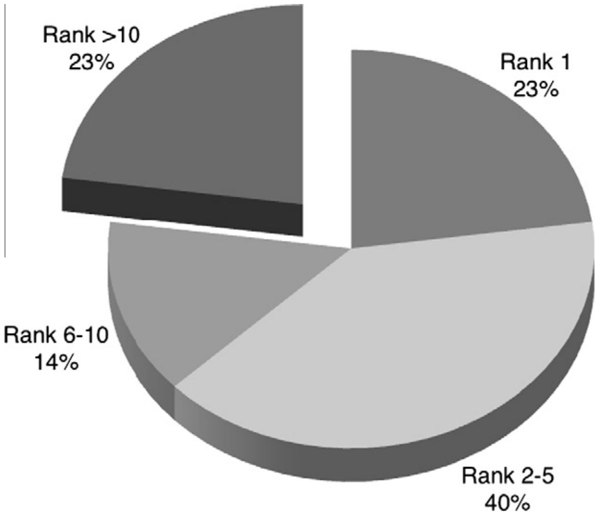

Fig. 2. Rank of first relevant method returned for Rhino 1.5R5 bugs.

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*" (step 1 of the query formation process) 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*" (step 2 of the query process) results in the same method being ranked first. Therefore, utilizing information from both the bug title and the more detailed bug summary can improve search results.

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 the following description: "*According ECMAScript standard, section 12.4, the exception name in the catch object should have DontDelete property*." Running the query "*ecma script exception name catch object don't delete obj*" results in *ScriptRuntime.newCatchScope* being returned in fifth 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 first.

4.2.2. Brief discussion of results

Overall, the LDA-based approach performed quite well. As illustrated in Fig. 2, for 77% (27/35) of the bugs analyzed the first relevant method was returned in the top 10 results and for 63% (22/35) of the bugs the first relevant method was returned in the top five results. Class-level results were even better, with more than one-half (54%) of the bugs resulting in the first relevant class returned with a rank of one, and all but three bugs resulting in the highest ranked class returned in the top 10 results. These results show the use of LDA in bug localization to be a very effective approach.

We therefore reject the null hypothesis (see Case Study 2 Hypothesis 1) and accept the alternate hypothesis that the LDA-based bug localization technique does possess sufficient accuracy over all the bugs available in one complete software system.

4.2.3. Detailed discussion of results

Like Poshyvanyk et al. [28], we consider identification of the first relevant method (or other source code entity, depending on the desired level of granularity) to be the key search criterion. Once a developer identifies the first relevant method, other related methods can be identified via coding links with the first method. Coding links include caller/callee relationships and co-location in the source code structure, for example, two methods being co-located in the same class or two classes being co-located in the same package. Thus, we use the rank of the first relevant method as the accuracy measure for our approach. Presentation of definitive empirical evidence to support our choice of accuracy measure is beyond the scope of this paper. However, in this section we provide an in-depth presentation of the coding links among methods related to three bugs from Rhino 1.5R5: one bug for which the first two relevant methods were ranked in the top 10 by our approach, one bug for which only the first relevant method was returned in the top 10 by our approach, and one bug for which the first relevant method was not returned in the top 10. We also performed a similar in-depth analysis for six additional bugs (two additional bugs from each category) and observed coding links similar to those that we present here. However, because such an analysis is space-intensive, we present a full discussion of only the three selected bugs, which are representative of different situations that can result from using our approach.

First, we present coding links among methods related to a bug for which the first two relevant methods were ranked in the top 10 by our approach. To address bug #254778, the Rhino developers modified 24 methods in five classes. Our approach ranked three of these methods in the top five (and four in the top 10). The first relevant method (ranked number 1 by our approach) is *IRFactory.createLabel*. Overall, seven of the 24 modified methods are located in the *IRFactory* class, including the three ranked highest by our approach. Those three highest-ranked methods, which are ranked 1–3 by our approach, all contain a call to *Node.Jump.setLabel*. This method was ranked 28 by our approach and is 1 of 4 modified methods located in the *Node.Jump* class. The only method in the Rhino source code that calls *Node.Jump.getLabel*, the counterpart to *Node.Jump.setLabel*, is *NodeTransformer.transformCompilationUnit_r*, which our approach ranked 52. *NodeTransformer.transformCompilationUnit_r* also calls *Node.Jump.getContinue* and *Node.Jump.setContinue*, which were ranked 117 and 157 by our approach, respectively, as well as *NodeTransformer.reportError*, which was ranked 93 by our approach. *NodeTransformer.reportError* calls *CompilerEnvirons.reportSyntaxError*, 1 of 4 modified methods located in the *CompilerEnvirons* class. Moreover, *CompilerEnvirons.reportSyntaxError* is the only method to access or modify the attribute *fromEval* (other than its getter and setter), and *Interpreter.compile*, which is ranked 1575 by our approach, is the only method in the Rhino source code that calls *CompilerEnvirons.isFromEval* (the getter for the *fromEval* attribute). The remaining four modified methods are located in the Parser class and are not explicitly linked at the code level with 20 methods already identified. However, those 20 methods implement semantic analysis related to labeled statements in JavaScript (recall that Rhino is an implementation of JavaScript), and the remaining four modified methods implement the syntactic analysis related to such labeled statements. The first of those four remaining methods, *Parser.matchLabel*, was ranked six by our approach and is one edge away from the other three remaining methods in the program call graph.

Next, we present coding links among methods related to a bug for which only the first relevant method was ranked in the top 10 by our approach. To address bug #258183, the Rhino developers modified 18 methods in four classes. Our approach ranked 1 of these methods in the top 10. The first relevant method (ranked number 3 by our approach) is *Interpreter.addExceptionHandler*. Overall, 10 of the 18 modified methods are located in the *Interpreter* class. A few of those 10 methods were not ranked in the top 100 (or even top 1000) by our approach. However, in addition to being co-located in the *Interpreter* class, all 10 of those methods can be identified by traversing the program call graph, starting from *Interpreter.addExceptionHandler* and moving one step (edge) at a time. The first step is obvious, because only one method in the *Interpreter* class, *Interpreter.generateICode*, calls *Interpreter.addExceptionHandler*. Moreover, other than the *Interpreter* class, only one class in the Rhino source code, *BodyCodegen*, contains a method that calls *Interpreter.addExceptionHandler*. The *BodyCodegen* class contains three of the 18 modified methods. Of the five remaining modified methods, two are located in the *ScriptRuntime* class and are only called by the *Interpreter.interpret* method (which is reachable as described four sentences ago). The three remaining modified methods are located in the *IRFactory* class. Similar to the previous bug, these final remaining modified methods are not explicitly linked at the code level with the 15 methods already identified. However, those 15 methods implement the interpreter logic related to exception handling in JavaScript, and the remaining three methods are the factory methods used to create the tree nodes processed by the interpreter. All three of the remaining methods are ranked in the top 15 by our approach.

Finally, we present coding links among methods related to one bug for which the first relevant method was not ranked in the top 10 by our approach. Despite the relative inaccuracy of our approach for bug #254915, it is worth noting that there are coding links among the modified methods. To address bug #254915, the Rhino developers modified six methods in three classes. The first relevant method (ranked 137 by our approach) is *ScriptRuntime.getBase*. Overall, 4 of the 6 modified methods are located in the *ScriptRuntime* class. In addition, *ScriptRuntime.getBase* calls *ScriptRuntime.bind*, which was ranked 398 by our approach. The only other method in the Rhino source code to call *ScriptRuntime.bind* is *Interpreter.interpret*, which was ranked 1314 by our approach. *Interpreter.interpret* also calls *ScriptRuntime.getBase* (the first relevant method), *ScriptRuntime.name* (ranked 812 by our approach), and *ScriptRuntime.setName* (ranked 1146 by our approach). The remaining modified method, *BodyCodegen.visitSetName*, is explicitly linked at the code level with *ScriptRuntime.setName* via the string literal “setName”, which does not appear anywhere else in the Rhino source code.

4.3. Case Study 3: Accuracy and scalability of LDA-based bug localization

The third case study provides an expanded study of the accuracy of our LDA-based approach with an analysis of over 300 bugs across 25 iterations of two software systems, Rhino and Eclipse. In this study we are primarily concerned with the scalability of our technique.

4.3.1. Rhino results

Table 10 shows the individual results for each iteration of the Rhino software. For each iteration, the number of bugs analyzed is listed along with the percentage of bugs with the first relevant method returned having a rank in the specified range. In addition, the average of the ranks of the first relevant method returned for each bug is included. The results varied quite a bit across iterations. The overall accuracy of LDA-based bug localization across all iterations of Rhino are summarized in Fig. 3.

The average of the ranks of the first relevant method returned for all bugs in each given software iteration is also listed in Table 10. These values were easily skewed by one or two bugs with a very bad rank. For example, leaving off the bug with the highest-ranked method having the worst result (rank of 1890) in Rhino 1.5R4 changes the average rank from 176 to 33.

There was considerable variation in the performance of our technique across iterations of the software. For example, in version 1.5R5, 77% of bugs had the first relevant method returned with a

Table 10
Accuracy of LDA-based bug localization per iteration of Rhino.

Version	No. bugs	Percentage of bugs			Average rank	Stdev of rank
		Rank 1	*Top 5*	*Top 10*		
1.4R3	4	25	25	25	291	474.08
1.5R1	7	43	57	57	39	52.24
1.5R2	3	0	0	33	44	40.67
1.5R3	11	27	40	50	40	61.69
1.5R4	13	23	46	62	176	517.13
1.5R5	35	23	63	77	17	46.39
1.6R1	11	9	27	36	99	233.17
1.6R2	6	33	50	50	290	654.20
1.6R3	1	100	100	100	1	0
1.6R4	12	17	25	25	1062	1107.67
1.6R6	1	0	0	0	104	0
1.6R7	2	0	0	50	252	345.07

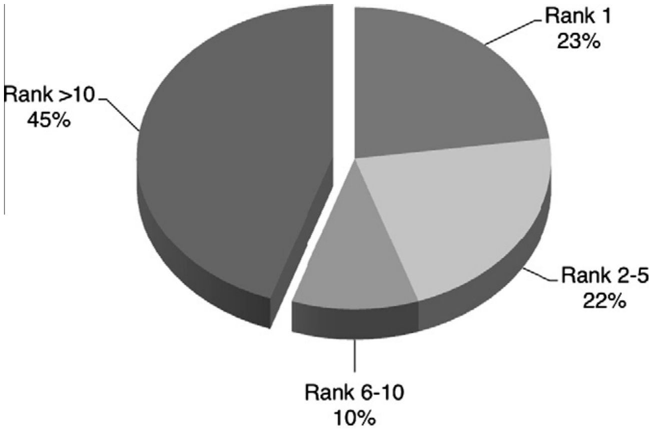

Fig. 3. Accuracy of LDA-based bug localization across 12 versions of Rhino.

rank of 10 or better. However, in version 1.6R4, only 25% of bugs had the first relevant method returned in the top 10. It is possible that doing a future in-depth analysis of those versions for which the technique exhibited poor performance vs. those with good performance may give insights into improving the technique.

Table 11 lists the queries used for a subset of the Rhino bugs. The entire list of queries used in this paper is available in [18]. Just as with the previous Rhino case study, utilizing information from the bug summary in addition to the bug title improved the results. For example, for bug #24023 in version 1.4R3, the bug title (“*Rethrowing an exception infinite loop*”) contained very little information about the bug. The query “*rethrowing exception*” resulted in the top ranked method, *Interpreter.interpret*, being ranked 316th. However, adding “*interpreter stack throw top try*” from the bug summary resulted in the same method being returned first. In addition, the other method fixed as a result of bug #24023, *Interpreter.generateICode* went from having a rank of 675 (first query) to 2 (second query).

Table 11
Subset of Rhino queries (LDA query words in bold).

Bug no.	Bug title {words added for second query} (words added for third query) *words removed for third query in italics*
24023	***Rethrowing*** an **exception** infinite loop **{interpreter stack throw top try}**
85880	Clearing of ⟨**function-name**⟩**.arguments** after function **calls** in **interpreter** mode **{print null object shell main} (-print -null)**
151337	EcmaError.**getLineSource**() returns 0 × 0 **characters {ascii end newline string char}**
191633	***Rhino*** **tokenizer** should not use ***recursion*** **{comments stack overflow error get}**
193700	Useless ***error message*** when retrieving **property** of **undefined object {parsing expression}**
201987	**Delete** “”.x throws **ClassCast*Exception*** **{code}**
205297	Infinite **recursion** with **LazilyLoadedCtor {scriptable object set by setter slot return value}**
214594	Wrong **line number** for ***exceptions*** in **while(*condition*)** in ***compiled*** mode **{statement error message print not defined}**

Consider bug #85880 from version 1.5R1. The two methods changed to correct bug #24023, *Interpreter.interpret* and *Interpreter.generateICode*, were also modified as a result of this bug, in addition to three methods in classes other than *Interpreter*. The original query for bug #85880 used only words from the bug title (“*Clearing of <function-name>.arguments after function calls in interpreter mode*”) which mentions *Interpreter*. However, running this query results in a rank of 757 for *Interpeter.interpret* and 1568 for *Interpreter.generateICode*. Refining the query in this case does not help (the ranks actually get worse). However, the constructor *NativeCall.NativeCall*, one of the other methods fixed, is ranked 10th with the first query and ranked first using the third query. Therefore, having a specific class or method name in the query does not guarantee good query performance for that method. This is consistent with the way LDA represents documents by going beyond simple terms.

For some bugs, the LDA-based technique exhibited poor performance given any of the three queries. For example, bug #201987 in Rhino 1.5R4 has the bug title “*delete* “”*.x throws ClassCastException*.” Using the query “*delete class cast exception*” results in method *ScriptRuntime.delete* ranked 183. Adding “*code*” from the bug summary results in the method being ranked 468, and removing “*exception*” results in the method being ranked 145 (the best result for this bug). Bug #214594 was even worse. The query with the best result for that bug, taken from the bug title alone (see Table 11), returned the first relevant method ranked 1890. With only 2312 documents (methods) in the collection, a maintainer would need to look through almost the entire source code base to find the relevant method. In such a case, the technique provides the maintainer with no useful information.

For some bugs with multiple methods fixed, all methods were ranked near the top of the list. For example, with bug #151337 in version 1.5R3, seven methods of class *LineBuffer* (*skipFormatChar*, *read*, *getLine*, *getOffset*, *startString*, *getString*, and *fill*) were modified due to the bug. All seven methods were returned in the top 20 results, with 5 of the 7 ranked in the top 10. Bug #191633, on the other hand, resulted in a modification to six methods of class *TokenStream* (*isJSIdentifier*, *isAlpha*, *isDigit*, *xDigitToInt*, *isJSSpace*, *getToken*). While the highest-ranked method (*getToken*) was third, only two other methods were ranked in the top 20 (rank 15 and 19). The remaining three methods were returned with ranks of 74, 290, and 1122. Great variations in the ranks of the methods fixed are perhaps an indication that some of the methods were directly related to the bugs and others were indirectly related (for example, impact of changing the other methods). In fact, examination of the patch for bug #191633 reveals there were major changes to *getToken*, which was the primary method fixed due to the bugs. Changes to the other methods occurred due to cleaning up the code (removal of unused method *isJSIdentifier*) and optimization-related updates. The LDA-based technique correctly identified *getToken* as the root of the bug.

4.3.2. Eclipse results

Fig. 4 shows overall accuracy of LDA-based bug localization for all 216 bugs in Eclipse using 100 topics. For Eclipse, we included rank values extending up to 100 and 1000, which represent about 0.05% and 0.5% of the total number of methods in Eclipse 3.4, respectively. Comparatively, 10 methods in version 1.6R7 of Rhino represent about 0.3% of all methods in that version. Twenty-two

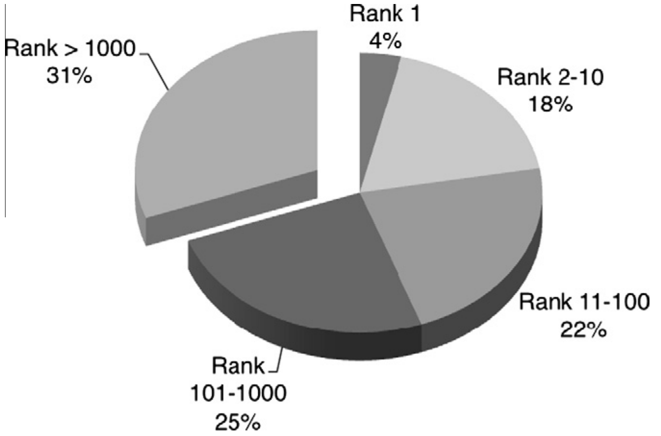

Fig. 4. Accuracy of LDA-based bug localization across 13 versions of Eclipse (100 topics).

percent of the bugs resulted in the first relevant method returned with a rank of 10 or better, 44% of bugs resulted in the first relevant method ranked 100 or better, and 69% of bugs resulted in the first relevant method with a rank of 1000 or better.

Table 12 summarizes accuracy of the technique for every iteration of the software using 100 topics. As with Rhino, average rank was easily skewed by a few bugs with poor performance. For example, in version 3.3.2, leaving off the bug with the worst performance, in which the highest-ranking method was ranked 107,068, results in an average rank of 771 instead of 7415.

The accuracy of bug localization varied across versions. Very few versions had bugs for which the first relevant method was returned with a rank of one. Iterations had as few as zero bugs to as many as 38% of bugs with the first relevant method returned in the top 10 results. In all but one iteration, at least one-half of bugs resulted in the highest-ranking relevant method returned in the top 1000 results.

Since Eclipse is much larger than Rhino, we ran LDA models for every iteration of Eclipse with 500 topics in addition to 100 topics. The overall accuracy for all Eclipse bugs using 500 topics is presented in Fig. 5. With 100 topics, 4% of bugs resulted in a relevant method ranked first; with the 500 topic model, 9% were ranked first. Forty-five percent of the bugs' highest-ranked relevant methods were ranked in the top 100 using 100 topics; with the 500 topic model, 53% of the bugs' methods ranked in the top 100. For the 500 topic model, almost three-quarters (72%) of the bugs had the first relevant method returned in the top 1000 results, which represents less than 1% of the total code.

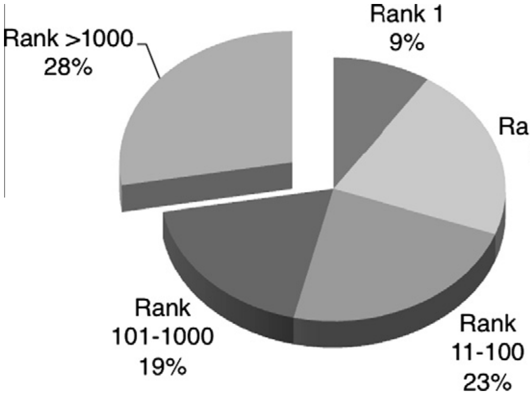

Fig. 5. Accuracy of LDA-based approach across 13 versions of Eclipse (500 topics).

Table 13 shows the results using 500 topics. Just as in the 100-topic version, the results were quite varied across iterations. Some did very well, such as 3.1.1, in which 60% of bugs had the first relevant method returned in the top 10 and 80% in the top 100. Almost all iterations had at least one-fourth of the bugs with results in the top 10, and many had at least half ranked in the top 100.

Just as the overall results improved when using 500 topics over 100 topics, the results improved for most individual iterations as well. However, that fact is not obvious when simply looking at average rank. For example, the average rank for version 3.1.1 was better in the 100 topic model (306 vs. 758). However, the percentage of bugs with the first relevant method returned with a rank in the specified ranges (Rank 1, Top 10, Top 100, and Top 1000) improved using 500 topics. The percentage of bugs with the first relevant method returned in the top 10 results went from 7% (100 topics) to 60% (500 topics) and the percentage of bugs with the first relevant method returned in the top 100 went from 50% to 80%.

Using 500 topics, not all bugs showed improvement. It appears that using 500 topics produces more extremes, i.e., the good ones got better and the bad ones got worse (often much worse). For example, in version 3.0.1 of the software, the highest-ranked method in bug #76378 went from a rank of 292 using 100 topics to nine using 500 topics. However, in the same version of the software, bug #77181 had the first relevant method returned with a rank of 401 using 100 topics and 7308 using 500 topics. Similarly, in version 3.3.1, the first relevant method returned for bug #214362 went from a rank of 63 (100 topics) to 3 (500 topics). However, the highest-ranked relevant method for bug #215843 had a rank of 5367 using 100 topics and a rank of 125,737 using 500 topics.

Table 12
Accuracy of LDA-based bug localization per each iteration of Eclipse (100 topics).

Version	No. bugs	Percentage of bugs				Average rank	Stdev of rank
		Rank 1	Top 10	Top 100	Top 1000		
3.0	15	0	33	47	60	6516	16856.28
3.0.1	14	0	14	43	86	2258	6203.80
3.0.2	6	0	0	17	33	18,111	22372.46
3.1	18	0	22	50	72	2730	6237.34
3.1.1	15	0	7	50	72	306	393.43
3.1.2	15	7	20	40	60	8366	15289.09
3.2	16	0	13	25	75	7932	27792.89
3.2.1	20	5	25	55	85	492	950.19
3.2.2	21	5	38	52	71	11,441	31158.80
3.3	21	10	43	52	71	11,234	28461.18
3.3.1	19	0	16	42	58	7566	15654.58
3.3.2	16	13	25	50	69	7415	26627.46
3.4	20	5	10	30	50	6309	11724.01

Table 13
Accuracy of LDA-based bug localization per each iteration of Eclipse (500 topics).

Version	No. bugs	Percentage of bugs				Average rank	Stdev of rank
		Rank 1	Top 10	Top 100	Top 1000		
3.0	15	20	33	47	67	1604	3122.36
3.0.1	14	7	29	64	79	758	1938.91
3.0.2	6	0	0	0	17	54,924	52612.12
3.1	18	6	28	56	78	6977	26244.40
3.1.1	15	13	60	80	80	758	2021.22
3.1.2	15	0	20	47	67	4448	8031.15
3.2	16	6	38	50	75	1796	2278.41
3.2.1	20	15	40	65	80	596	1510.43
3.2.2	21	14	33	42	76	19,562	47999.03
3.3	21	5	29	48	71	8521	30345.50
3.3.1	19	0	16	53	63	3834	29620.35
3.3.2	16	13	25	69	75	15,171	39994.19
3.4	20	15	30	50	80	5335	15915.35

Table 14 lists the queries used for a subset of the Eclipse bugs analyzed. Just as with Rhino, utilizing information found in the bug summary in addition to the bug title could result in a vast improvement in performance. For example, the bug title for bug #103089 (3.1.0) is "*Running JUnit tests constrained by package take minutes to run on jumbos.*" Using only words from the bug title, the query "*running JUnit tests package*" resulted in *TestSearchEngine.findTestCases* ranked 1614 (100 topics) and 354 (500 topics). However, adding words from the bug summary "*subpackage searching errors project java new type hierarchy*" results in the same method returned with a rank of five (100 topics) and one (500 topics).

Just as in the previous example, often the same queries (in the example, query 2) yielded the best performance in both the 100 and 500 topic models. In total, 63% of bugs in Eclipse had the best performance with the same query in both the 100 and 500 topic models. In other words, if query 2 for a bug yielded the best performance for the 100 topic model, it also yielded the best performance in the 500 topic model. For the other 37%, the query results could be quite different. For example, for bug #185254 in version 3.2.2 of Eclipse with 100 topics, query 1 results in the first relevant method returned with a rank of 164 (*BuildPluginAction.makeScripts*). However, query 2 yielded the best result, with a rank of seven for the other method modified, *ConvertedProjectsPage.createManifestFile*. Results were quite different in the 500 topic model. With query 2 (the best performer using 100 topics), the first relevant method returned (*BuildPluginAction.makeScripts*) was ranked 20. However, query 1 produced the best results using 500 topics for that bug, with *BuildPluginAction.makeScripts* ranked second.

Table 14
Subset of Eclipse queries.

Bug number	Bug title {words added for second query} (words added for third query) *words removed for third query in italics*
76378	[Progress] **Deadlock** in interaction between **ModalContext, UIThread,** and **background NotifyJob {lock listener obtain servicing sync exec request}**
77181	[**projection**] **Line numbers** not **updated** during **search** in **Java editor {perspective show dialog *folding enabled*}**
214362	For **grouped configurations, root files** are **gathered** in arbitrary **order {parts bin script assemble archive} (config)**
215843	[1.5][compiler] **Compiler *error*** with ***generic*** **covariant {super class interface}**
103089	[JUnit] **Running JUnit tests** constrained by **package** takes minutes to run on jumbos **{subpackage searching errors project java *new type hierarchy*}**
185254	Null Pointer Exception in **BuildPluginAction.makeScripts {convert project pde tools ant file create *meta inf manifest*}**

There was also a difference in which query produced the best overall result in 100 topics vs. 500 topics. Using 100 topics, query three yielded the best performance for more bugs (39%) than either query one (31%) or query two (29%). However, using 500 topics, the best result was most often achieved using query 1, which produced the best rank for the first relevant method returned for 38% of bugs. Query 2 performed the best for 29% of bugs and query 3 for 34%. Since query 3 was the top performer in the 100 topic model and query 1 in the 500 topic model, this suggests more query refinement is necessary using fewer topics for Eclipse. Using 500 topics, the first query performed the best, which suggests that refining the query is not as necessary in that case, reinforcing our decision that 100 topics were likely too few for Eclipse.

4.3.3. Discussion of results

Case Study 3 involved the analysis of over 300 bugs in 25 versions of two software systems. To our knowledge, this is the largest study of the accuracy of any IR-based approach to source code retrieval for bug localization. The case study demonstrates our LDA technique for bug localization can be effective across a large number of bugs in both a moderately sized and a large software system. Over one-half (55%) of the bugs in Rhino resulted in the top ranked method in the top 10 results returned. Almost one-quarter (23%) resulted in the first relevant method being returned with a rank of one. In Eclipse, 70% of the bugs in the 100 topic models and 72% in the 500 topic models result in the top ranked method returned with a rank of 1000 or better. Although 1000 methods are likely more than a developer would like to search through, it does reduce the search space considerably. For example, lacking a search tool such as ours, presumably most developers would focus their searches on individual parts of the software rather than performing a search through all parts of the software, so it is unlikely that a developer would simply search blindly through thousands of methods. However, by using our tool the developer could look for methods high on the search list that corresponded to areas of the code that the developer thinks might relate to the problem. Thus this does reduce the amount of code a debugger must search through in order to find a method in need of modification.

Thus for Case Study 3 Hypothesis 1, over the Rhino system, we reject the null hypothesis and accept the alternate hypothesis that the LDA-based bug localization technique does possess sufficient accuracy over all the bugs available in Rhino, given a definition of "sufficient accuracy," as before, in terms of the top 10 results returned.

For Case Study 3 Hypothesis 1, over the Eclipse system, we reject the null hypothesis and accept the alternate hypothesis that the LDA-based bug localization technique does possess sufficient accuracy over all the bugs available in Rhino, given a definition of "sufficient accuracy" in terms of the top 1000 results returned (see earlier discussion of our definition of "sufficient accuracy").

For both Rhino and Eclipse, we used the same number of topics across all iterations of the software. However, the number of methods (thus the number of documents in the document collection) changes considerably from the earliest versions analyzed to the latest versions. For example, Rhino version 1.4R3 contained 1173 documents in its document collection, while 1.6R7 had 3587. Similarly, Eclipse version 3.0 contained 115,677 documents whereas version 3.04 contained 203,315: a difference of 87,000 documents. In addition, there was an increase in the number of words in the vocabulary from version 3.0 to 3.4 of over 5000. It is likely that the same number of topics may need to be adjusted for different versions of the software, particularly for Eclipse.

4.4. Case Study 4: Software size vs. accuracy of LDA-based bug localization

The goal of the fourth case study is to examine the accuracy of our LDA-based bug localization technique vs. source code size (as measured by Lines of Code, Number of Classes, and Number of Methods). It is necessary to rule out size as a factor before we examine other factors that can potentially affect our approach.

This test using Spearman's r_s is performed at the 95% significance level ($\alpha = 0.05$). The accuracy of the approach is measured by the average of the ranks of the first relevant methods returned for all bugs across all iterations of the software being examined.

Table 15 shows correlations between the size measures and average rank for Rhino.

The correlations all have p-values > α; therefore, we cannot reject the null hypothesis, and we conclude there is no significant relationship between the software size measures and the accuracy of LDA-based bug localization as measured by average rank in Rhino. These results suggest the accuracy of the approach is not affected by the size of the Rhino software.

Table 16 shows correlations between the size measures and average rank for 100 topics and 500 topics in Eclipse.

The correlations all have p-values > α; therefore, we cannot reject the null hypothesis, and we conclude there is no significant relationship between the software size measures and the accuracy of LDA-based bug localization as measured by average rank in Eclipse, using both 100 and 500 topics. These results suggest the accuracy of the approach is not affected by the size of the Eclipse software.

4.4.1. Discussion of results

In both Rhino and Eclipse, no significant relationship was found between either Lines of Code, Number of Classes, or Number of Methods, and the accuracy of the LDA-based bug localization technique. This suggests the accuracy of the bug localization technique is not affected by the size of the system design. Thus, we can now examine other factors that may have an impact on accuracy.

4.5. Case Study 5: Stability vs. accuracy of LDA-based bug localization

The goal of the fifth case study is to examine the accuracy of our LDA-based bug localization technique vs. source code stability (as measured by *SDI*, SDI_{inh}, $SDI_{e,ck}$, and CK metrics). Our assumption is that this study is necessary since changeable or unstable code very often could result in poor or missing semantic tokens, which therefore could affect the results of our approach (see earlier discussion in Introduction).

First, Case Study 5 Hypothesis 1 (previously stated) is tested.

This test using Spearman's r_s is performed at the 95% significance level ($\alpha = 0.05$). The accuracy of the approach is measured by the average of the ranks of the first relevant methods returned for all bugs across all iterations of the software being examined.

Table 17 shows correlations between *SDI* and average rank for Rhino.

One might expect the accuracy of bug localization to decrease in unstable software versions. However, all of the correlations resulted in p-values > α; therefore, we cannot reject the null hypothesis, and we conclude there is no significant relationship between the *SDI* metrics and the accuracy of LDA-based bug localization as measured by average rank in Rhino. This leads us to believe the accuracy of the approach is not affected by stability of the software for Rhino. Table 18 shows correlations between *SDI* and average rank in Eclipse for 100 and 500 topics.

As in Rhino, all correlations resulted in p-values > α; therefore, we cannot reject the null hypothesis, and we conclude there is no significant relationship between the *SDI* metrics and the accuracy of LDA-based bug localization as measured by average rank in Eclipse. These results lead us to believe the accuracy of the approach is not affected by the stability of the software for Eclipse, using either 100 topics or 500 topics.

4.5.1. Relationship between average rank and CK metrics

This study tests whether a relationship exists between the accuracy of LDA-based bug localization and the CK metrics. Specifically, Hypothesis 2 is tested in this section.

Table 15
Software size measures vs. average rank in Rhino.

Software size measures (Rhino)	Rhino average rank	
	r_s	p-Value
Lines of Code	0.203	0.527
Number of Classes	0.151	0.639
Number of Methods	0.196	0.541

Table 16
Software size metrics vs. average rank in Eclipse.

Software size measures (Eclipse)	Eclipse average rank			
	100 Topics		500 Topics	
	r_s	p-Value	r_s	p-Value
Lines of Code	0.038	0.900	0.269	0.374
Number of Classes	0.080	0.796	0.237	0.436
Number of Methods	0.159	0.603	0.308	0.306

Table 17
Stability metrics vs. average rank in Rhino.

Stability metrics	Average rank	
	r_s	p-Value
SDI	−0.437	0.179
SDI_{inh}	−0.309	0.355
$SDI_{e,ck}$	−0.391	0.235

Table 18
Stability metrics vs. average rank in Eclipse.

Stability metrics	Eclipse average rank			
	100 Topics		500 Topics	
	r_s	p-Value	r_s	p-Value
SDI	−0.104	0.746	−0.056	0.863
SDI_{inh}	−0.077	0.812	−0.084	0.795
$SDI_{e,ck}$	−0.126	0.697	−0.084	0.795

This test is performed at the 95% significance level ($\alpha = 0.05$). The CK metrics were averaged across all classes for each iteration of Rhino and Eclipse. Table 19 shows the correlations between the average of each CK metric and average rank for Rhino.

One might expect the accuracy of bug localization to decrease as measures that tend to increase with code complexity, such as WMC, CBO, RFC, and LCOM, increase. In addition, once might expect an increase in such metrics as DIT or NOC, which could indicate refactoring has occurred and reuse is being utilized, would tend to increase accuracy. For example, if the code is very complex (reflected by high values of WMC, CBO, RFC, LCOM), it is less likely that the comments provided completely and fully document the code. Thus the bug could have occurred due to a problem in the code but the associated comment might not reflect the aspects of the code that caused the bug. Alternately, when reuse has occurred (reflected by higher values of DIT or NOC) it means that the code is more likely to have been well documented, since if it were not well documented and well understood it is unlikely to have been reused. Thus, the likelihood of our approach working well is higher.

However, as shown in Table 19, all of the correlations resulted in p-values > α. Thus, we cannot reject the null hypothesis and we conclude a significant relationship does not exist between the accuracy of the bug localization technique and the CK metrics in Rhino. These results imply the CK metrics would not be good indicators of the accuracy of the technique.

Table 20 shows the correlations between the average of each CK metric and average rank for 100 and 500 topics in Eclipse. For Eclipse using 100 topics and Eclipse with 500 topics, all of the correlations resulted in p-values > α. Again, we cannot reject the null hypothesis given such high p-values, and we conclude no significant relationship exists between the CK metrics and the accuracy of the bug localization technique in Eclipse.

4.5.2. Discussion of results

In both Rhino and Eclipse, no significant relationship was found between the *SDI* metrics and the accuracy of the LDA-based bug localization technique. This suggests the accuracy of the bug localization technique is not affected by the stability of the system design. Also, the CK metrics were not significantly correlated with average rank. This lack of significant correlation implies the CK metrics would not be good indicators of the accuracy of the technique. Overall, these studies tend to suggest the LDA bug localization technique should not be affected by unstable or complex code.

Table 19
Average CK metrics vs. average rank in Rhino.

Average CK metrics	Average rank	
	r_s	p-Value
WMC	0.147	0.649
DIT	0.133	0.681
NOC	−0.233	0.466
CBO	0.228	0.477
RFC	0.113	0.727
LCOM	0.308	0.330

Table 20
Stability metrics vs. average rank in Eclipse.

Average CK metrics	Eclipse average rank			
	100 Topics		500 Topics	
	r_s	p-Value	r_s	p-Value
WMC	−0.297	0.324	0.028	0.929
DIT	0.115	0.707	0.220	0.471
NOC	0.198	0.517	0.275	0.363
CBO	−0.231	0.448	0.077	0.803
RFC	−0.258	0.394	0.027	0.929
LCOM	−0.308	0.306	−0.281	0.353

5. Conclusions and future research

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.

The first case study examined the same bugs in Mozilla and Eclipse previously analyzed using LSI [28]. 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 10 results returned. For the LDA approach, the user query resulted in all eight (100%) of the bugs ranked in the top 10 when using 100 topics for Mozilla and both 100 and 500 topics for Eclipse. 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 considerably 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 were 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 10 for 77% of the bugs analyzed and a relevant class in the top 10 for 91% of the bugs. We extended our study of the accuracy of the technique by analyzing 322 bugs across 25 versions of two software systems, which to our knowledge, is the largest study performed on any IR-based bug localization technique. This study also showed LDA to be a viable technique for bug localization; in over one-half of the bugs analyzed, the search space (number of methods a debugger must look though) in Eclipse was reduced to less than 0.05% of the total methods and in Rhino was reduced to less than 0.5%.

From our study, we are able to conclude there is no significant relationship between stability and the accuracy of our approach, based on the visual examination of scatter plots together with the lack of significant correlations between the two variables. Future studies may be done to determine the effect of other factors on the bug localization technique. For example, as modifications are made to source code between software releases, it is possible the LDA model becomes less effective. Therefore, it may be prudent to build models on intermediate versions of the software when there are considerable changes between releases.

An interesting follow-up study would involve augmenting our approach in the following way. After querying the LDA model and obtaining the ranked list of methods, select the top n methods based on some threshold value. Next, construct a new LDA model using a smaller number of topics and only the documents for the top n methods. Query this new LDA model with the original query to obtain a re-ranking of the n methods. It is possible that such a strategy could improve the performance of our LDA-based technique, though the cost of the strategy might be prohibitive.

Another future direction for our work is to further study how best to choose the number of topics when creating an LDA model. In addition to varying the number of topics when creating an LDA model, such studies would involve varying the size and application domain of the software being studied. Finally, it would be interesting to study the degree to which the technical background of a user affects his/her ability to write queries that provide useful results.

For example, how well can beginning programmers make use of our approach when compared to senior programmers?

Acknowledgements

This work was funded in part by the National Aeronautics and Space Administration under Grants NAG5-12725 and NCC8-200, and by the National Science Foundation under Grants CCF-0915403 and CCF-0915559.

References

[1] M. Alshayeb, W. Li, An empirical study of system design instability metric and design evolution in an agile software process, J. Syst. Softw. 74 (3) (2005) 269–274.

[2] G. Antoniol, K. Ayari, M. Di Penta. Is it a bug or an enhancement? A text-based approach to classify change requests, in: Proc. Conf. of the Center for Advanced Studies on Collaborative Research: Meeting of Minds, Richmond Hill, CA, October, 2008, pp. 1–15.

[3] V.R. Basili, H.D. Rombach, The TAME project: towards improvement-oriented software environments, IEEE Trans. Softw. Eng. 14 (6) (1988) 758–773.

[4] D.M. Blei, A.Y. Ng, M.I. Jordan, Latent Dirichlet allocation, J. Mach. Learn. Res. 3 (2003) 993–1022.

[5] S.R. Chidamber, C.F. Kemerer, A metrics suite for object-oriented design, IEEE Trans. Softw. Eng. 20 (6) (1994) 476–493.

[6] S.C. Deerwester, S.T. Dumais, T.K. Landauer, G.W. Furnas, R.A. Harshman, Indexing by latent semantic analysis, J. Am. Soc. Informat. Sci. 41 (6) (1990) 391–407.

[7] M. Eaddy, T. Zimmermann, K. D Sherwood, V. Garg, G.C. Murphy, N. Nagappan, A. Aho, Do crosscutting concerns cause defects?, IEEE Trans Softw. Eng. 34 (4) (2008) 497–515.

[8] Eclipse Home Page. <http://www.eclipse.org>.

[9] B. Fluri, M, Wursch, H.C. Gall, Do code and comments co-evolve? On the relation between source code and comment changes, in: Proc. 14th Working Conf. on Reverse Engineering, Vancouver, BC, October 2007, pp. 70–79.

[10] GibbsLDA++. <http://www.gibbslda.sourceforge.net/>.

[11] M. Girolami, 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.

[12] T.L. Griffiths, M. Steyvers, Finding scientific topics, Proc. Nat. Acad. Sci. 101 (1) (2004) 5228–5235.

[13] 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.

[14] A. Kontostathis, Essential dimensions of latent semantic indexing (LSI), in: Proc. 40th Hawaii International Conference on System Sciences, Big Island, Hawaii USA, January 2007, pp. 1–8.

[15] A. Kuhn, S. Ducasse, T. Gîrba, Enriching reverse engineering with semantic clustering, in: Proc. 12th Working Conf. on Reverse Engineering, Pittsburg, PA, November 2005, pp. 133–142.

[16] W. Li, L. Etzkorn, C. Davis, J. Talburt, An empirical study of object-oriented system evolution, Informat. Softw. Technol. 42 (2000) 373–381.

[17] E. Linstead, P. Rigor, S. Bajracharya, C. Lopes, 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] S.K. Lukins, Source Code Retrieval for Bug Localization Using Latent Dirichlet Allocation and its Relationship to Stability of Agilely Developed Software, Doctoral Dissertation, University of Alabama in Huntsville, 2009.

[19] S.K. Lukins, N.A. Kraft, L.H. Etzkorn, Source code retrieval for bug localization using latent Dirichlet allocation, in: Proc. 15th Working Conf. on Reverse Engineering, Antwerp, Belgium, October 2008, pp. 155–164.

[20] A. Marcus, J.I. Maletic, A. Sergeyev, Recovery of traceability links between software documentation and source code, Int. J. Softw. Eng. Knowledge Eng. 15 (5) (2005) 811–836.

[21] G. Maskeri, S. Sarkar, 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.

[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] Mozilla Home Page. <http://www.mozilla.org>.

[24] H. Olague, L. Etzkorn, S. Gholston, S. Quattlebaum, Empirical validation of three metrics suites to predict fault-proneness of object-oriented classes developed using highly iterative or agile software development processes, IEEE Trans. Softw. Eng. 33 (6) (2007) 402–419.

[25] H.M. Olague, L.H. Etzkorn, W. Li, G. Cox, Assessing design instability in iterative (Agile) object-oriented projects, J. Softw. Maint. Evolut.: Res. Pract. 18 (2006) 237–266.

[26] Porter Stemming Algorithm. <http://tartarus.org/~martin/PorterStemmer/>.

[27] D. Poshyvanyk, Y.G. Guéhéneuc, A. Marcus, G. Antoniol, 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.

[28] D. Poshyvanyk, Y.G. Guéhéneuc, A. Marcus, G. Antoniol, V. Rajlich, Feature location using probabilistic ranking of methods based on execution scenarios and information retrieval, IEEE Trans. Softw. Eng. 33 (6) (2007) 420–432.

[29] D. Poshyvanyk, 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.

[30] Rhino. <http://www.mozilla.org/rhino/>.

[31] P. Roden, An Examination of Stability and Reusability in Highly Iterative Software, Ph.D. Dissertation, University of Alabama in Huntsville, 2008.

[32] C. Stein, G. Cox, L. Etzkorn, S. Gholston, S. Virani, P. Farrington, D. Utley, J. Fortune, Exploring the relationship between cohesion and complexity, J. Comput. Sci. 1 (2) (2005) 137–144.

[33] M. Steyvers, T. Griffiths, Probabilistic topic models, in: T. Landauer, D. McNamara, S. Dennis, W. Kintsch (Eds.), Handbook of Latent Semantic Analysis, Lawrence Erlbaum Associates, 2007.

[34] Understand Source Code Analysis and Metrics. <http://www.scitools.com/products/understand/>.

[35] X. Wei, B. Croft, LDA-based document models for ad-hoc retrieval, in: Proc. 29th Annu. Int. ACM SIGIR Conf. on Research & Development on Information Retrieval, WA, USA, 2006, pp. 178–185.