A Dissertation

On

SOFTWARE BUG LOCALIZATION USING TOPIC MODELS

Submitted in partial fulfillment of the requirement
for the award of degree of
MASTER OF TECHNOLOGY
Computer Technology and Application (CTA)

Submitted By:

TANU SHARMA

2K10/CTA/24

Under the Guidance of:
Dr. Kapil Sharma

Associate Professor
Computer Engineering Department
Delhi Technological University

DEPARTMENT OF COMPUTER ENGINEERING

DELHI TECHNOLOGICAL UNIVERSITY

(*Formerly Delhi College of Engineering)*

Bawana Road, Delhi-110042

MAY, 2013

CERTIFICATE

This is to certify that **TANU SHARMA** has carried out the work presented in this thesis report entitled, **“Software Bug Localization using Topic Models”**, under my supervision. The report embodies result of work and studies carried out by her and the contents of the thesis do not form the basis for the award of any other degree to the candidate or to anybody else.

Dr. Kapil Sharma
Associate Professor
Computer Engineering Department
Delhi Technology University, Delhi

ACKNOWLEDGMENT

First and foremost, praises and thanks to the God, the Almighty, for His showers of blessings throughout my M.Tech to get it completed successfully.

I want to express my sincere thanks to my HOD, Dr. Daya Gupta, Department of Computer Engineering, Delhi Technological University, Delhi for providing well equipped infrastructure support. I would like to express my deep and sincere gratitude to my project mentor Dr. Kapil Sharma, Associate Professor, Computer Engineering Department, Delhi Technological University, Delhi for giving me the opportunity to do research and providing invaluable guidance and constant encouragement throughout the project.

I am thankful to my elder brother, Tapan Sharma, Senior Advisory Software Engineer, Pitney Bowes, Noida for helping me in implementing and completing this work by his valuable industry experience. I am extremely grateful to my family for their love, prayers, caring and sacrifices for educating and preparing me for my future and for not letting me down at the time of crisis and showing me the silver linings in the dark cloud.

This thesis would have not been completed without the constant support of my best friends. Finally, I thank all the faculty and staff members of Bhagwan Parshuram Institute of Technology, Rohini, Delhi for extending a helping hand at every juncture of need.

TANU SHARMA
2K10/CTA/24
M.Tech (Computer Technology and Applications)
Department of Computer Engineering
Delhi Technology University, Delhi

ABSTRACT

Bug localization is a process of identifying the specific file of source code that is faulty and needs to be modified to fix the bug. Due to the increasing size and complexity of current software applications, automated solutions for bug localization can significantly reduce human effort and software development/maintenance cost. In this research work, bug localization has been performed using topic model of Information Retrieval. Pachinko Allocation Model (PAM) has been applied for the first time in bug localization. In this research work, PAM model of source code is built first. This model is then queried for locating bugs. The bug reports are considered as a query for the system for which files containing bugs need to be identified. This query is used by Inference engine to produce ranked list of files from source code. The top-ranked files are the one most likely to require modification to correct the bug. This work performs analysis and comparison of PAM and Latent Dirichlet Allocation (LDA) models based approach for bug localization using MALLET library in java. This library has been extended to incorporate PAM based bug localization using proposed Inference engine. For evaluating the performance of PAM and LDA based approach, the datasets downloaded from two open source projects i.e. Rhino and ModeShape have been used in this work. In case of Rhino dataset, for one bug report only 10% of dataset is needed to be reviewed. In case of ModeShape dataset, for one bug report only 1.5 % of dataset is needed to be reviewed. It has been observed that the bug localization technique using PAM model gives promising results as compared to LDA model.

CONTENTS

LIST OF FIGURES

LIST OF TABLES

LIST OF ABBREVIATIONS

AP	Average Precision
AOP	Aspect-Oriented Programming
$ccLDA$	Cross-Collection Topic Models
DAG	Directed Acyclic Graph
FRF	First Relevant File
$HLDA$	Hierarchical Topic Models
IR	Information Retrieval
LDA	Latent Dirichlet Allocation
$LLDA$	Labeled LDA
LSI	Latent Semantic Indexing
MAP	Mean Average Precision
$MALLET$	Machine Learning for Language Toolkit
NLP	Natural Language Processing
PAM	Pachinko Allocation Model
$pLSI$	Probabilistic Latent Semantic Indexing
$PLTM$	Polylingual Topic Model
RTM	Relational Topic Model
RTC	Relational Topic-based Coupling
$sLDA$	Supervised Topic Model
SVD	Singular Value Decomposition

LIST OF SYMBOLS

A	Term-Document matrix
C	Corpus
d_i	A Document
M	Total number of relevant files for a bug
N_d	Number of terms in a document
P_i	Precision at ith relevant file retrieved
V	Vocabulary
w_i	Term (word)
z_i	A Topic
α	Smoothing parameter for document topic distributions
β	Smoothing parameter for topic term distributions
φ	Word-Topic Probability Distribution
θ	Topic-Document Probability Distribution
$\theta_r^{(d)}$	A multinomial distribution over super topics
$\theta_{t_i}^{(d)}$	A multinomial distribution over sub topics
$g_i(\alpha_i)$	Dirichlet distribution associated with topic t_i.

CHAPTER 1

INTRODUCTION

1 INTRODUCTION

Due to the increasing size and complexity of software systems, efficient bug localization is required. This chapter introduces the debugging process and the objectives and contributions of this thesis. Chapter wise thesis coverage has been summarized at the end.

1.1 Background of the study

In today's era, software industries are competing with quality of the software which depends upon the sound software testing phase. Most of the software contains some bugs after being released, so it is most challenging to localize bug automatically and fix them before release. A software quality factor is a non-functional requirement for a software program which is not called up by the customer's contract, but nevertheless is a desirable requirement which enhances the quality of the software program.

In large and complex software systems; software aging, poor-documentation and developer mobility makes software project hard to understand for software developers (Lukins, Kraft, & Etzkorn, 2008). This may slow down software project progress and may increase overall software maintenance cost. In order to bring down the overall resource consumption of corrective software maintenance it is required to empower software developers with tools and techniques that can facilitate them in debugging and bug fixing. Debugging is a methodical process of finding and reducing the number of bugs, or defects, in a computer program or a piece of electronic hardware, thus making it behave as expected. Debugging tends to be harder when various subsystems are tightly coupled, as changes in one may cause bugs to emerge in another. The first Software bug was seen by Grace Murray Hopper in year 1947 on Harvard University Mark II Aiken Relay Calculator (a primitive computer). It starts from possibly unknown initial conditions and the end cannot be predicted, except statistically and the duration of debugging, cannot be constrained. It demands intuitive leaps, conjectures, experimentation, intelligence and freedom which are impossible without detailed design knowledge. Debugging process of a program is describes as a chain of three steps (Katz & Anderson, 1987):

i. Finding the potential location of bug
ii. Fixing the bug
iii. Testing the program

Bug fixing is a complex task as it requires understanding of bug and source code. Bug fixing task consists of various sub-task such as understanding the bug, locating the cause of bug and finally fixing it. In most of the bug fixing cases, locating cause of bug (bug localization) consumes most of the developer's time (Chang, Bertacco, & Markov, 2005). Basically two types of techniques are used for performing bug localization, one is static and another is dynamic. Static bug localization techniques work on the source code or a static model of the source code, while dynamic bug localization techniques work on execution traces. In static bug localization, neither operational software nor a test case is required. While dynamic bug localization techniques, requires the working software and also the test case that triggers the bug. The major drawback of dynamic technique is that a program or software developed for locating bugs cannot be made language independent. This work focuses on the task of bug localization (locating the bugs in the source code) using topic models of Information Retrieval (IR).

1.2 Research Objective

The main objectives of this work are:

- To propose a topic model based approach for bug localization that can perform better than the existing approaches.
- To automate bug localization process irrespective of the programming language used in the source code where bug has to be located.
- To propose an approach for bug localization in which bugs can be located in early stages of development also.

In this work, PAM topic model of IR has been used for performing bug localization.

1.3 Contribution of research work

The major contributions of this work are:

- The proposed approach for performing bug localization is independent of the programming language of the source code.

- Bug localization can be easily performed in the initial stages of development and no test case or test suite is required for performing bug localization.
- For locating bugs only 1.5 % of ModeShape source code is needed to be reviewed. While 10% of Rhino source code is required to be reviewed for locating the cause of bugs.
- PAM based approach for bug localization performs better than LDA(Latent Dirichlet Allocation) based approach in terms of both MAP(Mean Average Precison) and First relevant file method.
- For Rhino dataset, the value of MAP is 0.157 and 0.202 using LDA and PAM based approaches respectively.
- For ModeShape dataset, the value of MAP is 0.100 and 0.142 using LDA and PAM based approaches respectively.

1.4 Thesis Outline

The next chapter provides the background of bug localization and various topic models in Information Retrieval. Chapter 3 discusses the details of the topic models and methodology used for locating bugs in this work. Chapter 4 discusses the datasets used in this work and the various evaluation metrics used for comparing and evaluating the result. This chapter also describes the experimental set up and simulation. In chapter 5, analysis of result has been done. Conclusion and future scope of this work has been included at the end.

CHAPTER 2

LITERATURE SURVEY

2 LITERATURE SURVEY

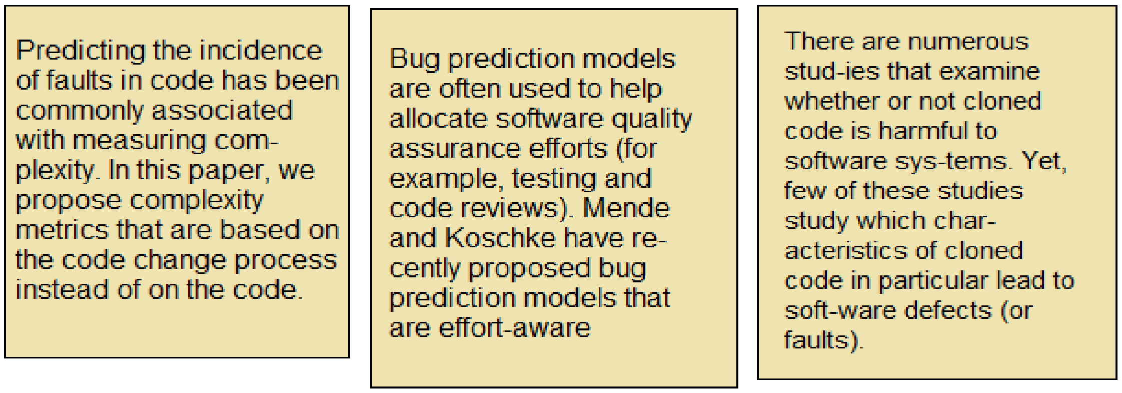
Bug localization is a process of mapping a bug back to the code that might have caused it. Performing bug localization is relatively time consuming and costly. For this reason, many techniques are available for facilitating the task of bug localization. This chapter provides background for various topic models in Information Retrieval and the task of bug localization. Details of topic models and their applications are discussed first followed by the details about bug, bug tracking system and bug localization.

2.1 Introduction to Topic Modeling

A topic model (or latent topic model or statistical topic model) refers to a model designed to automatically extract topics from a corpus of text documents (Anthes, Dec,2010) (Blei & Lafferty, Topic models, 2009) (Steyvers & Griffiths, 2007). A collection of terms that co-occur frequently in the documents of the corpus, for example {mouse, click, drag, right, left} and {user, account, password, authentication} makes a topic. Topic models are algorithms for discovering the main themes that pervade a large and otherwise unstructured collection of documents. Topic models can organize the collection according to the discovered themes.

Topic modeling is a suite of algorithms that aim to discover and annotate large archives of documents with thematic information (Blei, David M., 2012). Topic modeling algorithms are statistical methods that analyze the words of the original texts to discover the themes that run through them, how those themes are connected to each other, and how they change over time.

Topic modeling algorithms do not require any prior annotations or labeling of the documents, the topics emerge from the analysis of the original texts. Due to the nature of language use, the terms that constitute a topic are often semantically related (Blei, Ng, & Jordan, 2003)



Predicting the incidence of faults in code has been commonly associated with measuring com-plexity. In this paper, we propose complexity metrics that are based on the code change process instead of on the code.

Bug prediction models are often used to help allocate software quality assurance efforts (for example, testing and code reviews). Mende and Koschke have re-cently proposed bug prediction models that are effort-aware

There are numerous stud-ies that examine whether or not cloned code is harmful to software sys-tems. Yet, few of these studies study which char-acteristics of cloned code in particular lead to soft-ware defects (or faults).

Figure 1: Corpus of three documents

Figure 1 shows the corpus of three documents. The terminology used in topic model is explained below with reference to this figure.

a) **Term (word)** w_i**:** A string of one or more alphanumeric characters. In figure 1, there are total of 101 terms. For example, predicting, bug, there, have, bug and of are all terms. Terms might not be unique in a given document.

b) **Document** d_i: An ordered set of N terms, $w_1, w_2 \ldots\ldots w_N$. In above figure, there are three documents : d_1, d_2 and d_3. d_1 has N = 34 terms, d_2 has N = 35 terms, and d_3 has N = 32 terms

c) **Corpus :** An ordered set of n documents $d_1 \ldots . d_n$. In figure 1, there is one corpus, which consists of n = 3 documents: d_1, d_2 and d_3.

d) **Vocabulary :** The unordered set of m unique terms that appear in a corpus. In figure 1, the vocabulary consists of m = 71 unique terms across all three documents: code, of, are, that, to, the, software, …

e) **Term-document matrix *A*:** An m x n matrix whose ith, jth entry is the weight of term w_i in document d_j.

$$A = \begin{array}{l|ccc|} & d_1 & d_2 & d_3 \\ \hline \text{code} & 3 & 1 & 2 \\ \text{of} & 2 & 0 & 2 \\ \text{arc} & 1 & 2 & 1 \end{array}$$

In figure, the term code appears three times in document d_1 and two times in document d_2.

2.2 Topic Models in Information Retrieval

Various topic models in Information Retrieval have been discussed below

2.2.1 Information Retrieval

An Information Retrieval system is a software program that stores and manages information on documents, often textual documents but possibly multimedia. The system assists users in finding the information they need. It does not explicitly return information or answer questions. Instead, it informs on the existence and location of documents that might contain the desired information. Some suggested documents will, hopefully, satisfy the user's information need. These documents are called relevant documents (Hiemstra, 2009).

The goal of any IR system is to identify documents relevant to a user's query. In order to do this, an IR system must assume some specific measure of relevance between a document and a query, i.e., an operational definition of a relevant document with respect to a query. A fundamental problem in IR research is thus to formalize the concept of relevance; a different formalization of relevance generally leads to a different retrieval model (Zhai, October,2007).

Since the IR based approaches were originally developed for natural languages, there exist some challenges when one tries to adapt them to retrieval from software libraries. The two key challenges are: vocabulary mismatch and the lack of availability of good evaluation datasets. Vocabulary mismatch occurs when a query contains a word that was not seen before in the documents used for model construction. For the case of software libraries, the vocabulary mismatch problem arises from the use of abbreviations and concatenations of variable names and identifiers by the developers at the time of code development. Such words are called hard-words. The words used in a query may carry the same semantic intent as portions of the hard-words, but may not match them structurally.

Table 1: Terminologies in Information Retrieval and Bug Localization

Terminology in IR	**Terminology in Bug Localization**
Document	Source files of the software library
Query	Bug report and/or its textual description
Terms	Identifier names
Retrieval	Bug localization
Index	Source library

2.2.2 Topic Models

Topic models were originally developed in the field of natural language processing (NLP) and IR as a means of automatically indexing, searching, clustering and structuring large corpora of unstructured and unlabeled documents. Using topic models, documents can be represented by the topics within them, and thus the entire corpus can be indexed and organized in terms of this discovered semantic structure. Topic models enable a low-dimensional representation of text, which uncovers latent semantic relationships and allows faster analysis on text (Thomas S. W., 2012).

A variety of probabilistic topic models have been proposed to analyze the content of documents and the meaning of words (Blei, Ng, & Jordan, 2003) (Hoffman, 1999) (Blei & Lafferty, Topic models, 2009). These models all use the same fundamental idea, that a document is a mixture of topics but make slightly different statistical assumptions.

Authors (Deerwester, Dumais, Landauer, Furnas, & Harshman, 1990) proposed Latent Semantic Indexing (LSI), an indexing and retrieval model that used a mathematical technique called singular value decomposition (SVD) to identify patterns in the relationships between the terms and concepts contained in an unstructured collection of text. LSI is based on the principle that words that are used in the same contexts tend to have similar meanings. Hofmann (Hoffman, 1999) introduced the probabilistic topic approach to document modeling in his Probabilistic Latent Semantic Indexing method (pLSI; also known as the aspect model).

Latent Dirichlet Allocation (LDA), a popular probabilistic topic model has been proposed by authors (Blei, Ng, & Jordan, 2003). LDA has largely replaced PLSI. One of the reasons it is so popular is because it models each document as a multi-membership mixture of K corpus-wide topics, and each topic as a multi membership mixture of the terms in the corpus vocabulary. This means that there are a set of topics that describe the entire corpus, each document can contain more than one of these topics, and each term in the entire repository can be contained in more than one of these topic. Hence, LDA is able to discover a set of ideas or themes that well describe the entire corpus (Blei, David M., 2012).

Several variants of LDA have been proposed. All of these variants apply additional constraints on the basic LDA model in some way.

Authors (Blei, Griffiths, Jordan, & Tenenbaum, 2004) proposed Hierarchical Topic Model (HLDA) that discovers a tree-like hierarchy of topics within a corpus, where each additional level in the hierarchy is more specific than the previous. For example, a super-topic “user interface” might have sub-topics “toolbar” and “mouse events”.

Authors (Rosen-Zvi, Griffiths, Steyvers, & Smyth, 2004) proposed Author-Topic Model. The author-topic model considered one or more authors for each document in the corpus. Each author is then associated with a probability distribution over the discovered topics. For example, the author Stephen King would have a high probability with the “horror” topic and a low probability with the “dandelions” topic.

Authors (Li & McCallum, 2006) introduced Pachinko Allocation Model (PAM) that provided connections between discovered topics in an arbitrary directed acyclic graph.

Authors (Blei & McAuliffe, Supervised topic models, 2008) proposed Supervised Topic Models (sLDA). sLDA considered documents that are already marked with a response variable (e.g., movie reviews with a numeric score between 1 and 5), and provides a means to automatically discover topics that help with the classification (i.e., predicting the response variable) of unseen documents.

Paul (Paul, 2009.) introduced Cross-Collection Topic Models (ccLDA) which discovered topics from multiple corpora, allowing the topics to exhibit slightly different behavior in each corpus. For example, a “food” topic might contain the words {food cheese fish chips} in a British corpus and the words {food cheese taco burrito} for a Mexican corpus.

Authors (Ramage, Hall, Nallapati, & Manning, 2009) introduced Labeled LDA (LLDA) LLDA takes as input a text corpus in which each document is labeled with one or more labels (such as Wikipedia) and discovers the term-label relations. LLDA discovers a set of topics for each label and allows documents to only display topics from one of its labels.

Authors (Mimno, Wallach, Naradowsky, Smith, & McCallum, 2009) proposed Polylingual Topic Model (PLTM). PLTM can handle corpora in several different languages, discovering aligned topics in each language. For example, if PLTM runs on English and German corpora, it might discover the aligned “family” topics {child parent sibling} and {kind eltern geschwister}.

Authors (Chang & Blei, 2009) introduced Relational Topic Models (RTM). RTM models documents as does LDA, as well as discovers links between each pair of documents. For example, if document 1 contained the “planets” topic, document 2 contained the “asteroids” topic, and document three contained the “Michael Jackson” topic, then RTM would assign a stronger relationship between documents 1 and 2 than between documents 1 and 3 or documents 2 and 3, because topics 1 and 2 are more closely related to each other.

2.3 Applications of Topic Models in Software Engineering

The various applications of topic models in field of software engineering are discussed below.

2.3.1 Concept Location

The task of concept location (or feature location) is to identify the parts (e.g., documents or methods) of the source code that implement a given feature of the software system. This is useful for developers wishing to debug or enhance a given feature. For example, if the so-called file printing feature contained a bug, then a concept location technique would attempt to automatically find those parts of the source code that implement file printing (i.e., parts of the source code that are executed when the system prints a file). Related to concept location is aspect-oriented programming (AOP), which aims at providing developers with the machinery to easily implement aspects of functionality whose implementation spans over multiple source code documents.

Authors (Linstead, Rigor, Bajracharya, Lopes, & Baldi, Mining Eclipse developer contributions via author-topic models, 2007) were the first to use LDA to locate concepts in source code in the form of LDA topics. The proposed approach can be applied to individual systems or large collections of systems to extract the concepts found within the identifiers and comments in the source code. The authors demonstrated how to group related source code documents based on comparing the documents' topics.

Authors (Linstead, Rigor, Bajracharya, Lopes, & Baldi, Mining Eclipse developer contributions via author-topic models, 2007)applied a variant of LDA, the Author-Topic model, to source code to extract the relationship between developers (authors) and source code topics. The proposed technique allows the automated summarization of "who has worked on what", and the authors provided a brief qualitative argument as to the effectiveness of this approach.

Authors (Maskeri, Sarkar, & Heafield, 2008) applied LDA to source code to extract the business concepts embedded in comments and identifier names. The authors applied a weighting scheme for each keyword in the system, based on where the keyword is found (e.g., class name, parameter name, method name). The authors found that their LDA-based approach is able to successfully extract business topics, implementation topics, and cross-cutting topics from source code.

Authors (Baldi, Lopes, Linsteda, & Bajracharya, 2008) proposed a theory that software concerns are equivalent to the latent topics found by statistical topic models. Further, they proposed that aspects are those latent topics that have a high scattering metric. The authors applied their approach to a large set of open-source projects to identify the global set of topics, as well as perform a more detailed analysis of a few specific projects. The authors found that latent topics with high scattering metrics are indeed those that are typically classified as aspects in the AOP community.

Authors (Savage, Dit, Gethers, & Poshyvanyk, 2010) introduced a topic visualization tool, called TopicXP, which supports interactive exploration of discovered topics located in source code.

2.3.2 Traceability Recovery

Traceability recovery aims to automatically uncover links between pairs of software artifacts, such as source code documents and requirements documents. This allows a project stakeholder to trace a requirement to its implementation, for example to ensure that it has been implemented correctly or not. Traceability recovery between pairs of source code documents is also important for developers wishing to learn which source code documents are somehow related to the current source code file being worked on.

Authors (Asuncion, Asuncion, & Taylor, 2010) introduced a tool called TRASE that uses LDA for prospectively, as opposed to retrospectively, recovering traceability links amongst diverse artifacts in software repositories. This means that developers can create and maintain traceability links as they work on the project. The authors demonstrated that LDA outperforms LSI in terms of precision and recall.

2.3.3 Source Code Metrics

Bug prediction (or defect prediction or fault prediction) tries to automatically predict which parts (e.g., documents or methods) of the source code are likely to contain bugs. This task is often accomplished by collecting metrics on the source code, training a statistical model to the metrics of documents that have known bugs, and using the trained model to predict whether new documents will contain bugs.

Authors (Linstead & Baldi, Mining the coherence of GNOME bug reports with statistical topic models, 2009) applied LDA to the bug reports in the GNOME project with the goal of measuring the coherence of a bug report, i.e., how easy to read and how focused a bug report is. This coherence metric is defined as the tangling of LDA topics within the report, i.e., how many topics are found in the report (fewer are better).

Authors (Liu, Poshyvanyk, Ferenc, Gyimothy, & Chrisochoides, 2009) applied LDA to source code methods in order to compute novel class cohesion metric called Maximum Weighted Entropy (MWE). MWE is computed based on the occupancy and weight of a topic in the methods of a class. The authors demonstrated that this metric captures novel variation in models that predict software faults.

Authors (Gethers & Poshyvanyk, 2010) introduced a new coupling metric, the Relational Topic-based Coupling (RTC) metric, based on a variant of LDA called Relational Topic Models (RTM). RTM extends LDA by explicitly modeling links between documents in the corpus. RTC uses these links to define the coupling between two documents in the corpus. The authors demonstrated that their proposed metric provides value because it is statistically different from existing metrics.

2.3.4 Software Evolution and Trend Analysis

Authors (Linstead, Lopes, & Baldi, 2008) applied LDA to several versions of the source code of a project in an effort to identify the trends in the topics over time. Trends in source code histories can be measured by changes in the probability of seeing a topic at specific version. When documents pertaining to a particular topic are first added to the system, for example, the topics will experience a spike in overall probability.

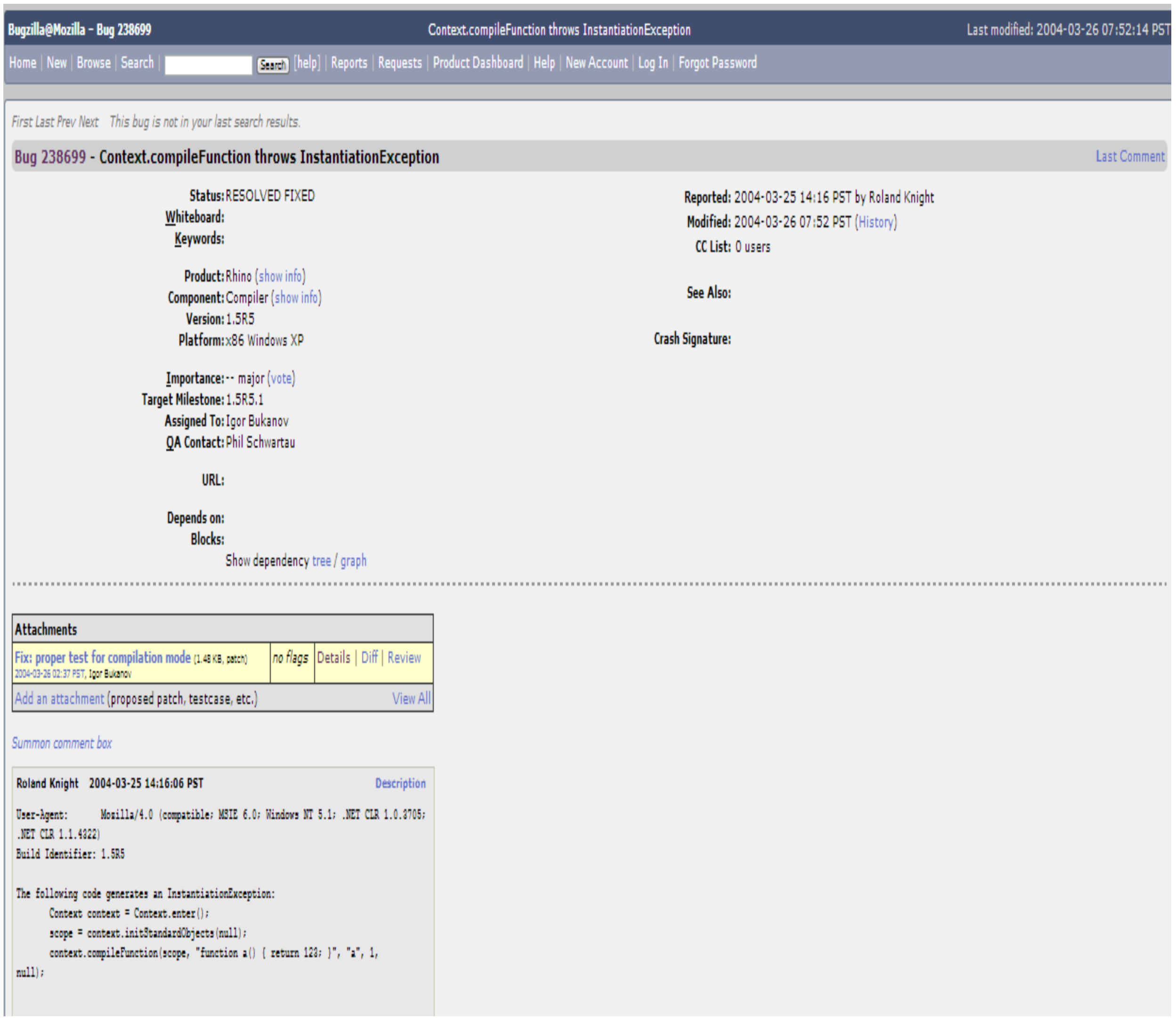
Authors (Thomas, Adams, Hassan, & Blostein, 2010) evaluated the effectiveness of topic evolution models for detecting trends in the software development process. The authors applied LDA to a series of versions of the source code and calculated the popularity of a topic over time. The authors manually verified that spikes or drops in a topic's popularity indeed coincided with developer activity mentioned in the release notes and other project documentation, providing evidence that topic evolution models provide a good summary of the software history.

2.4 Bug Localization

Bug localization is a process of mapping a bug back to the code that might have caused it. Bug and the various stages in its life cycle have been discussed below. Bug tracking system and use of IR models in bug localization has also been discussed below.

2.4.1 Defining bug

"A computer bug is an error, flaw, mistake, failure, or fault in a computer program that stops it from working correctly or produces an incorrect result. Bugs arise from mistakes and errors, made by people, in either a program's source code or its design." The first Software bug was seen by Grace Murray Hopper in year 1947 on Harvard University Mark II Aiken Relay Calculator (a primitive computer).



Bugzilla@Mozilla – Bug 238699 | Context.compileFunction throws InstantiationException | Last modified: 2004-03-26 07:52:14 PST

Home | New | Browse | Search | [Search] [help] | Reports | Requests | Product Dashboard | Help | New Account | Log In | Forgot Password

First Last Prev Next This bug is not in your last search results.

Bug 238699 - Context.compileFunction throws InstantiationException | Last Comment

Status:	RESOLVED FIXED	Reported:	2004-03-25 14:16 PST by Roland Knight
Whiteboard:		Modified:	2004-03-26 07:52 PST (History)
Keywords:		CC List:	0 users
Product:	Rhino (show info)		
Component:	Compiler (show info)	See Also:	
Version:	1.5R5		
Platform:	x86 Windows XP	Crash Signature:	
Importance:	-- major (vote)		
Target Milestone:	1.5R5.1		
Assigned To:	Igor Bukanov		
QA Contact:	Phil Schwartau		
URL:			
Depends on:			
Blocks:			

Show dependency tree / graph

Attachments		
Fix: proper test for compilation mode (1.48 KB, patch) 2004-03-26 02:37 PST, Igor Bukanov	no flags	Details | Diff | Review
Add an attachment (proposed patch, testcase, etc.)		View All

Summon comment box

Roland Knight 2004-03-25 14:16:06 PST | Description

```
User-Agent:       Mozilla/4.0 (compatible; MSIE 6.0; Windows NT 5.1; .NET CLR 1.0.3705;
.NET CLR 1.1.4322)
Build Identifier: 1.5R5

The following code generates an InstantiationException:
      Context context = Context.enter();
      scope = context.initStandardObjects(null);
      context.compileFunction(scope, "function a() { return 123; }", "a", 1,
null);
```

Figure 2: Bug Report in Bugzilla

2.4.2 Stages in Life cycle of Bug

In software development process, the bug has a life cycle (Rakesh). The bug should go through the life cycle to be closed. A specific life cycle ensures that the process is standardized. The bug attains different states in the life cycle. The different states of a bug can be summarized as follows:

1. New
2. Open
3. Assign
4. Test
5. Verified

6. Deferred
7. Reopened
8. Duplicate
9. Rejected
10. Closed

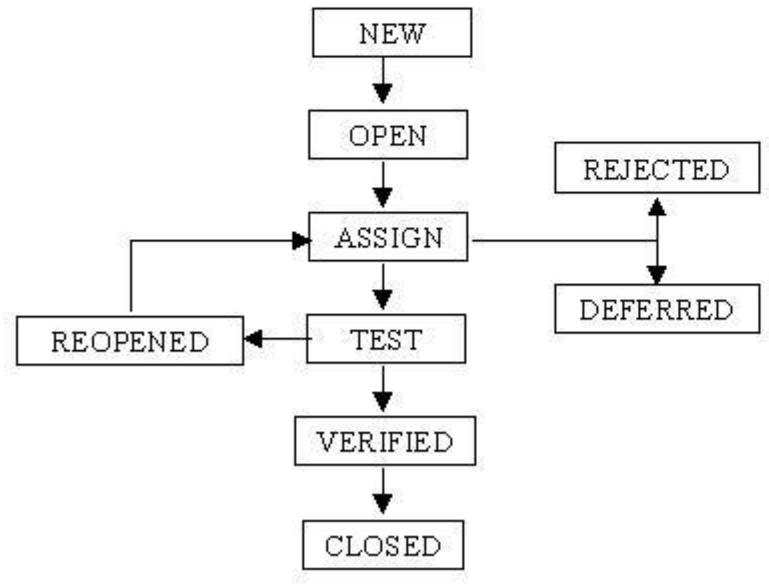


Figure 3: Bug Life Cycle

Description of Various Stages:

1. New: When the bug is posted for the first time, its state will be "NEW". This means that the bug is not yet approved.
2. Open: After a tester has posted a bug, the bug is approved as genuine by the lead of the tester the state is changed as "OPEN".
3. Assign: Once the lead changes the state as "OPEN", the bug is assigned to corresponding developer or developer team. The state of the bug now is changed to "ASSIGN".
4. Test: Once the bug is fixed by developer, the bug is assigned to the testing team for next round of testing. Before releasing the software with bug fixed, the state of bug is changed to "TEST". It specifies that the bug has been fixed and is released to testing team.

5. Deferred: The bug, changed to deferred state means the bug is expected to be fixed in next releases. The reasons for changing the bug to this state have many factors. Some of them are priority of the bug may be low, lack of time for the release or the bug may not have major effect on the software.
6. Rejected: If the developer feels that the bug is not genuine, the bug is rejected and the state of the bug is changed to "REJECTED".
7. Duplicate: If the bug is repeated twice or the two bugs mention the same concept of the bug, then status of one bug is changed to "DUPLICATE".
8. Verified: Once the bug is fixed and the status is changed to "TEST", the tester tests the bug. If the bug is not present in the software, the bug is approved as fixed and the status is changed to "VERIFIED".
9. Reopened: If the bug still exists even after the bug is fixed by the developer, the status is changed to "REOPENED" by tester. The bug traverses the life cycle once again.
10. Closed or Fixed: Once the bug is fixed, it is tested by the tester. If the bug no longer exists in the software, the status is changed to "CLOSED". This state means that the bug is fixed, tested and approved.

2.4.3 Bug tracking System

A bug tracking system or defect tracking system is a software application that is designed to keep track of reported software bugs in software development efforts. It may be regarded as a type of issue tracking system.

Many bug tracking systems, such as those used by most open source software projects, allow users to enter bug reports directly. Other systems are used only internally in a company or organization doing software development. Typically bug tracking systems are integrated with other software project management applications.

A major component of a bug tracking system is a database that records facts about known bugs. Facts may include the time a bug was reported, its severity, the erroneous program behavior, and details on how to reproduce the bug; as well as the identity of the person who reported it and any programmers who may be working on fixing it. Typical bug tracking systems support the concept of the life cycle for a bug which is tracked through status assigned to the bug. Bugzilla is the very first Bug Tracking System developed for tracking bugs in the year 1998. Bugzilla is written in Perl language.

In 2000, Mantis Bug Tracker was introduced written in PHP. Mantis introduced a much nicer user interface than Bugzilla, and offered more customization, including customization of the bug workflow and state transitions.

JIRA, a commercial product launched in 2003 and built in Java, represented another step forward for issue tracking systems, adding additional customization options, and a powerful plug-in architecture. The JIRA platform tends to work best for large enterprise software projects.

In 2006, two similar projects were introduced: Trac and Redmine. Both are open-source project management and issue tracking systems. Both offer web-based ticketing system similar to Mantis, along with support for milestones, Wiki-style documentation, and source integration. Trac is written in Python, whereas Redmine is developed on the recently popular Ruby on Rails framework.

2.4.4 Information Retrieval Model for Bug localization

Information Retrieval (IR) can be defined as: “Retrieving relevant documents (or documents that satisfy user information need) from large and unstructured collection of documents" (Anvik, Hiew, & Murphy, 2006).

Information Retrieval is an art and science of searching (or retrieving) relevant documents from the large collection of documents, for example:

- Searching for articles on image processing.
- Retrieving web pages relevant to endangered species.
- Retrieving advertisement on latest laptops brands present in the market.

All these are real world examples that are encountered in daily life. Web search engines such as Google, Yahoo, Bing etc. are the biggest applications of IR system. These search engines indexes millions of documents (unstructured or semi-structured nature) which are used for IR model building. When a user input's a query this IR model is used to provide user ranked list of document which are ordered according to their relevance to the given query (Sangeeta, 2011).

IR models are gaining popularity in bug localization domain mainly because of two reasons: 1) scalability, and 2) language independence (Rao & Kak, 2011). These features of IR model allow automated bug localization tools to remain applicable as software grows in size and complexity.

For bug localization problem IR models have been built using software source code information. In addition to source code other information present with the software system such as software documentation, software specification or previous bug locations has also been used for IR model formation. Document collection represents at which level of granularity bug localization system need to locate the bug, it can be at statement, method, class, or file level. Document collection is formed from source code by breaking it into desired level of granularity. Any new bug report is considered as a query for the system for which relevant documents need to be retrieved. New bug reports are converted to query using query formation module. All this information (IR model, document collection, query) is used by query engine module to produce ranked list of documents from document collection. Documents are ranked in order of their relevance with respect to current query. These ranked documents can be used by software developers to predict bug location during bug fixing.

Authors (Hayes, Nichols, Kraft, & Anderson) proposed a technique for bug localization in which LSI model has been used. And to improve the efficiency, historical patch data has also been used. For locating a given bug combined result of both previous history and LSI based approach has been used.

Shao (Shao, 2011) proposed an improvement in LSI based bug localization by combining the structural information as well. In this work, LSI has been combined with call graphs for the task of locating bugs. This LSI-Call Graph based approach has shown better results as compared to LSI based approach.

2.5 Bug Localization Using Topic Models

In recent times, researchers have developed automated static bug localization location techniques (Poshyvanyk & Marcus, 2007) (Lukins, Kraft, & Etzkorn, 2008) (Lal & Sureka, 2012)using topic models of Information Retrieval (IR) such as Latent Semantic Indexing (LSI) (Deerwester, Dumais, Landauer, Furnas, & Harshman, 1990), Latent Dirichlet allocation (LDA) (Blei, Ng, & Jordan, 2003) and N-Gram (Wei X. , 2007). These techniques show efficacy but leave room for some improvement.

Authors (Lukins, Kraft, & Etzkorn, 2008) presented an LDA based approach to bug localization at method level granularity. In five case studies on Mozilla, Eclipse, and Rhino they demonstrate show that the approach outperforms LSI as well as the accuracy and scalability of the approach. Further, the authors show that the approach is not sensitive to the size of the subject software system and that there is no relationship between the accuracy of the approach and the stability of the subject system. Finally, the authors demonstrate that coding links can be used to navigate from the first relevant method in the ranking to other methods modified to correct the bug.

Authors (Lal & Sureka, 2012) used N-gram model for the task of bug localization at file level granularity. The authors used experimental datasets from two open source project (JBoss and Apache). Experimental results reveal that the median value for the SCORE metric for JBOSS and Apache dataset is 99.03% and 93.70% respectively.

2.6 Motivation

After going through various research proposals in the area of localizing bug, it has been observed that there is a scope of improvement in the technique for locating the bugs. Topic models like LDA, N-gram, etc have already been used for this task but still there exists some topic models which have shown promising results in various fields and are not used for the task of bug localization yet. One such model is Pachinko Allocation Model (PAM), which captures arbitrary, nested, and possibly sparse correlations between topics using a directed acyclic graph (DAG) and has not been used for locating bugs yet.

Also, the topics discovered by LDA capture correlations among words, but LDA does not explicitly model correlations among topics. This limitation arises because the topic proportions in each document are sampled from a single Dirichlet distribution. As a result, LDA has difficulty modeling data in which some topics co-occur more frequently than others. Motivated by the desire to present more accurate approach for locating bugs by discovering large numbers of fine-grained topics and finding correlations between them, PAM model has been used in this research work for the task of bug localization.

CHAPTER 3
BUG LOCALIZATION USING TOPIC MODELS

3 BUG LOCALIZATION USING TOPIC MODELS

In this chapter, bug localization has been done at file level granularity using LDA and PAM models. LDA model has already been used for this task while PAM has not been used yet for the task of bug localization. LDA and PAM models are discussed in this chapter. In the last section, the methodology used in this work for performing bug localization using PAM has been discussed.

3.1 Latent Dirichlet Allocation (LDA) Model

Latent Dirichlet Allocation is a powerful learning algorithm for automatically and jointly clustering words into "topics" and documents into mixtures of topics. It has been successfully applied to model change in scientific fields over time.

Latent Dirichlet Allocation (LDA) is a popular probabilistic topic model (Blei, Ng, & Jordan, 2003) that has largely replaced pLSI. One of the reasons it is so popular is because it models each document as a multi-membership mixture of K corpus-wide topics, and each topic as a multi membership mixture of the terms in the corpus vocabulary. This means that there are a set of topics that describe the entire corpus, each document can contain more than one of these topics, and each term in the entire repository can be contained in more than one of these topic. Hence, LDA is able to discover a set of ideas or themes that well describe the entire corpus (Blei & Lafferty, Topic models, 2009).

To generate a document, LDA first samples per-document multinomial distribution over topics from a Dirichlet distribution. Then it repeatedly samples a topic from this multinomial and samples a word from the topic. Before an LDA analysis can be performed on the document collection, the following parameters must be set.

- The number of topics
- The number of iterations for the Gibbs sampling process
- α, a hyper parameter of LDA, determines the amount of smoothing applied to the topic distributions per document (Griffiths & Steyvers, 2004) .
- β, a hyper parameter of LDA, determines the amount of smoothing applied to the word distributions per topic (Griffiths & Steyvers, 2004)

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

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

LDA is based on a fully generative model that describes how documents are created. Intuitively, this generative model makes the assumption that the corpus contains a set of K corpus-wide topics, and that each document is comprised of various combinations of these topics. Each term in each document comes from one of the topics in the document (Wei & Croft, 2006) (Thomas S. W., 2012). This generative model is formulated as follows:

1. Choose a topic $\theta_{d\sim}\text{Dirichlet}(\alpha)$ for document d.

2. For each of the N terms w_i:

(a) Choose a topic $z_{k\sim}\text{Multinomial}\ \theta_d$.

(b) Choose a term w_i from $P(w_i|z_k,\beta)$.

Here, $P(w_i|z_k,\beta)$ is a multinomial probability function, α is a smoothing parameter for document-topic distributions, and β is a smoothing parameter for topic-term distributions.

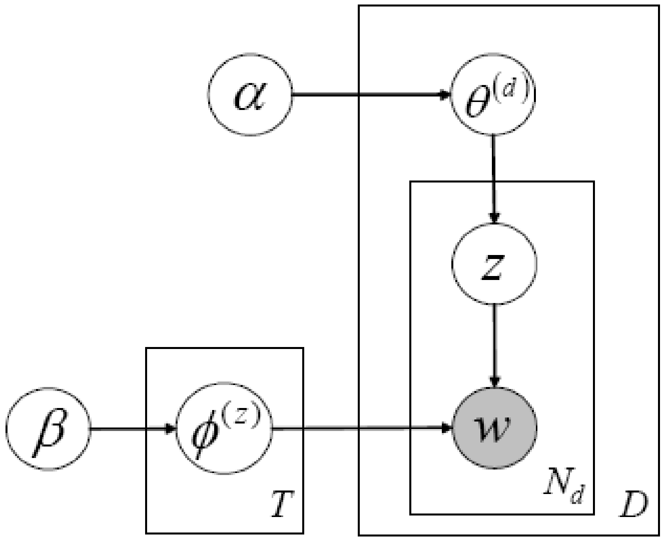


Figure 4: Graphical notation for LDA

LDA model with repeated sampling steps can be conveniently illustrated using plate notation as shown in figure 4. In this graphical notation, shaded and un shaded variables indicate observed and latent (i.e., unobserved) variables respectively. The variables φ and θ, as well as z (the assignment of word tokens to topics) are the three sets of latent variables that one would like to infer. The hyper parameters α and β are considered as constants in the model. Arrows indicate conditional dependencies between variables while plates (the boxes in figure 4) refer to repetitions of sampling steps with the variable in the lower right corner referring to the number of samples. For example, the inner plate over z and w illustrates the repeated sampling of topics and words until N_d words have been generated for document d. The plate surrounding $\theta(d)$ illustrates the sampling of a distribution over topics for each document d for a total of D documents. The plate surrounding $\varphi(z)$ illustrates the repeated sampling of word distributions for each topic z until T topics have been generated.

Two levels of this generative model allow three important properties of LDA to be realized: documents can be associated with multiple topics, the number of parameters to be estimated does not grow with the size of the corpus, and, since the topics are global and not estimated per document, unseen documents can easily be accounted (Thomas S. W., 2012).

One assumption that LDA makes is the “bag of words” assumption that the order of the words in the document does not matter. For more sophisticated goals such as language generation it is patently not appropriate. There have been a number of extensions to LDA that model words non exchangeable. Another assumption is that the order of documents does not matter. A third assumption about LDA is that the number of topics is assumed known and fixed.

3.2 Pachinko Allocation Model

Pachinko allocation model (PAM), uses a directed acyclic graph (DAG) structure to represent and learn arbitrary nested, and possibly sparse topic correlations. In PAM, the concept of topics is extended to be distributions not only over words, but also over other topics. The model structure consists of an arbitrary DAG, in which each leaf node is associated with a word in the vocabulary, and each non-leaf “interior” node corresponds to a topic, having a distribution over its children. An interior node whose children are all leaves would correspond to a traditional LDA topic. But some interior nodes may also have children that are other topics, thus representing a mixture over topics. With many such nodes, PAM therefore captures not only correlations among words (as in LDA), but also correlations among topics themselves. Figure 5 and figure 6 shows the structure of LDA and PAM models respectively.

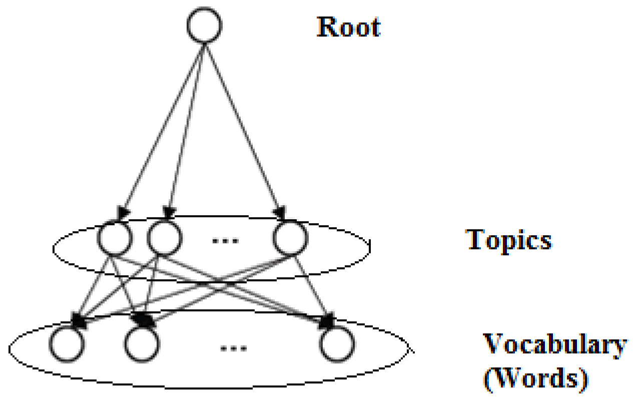


Figure 5: LDA Structure

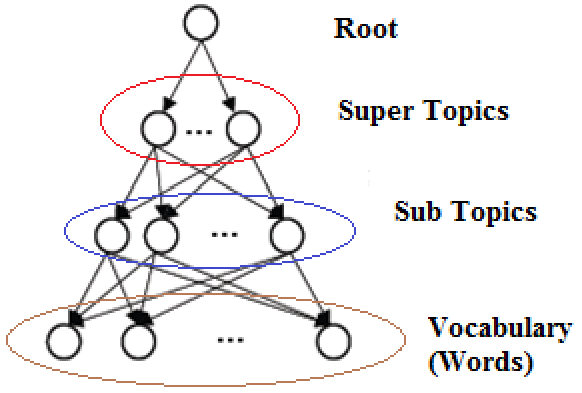


Figure 6: PAM Structure

For example, consider a document collection that discusses four topics: cooking, health, insurance and drugs. The cooking topic co-occurs often with health, while health, insurance and drugs are often discussed together. A DAG can describe this kind of correlation. Four nodes for the four topics form one level that is directly connected to the words. There are two additional nodes at a higher level, where one is the parent of cooking and health, and the other is the parent of health, insurance and drugs (Li & McCallum, 2006).

In PAM each interior node's distribution over its children could be parameterized arbitrarily. PAM model consists of a DAG, with each interior node containing a Dirichlet distribution over its children. To generate a document from this model, first sampling a multinomial from each Dirichlet is done. Then, to generate each word of the document, one has to begin at the root of the DAG, sampling one of its children according to its multinomial, and so on sampling children down the DAG until leaf is reached, which yields a word. The model is named for pachinko machines, a game popular in Japan, in which metal balls bounce down around a complex collection of pins until they land in various bins at the bottom.

It is easy to see that LDA can be viewed as a special case of PAM: the DAG corresponding to LDA is a three-level hierarchy consisting of one root at the top, a set of topics in the middle and a word vocabulary at the bottom. The root is fully connected to all the topics, and each topic is fully connected to all the words.

PAM connects words in V and topics in T with an arbitrary DAG, where topic nodes occupy the interior levels and the leaves are words. It is a four-level hierarchy consisting of one root topic r, s topics at the second level $T = \{t_1, t_{2,\ldots\ldots,} t_s\}$, s'topics at the third level $T' = \{t'_1, t'_{2,\ldots\ldots,} t'_{s'}\}$ and words at the bottom. The topics at the second level are called super-topics and the ones at the third level as sub-topics. The root is connected to all super-topics, super-topics are fully connected to sub-topics and sub-topics are fully connected to words. $g_i(\alpha_i)$ is Dirichlet distribution associated with topic t_i.

The generative process for a document d in PAM is as follows:

1. Sample $\theta_r^{(d)}$ from the root $g_r(\alpha_r)$, where $\theta_r^{(d)}$ is a multinomial distribution over super topics.
2. For each super-topic t_i, sample $\theta_{t_i}^{(d)}$, where $\theta_{t_i}^{(d)}$ is a multinomial distribution over sub-topics.

3. For each word w in the document,

a) Sample a super-topic z_w from $\theta_r^{(d)}$.

b) Sample a sub-topic z'_w from $\theta_{zw}^{(d)}$.

c) Sample word w from $\varphi_{z'_w}$

Following this process, a joint probability for generating a document , a super-topic assignment $z^{(d)}$, a sub-topic assignment $z'^{(d)}$ and a multinomial distribution $\theta^{(d)}$ is calculated as:

$$P\left(d, z^{(d)}, z'^{(d)}, \theta^{(d)} \middle| \alpha, \varphi\right) =$$
$$P(\theta_r^{(d)}|\alpha_r) \prod_{i=1}^{s} P\left(\theta_{t_i}^{(d)}|\alpha_i\right) \times \prod_w (P(z_w|\theta_r^{(d)}) P(z'_w|\theta_{zw}^{(d)})\, P(w|\varphi_{z'_w})) \qquad (1)$$

Integrating out $\theta^{(d)}$ and summing over $z^{(d)}$ and $z'^{(d)}$, the marginal probability of the document is calculated as:

$$P(d|\alpha, \varphi) =$$
$$\int P(\theta_r^{(d)}|\alpha_r) \prod_{i=1}^{s} P(\theta_{t_i}^{(d)}|\alpha_i) \times \prod_w (P\left(z_w \middle| \theta_r^{(d)}\right) P\left(z'_w \middle| \theta_{zw}^{(d)}\right) P\left(w \middle| \varphi_{z'_w}\right)) d\theta^{(d)} \qquad (2)$$

The probability of generating the whole corpus is the product of the probability for every document, integrating out multinomial distributions for sub-topics φ as:

$$P(D|\alpha, \beta) = \int \prod_{j=1}^{s'} P(\varphi_{t'_j}|\beta) \prod_d P(d|\alpha, \varphi) d\varphi \qquad (3)$$

3.3 Bug Localization Using Pachinko Allocation Model

To perform PAM-based bug localization on a given version of a software system, first a PAM model of the source code is built. Then, the created model is queried as often as necessary to localize bugs existing in that version. The implementation work has been done by extending MALLET (McCallum, 2002) library of JAVA.

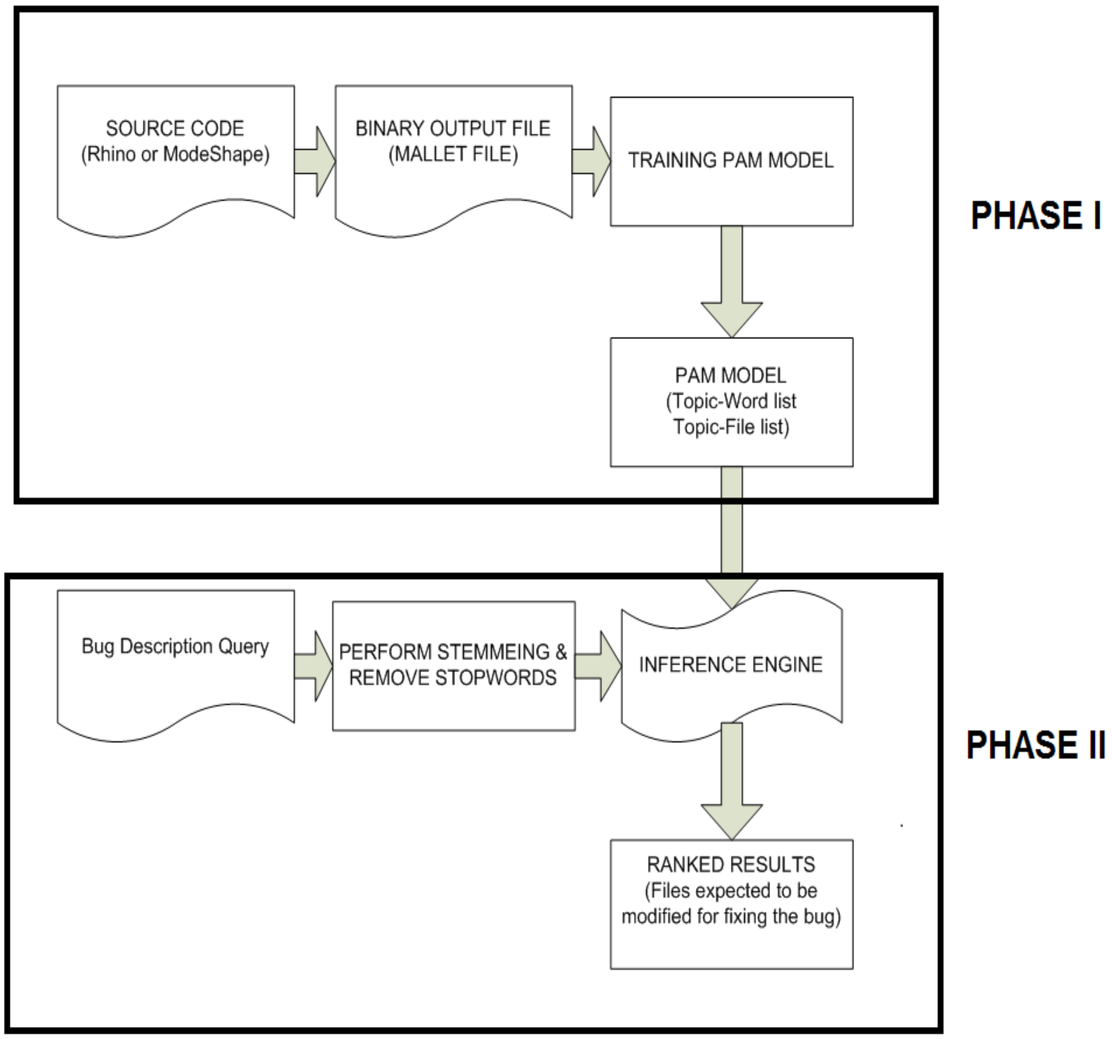

SOURCE CODE (Rhino or ModeShape)
BINARY OUTPUT FILE (MALLET FILE)
TRAINING PAM MODEL
PHASE I
PAM MODEL (Topic-Word list Topic-File list)
Bug Description Query
PERFORM STEMMEING & REMOVE STOPWORDS
INFERENCE ENGINE
PHASE II
RANKED RESULTS (Files expected to be modified for fixing the bug)

Figure 7: Bug Localization Using PAM Model

Figure 7 shows the two main phases involved in performing bug localization using PAM model.

3.3.1 Preprocessing of Source Code

Before PAM model can be applied to source code, several preprocessing steps are generally taken in an effort to reduce noise and improve the resulting topics. These steps are same for other topic models also.

a. Characters related to the syntax of the programming language (e.g., “&&”, “->”) are removed; programming language keywords (e.g., “if”, “while”) are removed.

b. Identifier names are split into multiple parts based on common naming conventions (e.g., “oneTwo”, “one_two”).
c. Common English-language stopwords (e.g., “the”, “it”, “on”) are removed.
d. Word stemming is applied to find the root of each word (e.g., “changing” becomes “chang”).

3.3.2 Construction of PAM Model

Two steps are necessary to construct a PAM model of a software system: (1) building the document collection from the source code, and (2) Training the PAM Model.

Step1: Building the Binary output file.

In this step, source code is passed through the program written for implementation task. Binary output file gets created after this first step. Semantic information is extracted and stemming is performed to eliminate stop words. Porter stemmer removed the word suffix and eliminated the variations and repetition of words.

Step 2: Training the PAM Model

In this step, training of the topic model from the data file generated in the previous step is done. As an output, topic-words list is generated. But before a PAM model can be trained, the following parameters must be set.

- The number of sub topics
- The number of sub topics
- The number of iterations for the Gibbs sampling process
- α, a hyper parameter of PAM, determines the amount of smoothing applied to the topic distributions per document (Li & McCallum, 2006)
- β, a hyper parameter of PAM, determines the amount of smoothing applied to the word Distributions per topic (Li & McCallum, 2006).

By default, the number of super topics are generally half of the subtopics but could be larger than it. At this point, a static PAM model of the source code has been constructed. This model can then be queried for each bug discovered.

3.3.3 Query the PAM Model using Inference Engine

PAM model generated in previous phase can be queried now. Terms in the query should be preprocessed in the same manner as the source code, e.g., stop words removed and stemming performed. Each query results in a list of source code elements ranked by similarity to the query (most similar elements ranked highest). In this work, Inference engine for querying the PAM model has been proposed by extending MALLET library in java.

In this work, source code queries have been formulated manually by utilizing information about bugs extracted from the bug title and description entered into the software's bug repository by the person initially reporting the bug. Separate files were prepared for each bug containing its summary and description. These files act as the query for the Inference engine. The query is one by one into the engine to get the topic corresponding to the query content. After the first phase list of topics corresponding to each file has already generated. This data was then used to know the files that were having the maximum percentage of the topic. The files are arranged in the decreasing order of the topic percentage, file with maximum percentage at the top. In this way the list of expected files to be modified is created. Details regarding the formation of queries for each case study are discussed in the description for each study.

CHAPTER 4
CASE STUDY

4 CASE STUDY

In this chapter, details about the data sets have been given. This chapter also discusses the experimental set up and simulation. The main phases of the proposed work are also discussed in this chapter.

4.1 Experimental Data Sets

To assess the viability of a PAM-based approach to bug localization and to compare it with LDA based bug localization, two case studies have been performed on two different software systems. Both the case studies use the approach outlined in Section 3.3 to perform bug localization. To determine the accuracy of the predictions for each bug, the PAM and LDA query results were compared to relevant files for the bug i.e., the actual files fixed by developers to correct the bug. These relevant source code files were determined by examining the software patch for each bug posted in the software's bug tracking system.

For the task of evaluating the performance of the proposed work, source codes have been downloaded for two popular open source projects: Rhino (Rhino) and Modeshape. The dataset is publicly available as a result of which the experiments performed in this work can be replicated in future for improvement. The data about the bugs has been downloaded from two Bug Tracking systems i.e. Bugzilla and JIRA for Rhino and Modeshape respectively.

4.1.1 Rhino

Rhino is an open-source implementation of JavaScript written entirely in Java. It is typically embedded into Java applications to provide scripting to end users. It is embedded in J2SE 6 as the default Java scripting engine. In this work, source code of Rhino (Rhino) version 1.7 release (1.7R) with 219 files has been used. Bugs and its details have been downloaded from Bugzilla. Bugs that fulfill the following criteria have been used for analysis of result:

a) Bugs existing in the Rhino version 1.7
b) Bugs with the status "Closed" or "Resolved"
c) Bugs requiring modification at file level

Table 2 lists the bug id and bug summary from Rhino's bug repository (Bugzilla) for each bug examined.

Table 2: Bug id and Summary for Rhino

Bug id	**Bug Summary**
220367	NPE when accessing RegExp.$1 after matching /(a)|(b)/ against "b"
510265	Make the source property of RegExp instances conform to the spec
684131	AstNode missing operator name "^=" (ASSIGN_BITXOR) - patch included
537483	JSON.parse doesn't correctly add properties with numeric identifiers
513549	Rhino's new JSON.parse breaks on trailing whitespace
507104	Make RegExp.prototype.constructor non-enumerable
505524	Implement Date.toJSON
442922	New E4X Dom based XML implementation is not serializable
400159	Make org.mozilla.javascript.Synchronizer act on native Java objects when available
255595	Factory class for Context creation
281067	ThreadLocal in Context prevents class unloading
258959	ScriptableInputStream doesn't use Context's applicationClassLoader to resolve classes
245882	JavaImporter constructor
236193	Only active Context for compilation
236117	Context sealing API for Rhino
76683	RegExp regression (NullPointerException)
201987	delete "".x throws ClassCastException
198086	optimizer enhancement: generate only single class per script and all its functions
214997	build.xml changes: clean and help targets

4.1.2 ModeShape

ModeShape is a distributed, hierarchical, transactional, and consistent data store with support for queries, full-text search, events, versioning, references, and flexible and dynamic schemas. It is very fast, highly available, extremely scalable, and it is 100% open source and written in Java. Bugs and its details have been downloaded from JIRA. For this work, ModeShape version 3.1.1 with source code of 1660 files has been used Bugs that fulfill the following criteria have been used for analysis of result:

a) Bugs existing in the ModeShape version 3.1.1

b) Bugs with the status “Closed” or “Resolved”

c) Bugs requiring modification at file level

Table 1 lists the bug id and bug summary from ModeShape’s bug repository(JIRA) for each bug examined.

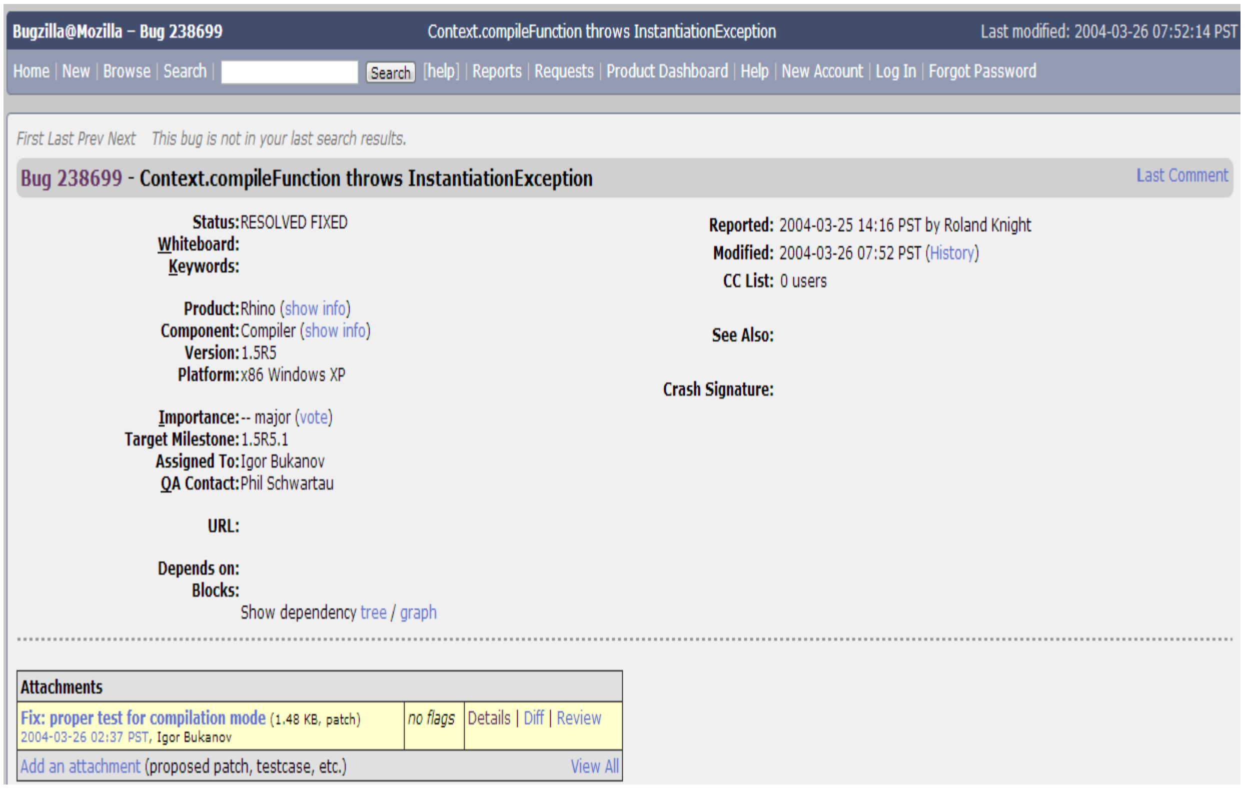
Table 3: Bug id and Summary for ModeShape

Bug id	**Bug Summary**
MODE-1878	Same name siblings are incorrectly prevented in some cases
MODE-1837	Sometimes query returns duplicated records after commiting a transaction that contains VersionManager.checkin() call when using a real JTA transaction manager
MODE-1769	org.infinispan.marshall.NotSerializableException: org.infinispan.schematic.internal.SchematicEntryLiteral when using async cache store
MODE-1751	Updating reference with already assigned node brings referential integrity exceptions
MODE-1748	Importing XML throws VersionException "node is checked in, preventing this action"
MODE-1414	Sequencing VDB project causes NullPointerException
MODE-1269	Methods to re-index content are not public
MODE-1207	WSDL sequencer does not sequence document correctly.
MODE-1131	Exception Querying For Workspace Areas Using IRestClient
MODE-1036	Modeshape unit test JpaConnectorNoCreateWorkspaceTest freezes when Oracle is used.
MODE 1016	ConstraintViolationException is thrown when importing sample drools rules from XML file.
MODE1013	XML sequencer doesn't work correctly
MODE 1004	Teiid VDB sequencer incorrectly set the value of the vdb:builtIn property
MODE 972	CND sequencer doesn't work correctly
MODE 950	Text sequencer does not sequence CSV file correctly
MODE 927	Unable to delete file through REST interface
MODE 902	NotSerializableException in the JpaRepository when using HSQL as the repository
MODE 815	DDL Sequencer doesn't populate primary key correctly when constraint defined on a column
MODE 802	Some valid non-identity joins will produce an error upon execution
MODE 797	Session.getWorkspace().getAvailableWorkspaceNames() does not match those available in the underlying source
MODE 793	Version Storage Does Not Preserve Cardinality of Properties
MODE 792	Checking Out an Already Checked-Out Node Resets Changes on That Node
MODE 790	XPath Query with Compound Predicate Not Translated Correctly

4.2 Bug Tracking Systems: Bugzilla and JIRA

A bug tracking system or defect tracking system is a software application that is designed to help keep track of reported software bugs in software development efforts. It may be regarded as a type of issue tracking system. Many bug tracking systems, such as those used by most open source software projects, allow users to enter bug reports directly. Other systems are used only internally in a company or organization doing software development. Typically bug tracking systems are integrated with other software project management applications. In this research work, Bugzilla and JIRA have been used for collecting the bug details of the software.

Bugzilla as shown in figure 8 is a "Defect Tracking System" or "Bug-Tracking System". Defect Tracking Systems allow individual or groups of developers to keep track of outstanding bugs in their product effectively. Most commercial defect-tracking software vendors charge enormous licensing fees.


Bugzilla@Mozilla – Bug 238699 Context.compileFunction throws InstantiationException Last modified: 2004-03-26 07:52:14 PST

Home | New | Browse | Search | Search [help] | Reports | Requests | Product Dashboard | Help | New Account | Log In | Forgot Password

First Last Prev Next This bug is not in your last search results.

Bug 238699 - Context.compileFunction throws InstantiationException Last Comment

Status: RESOLVED FIXED
Whiteboard:
Keywords:

Product: Rhino (show info)
Component: Compiler (show info)
Version: 1.5R5
Platform: x86 Windows XP

Importance: -- major (vote)
Target Milestone: 1.5R5.1
Assigned To: Igor Bukanov
QA Contact: Phil Schwartau

URL:

Depends on:
Blocks:
Show dependency tree / graph

Reported: 2004-03-25 14:16 PST by Roland Knight
Modified: 2004-03-26 07:52 PST (History)
CC List: 0 users

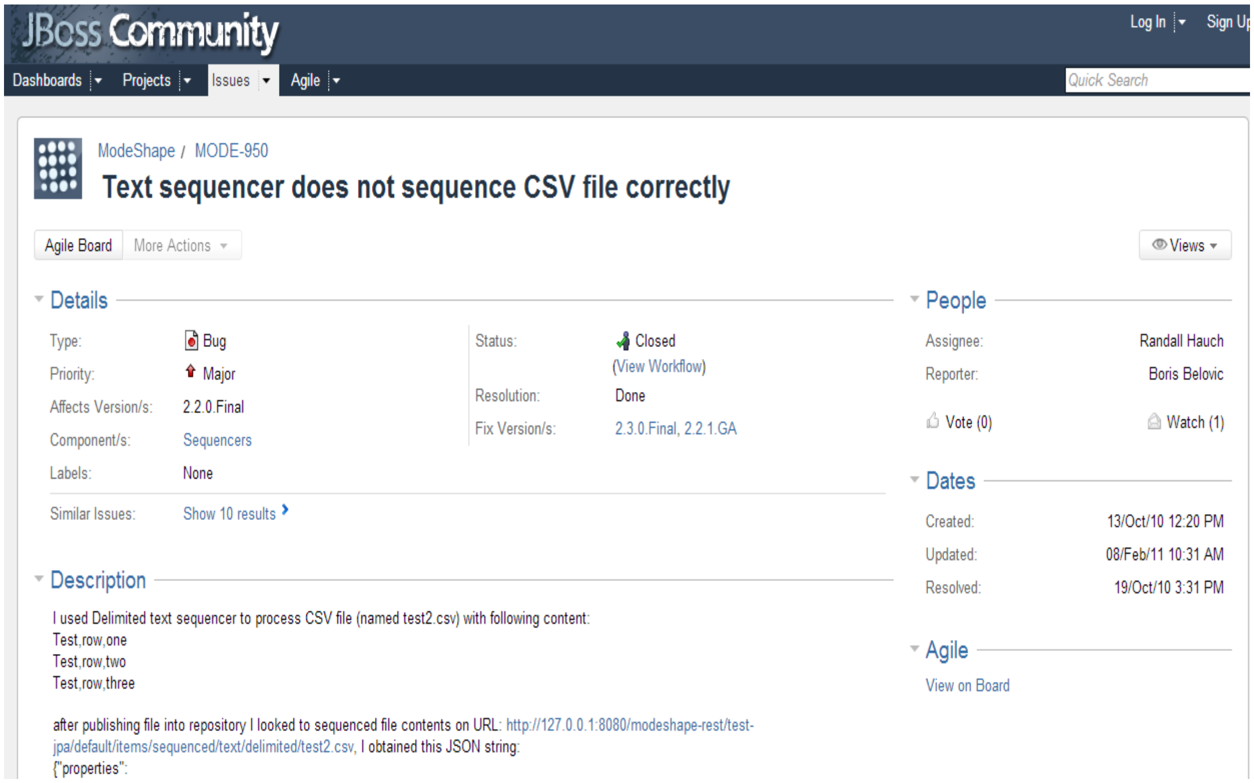
See Also:

Crash Signature:

Attachments		
Fix: proper test for compilation mode (1.48 KB, patch) 2004-03-26 02:37 PST, Igor Bukanov	no flags	Details | Diff | Review
Add an attachment (proposed patch, testcase, etc.)		View All

Figure 8: Bugzilla: Bug Tracking System for Rhino

Despite being "free", Bugzilla has many features its expensive counterparts lack. Consequently, Bugzilla has quickly become a favorite of thousands of organizations across the globe. In this work, Bugzilla has been used for collecting the details about bugs in Rhino.



ModeShape / MODE-950

Text sequencer does not sequence CSV file correctly

Details

Type:	Bug	Status:	Closed (View Workflow)
Priority:	Major	Resolution:	Done
Affects Version/s:	2.2.0.Final	Fix Version/s:	2.3.0.Final, 2.2.1.GA
Component/s:	Sequencers		
Labels:	None		
Similar Issues:	Show 10 results		

People

Assignee:	Randall Hauch
Reporter:	Boris Belovic
Vote (0)	Watch (1)

Dates

Created:	13/Oct/10 12:20 PM
Updated:	08/Feb/11 10:31 AM
Resolved:	19/Oct/10 3:31 PM

Agile

View on Board

Description

I used Delimited text sequencer to process CSV file (named test2.csv) with following content:
Test,row,one
Test,row,two
Test,row,three

after publishing file into repository I looked to sequenced file contents on URL: http://127.0.0.1:8080/modeshape-rest/test-jpa/default/items/sequenced/text/delimited/test2.csv, I obtained this JSON string:
{"properties":

Figure 9: JIRA: Bug Tracking System for ModeShape

JIRA as shown in figure 9 is a bug tracking system, developed by Atlassian, used for bug tracking, issue tracking and project management. The product name, JIRA, is not an acronym but rather a truncation of "Gojira", the Japanese name for Godzilla. In this work, JIRA has been used for collecting the details about bugs in ModeShape.

4.3 Performance and Evaluation Metrics

4.3.1 Mean Average Precision

Mean average precision (computed for a set of queries i.e., for the set of bug reports in the evaluation dataset) is equal to the mean of the Average Precision (AP) scores for each query in the experimental dataset. AP consists of computing the precision of the system at the rank of every relevant document retrieved. MAP is a well know metric to measure retrieval performance for IR systems.

Equation 1 and equation 2 gives the formula for calculating AP and MAP where P_i denotes the precision at i^{th} relevant file retrieved and M denotes total number of relevant files for a bug.

$$AP = \frac{(P_1 + P_2 + \cdots P_M)}{M} \tag{1}$$

$$\text{MAP} = \frac{(AP_1 + AP_2 + \cdots AP_M)}{N} \tag{2}$$

4.3.2 Rank of First Relevant File

Rank of first relevant file means rank of the first relevant file retrieved for a particular bug. In this work, this metric has been used to calculate number of bugs finding their relevant file at given rank. For example, in a given bug report, if the first relevant file is found at 3rd position, then rank will be 3. In this work, three ranges of rank has been used. First, rank less 5(Rank<5), second is rank between 6 to 10(6<=Rank<=10) and third is rank between 11 to 20(11<=Rank<=20). The higher the metric value, better the bug localization performance.

4.4 Experimental Set Up and Simulation

In this work for performing bug localization using topic models, an open source library called MALLET has been used. MALLET is a Java-based package for statistical natural language processing, document classification, clustering, topic modeling, information extraction, and other machine learning applications to text. This library provides almost various machine learning algorithms which include Classification techniques, sequence tagging, topic modeling and graph models.

There are two main phases viz. constructing the PAM model for source code and querying that model for a given query. Using MALLET, the PAM model for the source code has been constructed. MALLET library does not provide option for querying the PAM model. For this purpose, MALLET library has been extended to inference the PAM model using the proposed inference engine. This work is an incremental contribution to the existing MALLET library.

Source codes of Rhino and ModeShape software act as input for the first phase of bug localization. Thereafter bugs and their details of Rhino and ModeShape, taken from Bugzilla and JIRA respectively act as input for the second phase.

Steps to perform Phase I:

Source codes of Rhino and Modeshape are saved separately in directories. MALLET batch file takes the directory path as an input, processes all the files present under the directory and creates a binary output file in the format described by MALLET. Figure 10 describes the various steps involved in phase I. Therefore, in the first phase, the source code path has been passed into the MALLET batch file which returned the binary MALLET file as an output. In the next step of the first phase, the PAM model from the MALLET data file generated in the previous step is trained. After this, the topic-words list is generated.

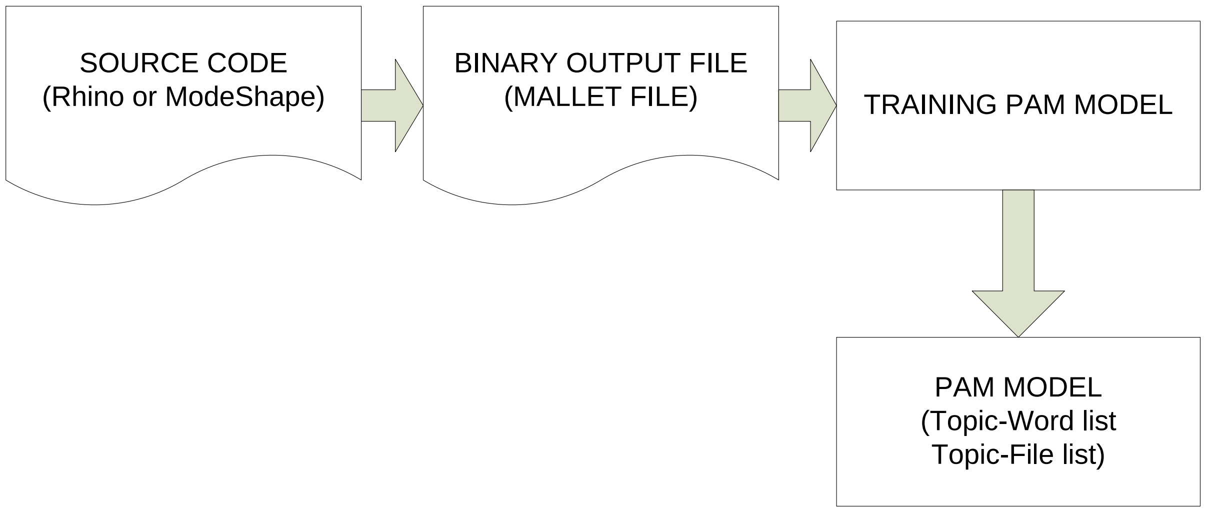


Figure 10: Phase I in PAM based bug localization

Steps to perform Phase II:

For performing phase II, Inference engine has been proposed. In MALLET, inferencing can be performed for LDA model but not for PAM model. Therefore, in this work, Inference engine has been developed for querying the PAM model built during phase I. After the first phase, the list of topics(super topic and sub topic) corresponding to each file has already been generated.

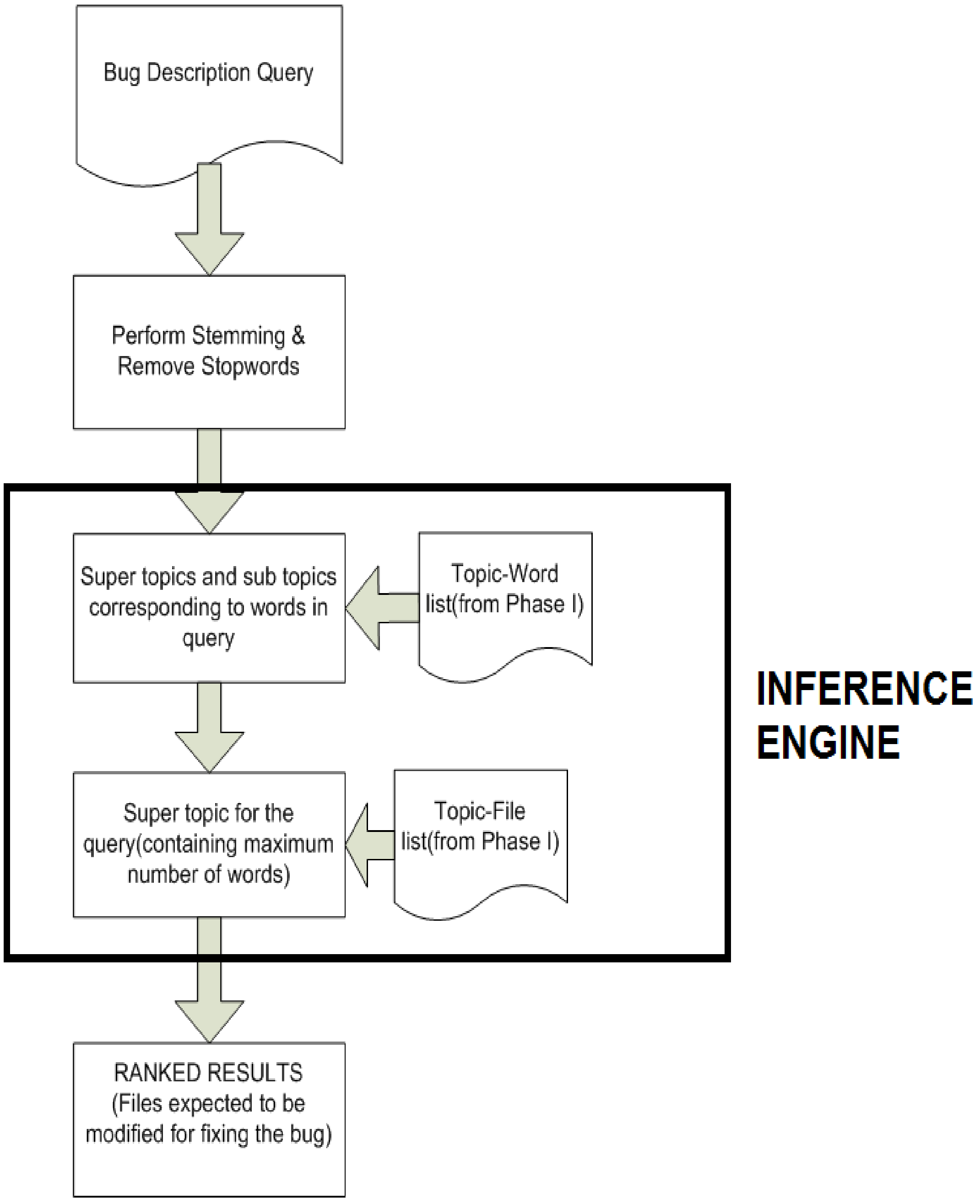

Bug Description Query
Perform Stemming & Remove Stopwords
Super topics and sub topics corresponding to words in query
Topic-Word list(from Phase I)
INFERENCE ENGINE
Super topic for the query(containing maximum number of words)
Topic-File list(from Phase I)
RANKED RESULTS (Files expected to be modified for fixing the bug)

Figure 11: Phase II in PAM based bug localization

Figure 11 shows the various steps involved in phase II. In the second phase of the PAM based bug localization, 20 bugs of Rhino & Modeshape software have been collected from Bugzilla and JIRA respectively. These bugs and their complete description are saved in separate files. These files contain the bug description as well as the steps to reproduce each bug. This acted as the query for the Inference engine.

The various steps in phase II are:

1. The bug files acting as query are entered one by one into the Inference engine.
2. The stop words are removed and stemming is done on the query.
3. The words in the query are read one by one and the corresponding topic is found (using topic-words list fetched in phase I).
4. The number of words per topic is counted to get the topic with which maximum number of words belonged to.
5. The topic found in the step 3, becomes the topic of particular query.
6. Now using the topic-file list, one can know the number of files under that topic.

In this way the list of expected files to be modified is generated in decreasing order of probability.

CHAPTER 5

EXPERIMENTAL RESULTS AND ANALYSIS

5 EXPERIMENTAL RESULTS AND ANALYSIS

In this chapter, performance of LDA and PAM based approaches for bug localization has been compared. Two case studies Rhino and ModeShape have been used for this task. Bug localization using LDA and PAM models has been done using MALLET library in Java. This chapter discusses and compares the results of LDA and PAM based bug localization on Rhino and ModeShape respectively.

5.1 Results for Rhino

For Rhino, it has been observed that value of Mean Average Precision (MAP) is 0.157 in case of LDA based approach for bug localization. While performing bug localization using PAM, value of MAP becomes 0.202. This comparison has been clearly shown in Figure 12. It clearly shows that the MAP of PAM based approach is better than LDA based approach for bug localization.

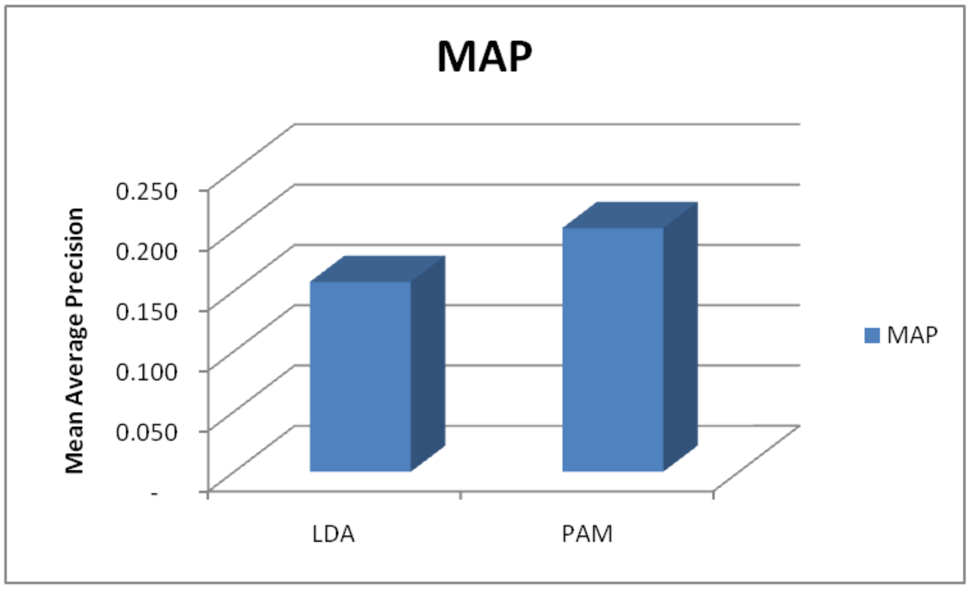


Figure 12: Comparison between LDA and PAM approaches using MAP for Rhino

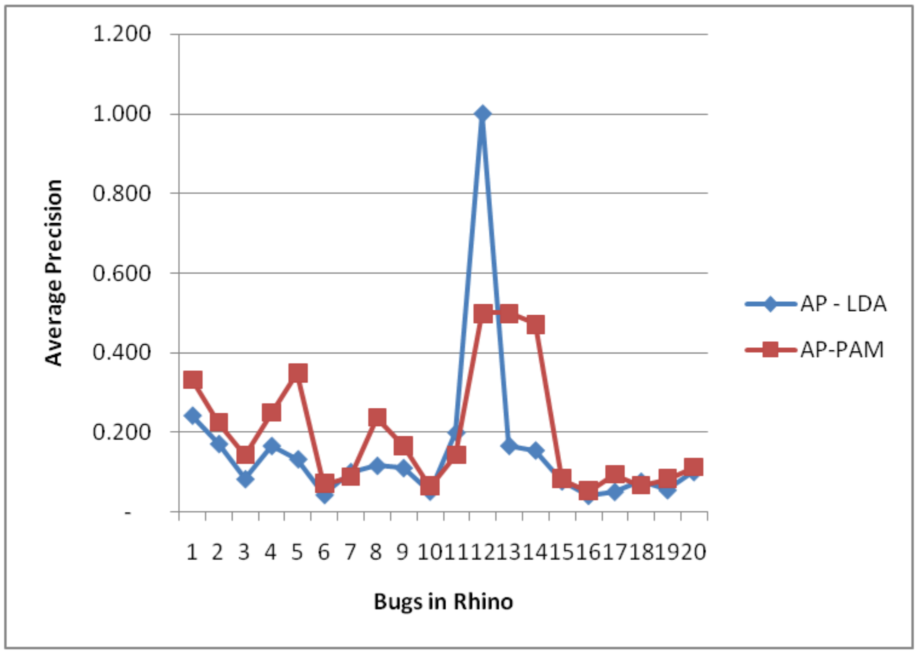


Figure 13: AP values in Rhino data set for LDA and PAM based approaches

Figure 13 shows the different values of average precision calculated for bugs separately. In LDA based approach, 20% of bugs are located at Rank less than 5.

28% of bugs are located at rank between 6 to 10 and 50% of bugs are located at rank between 11to 20. Figure 14 illustrates the bugs located with the respective rank ranges.

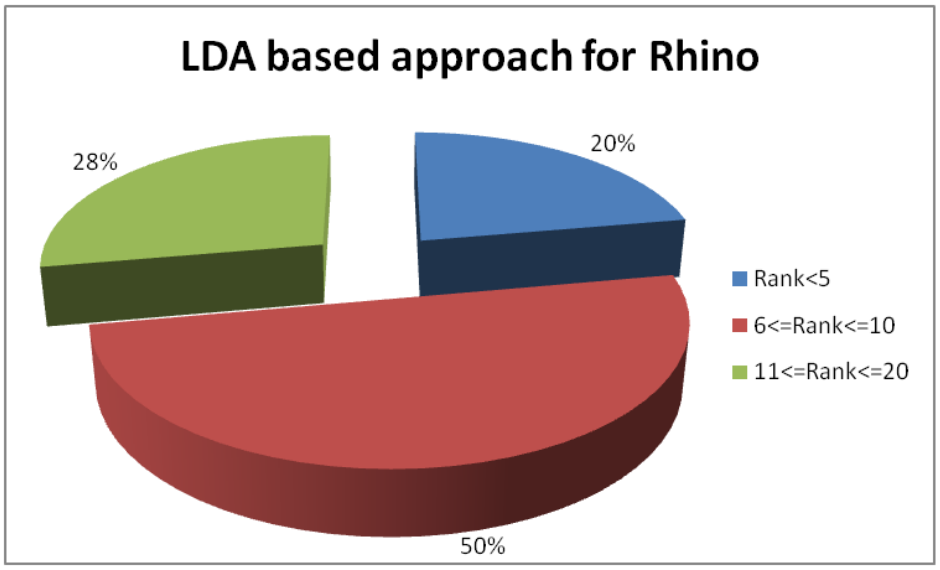


Figure 14: Rank of Relevant files using LDA approach for Rhino

While in PAM based approach for bug localization, 35% of bugs are located at Rank less than 5. 25% of bugs are located at rank between 6 to 10 and 40% of bugs are located at rank between 11 to 20. Figure 15 illustrates the bugs located with the respective rank ranges.

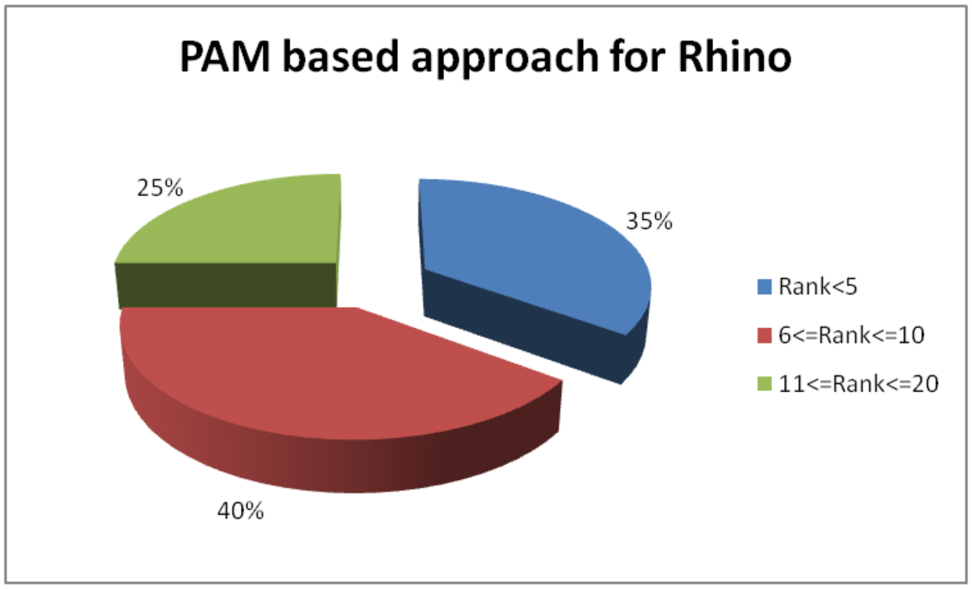


Figure 15: Rank of Relevant files using PAM approach for Rhino

It can be clearly seen from the results that for Rhino, PAM based approach has performed better than LDA based approach for bug localization, for both MAP and Ranking metrics.

5.2 Results for ModeShape

For ModeShape, it has been observed that value of Mean Average Precision (MAP) is 0.100 in case of LDA based bug localization. While performing bug localization using PAM, value of MAP becomes 0.142.

This comparison has been clearly shown in figure 16. It shows that the MAP of PAM based approach for bug localization is better than LDA based approach for bug localization.

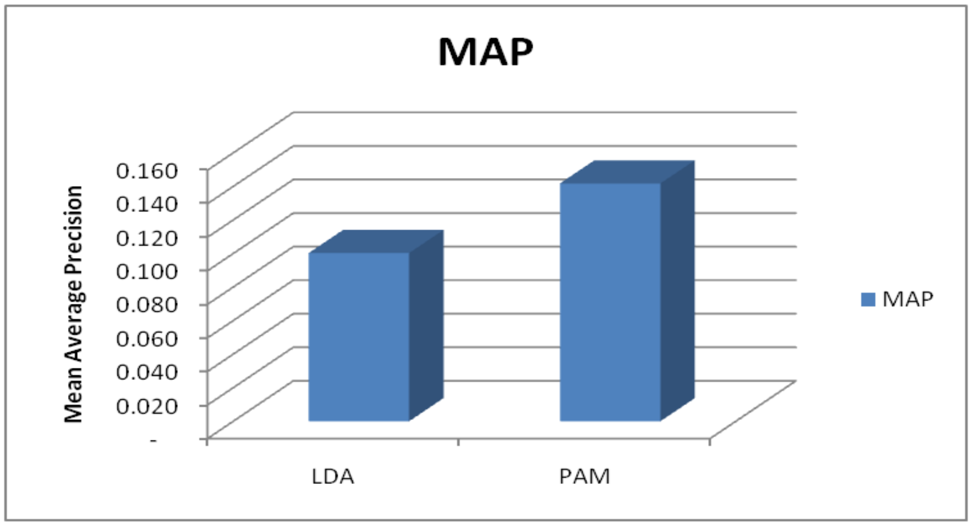


Figure 16: Comparison between LDA and PAM approaches using MAP for ModeShape

Figure 17 shows the different values of average precision calculated for bugs separately.

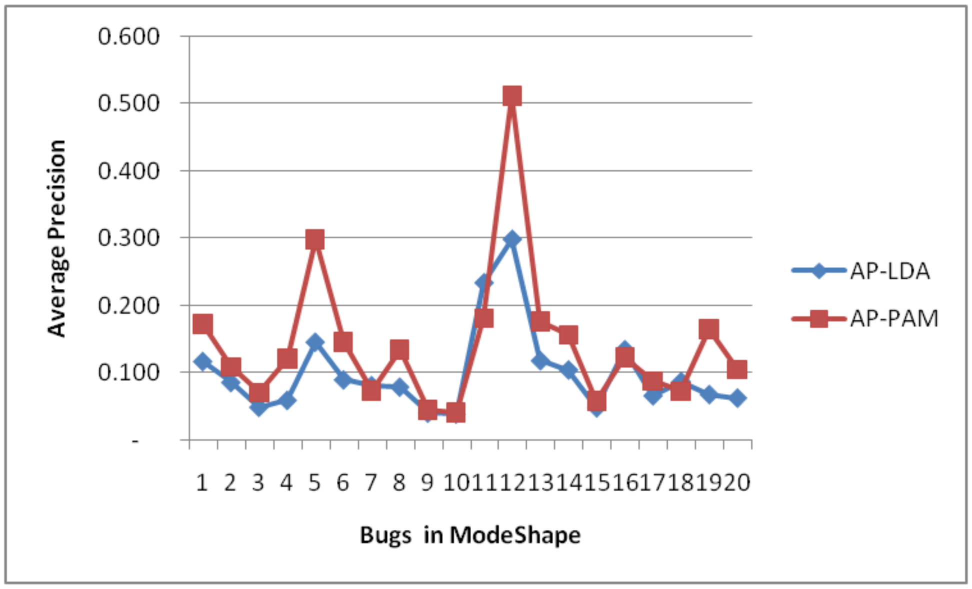


Figure 17: AP values in ModeShape data set for LDA and PAM based approaches

In LDA based approach, 17% of bugs are located at Rank less than 5. 39% of bugs are located at rank between 6 to 10 and 44% of bugs are located at rank between 11to 20. Figure 18 illustrates the bugs located with the respective rank ranges.

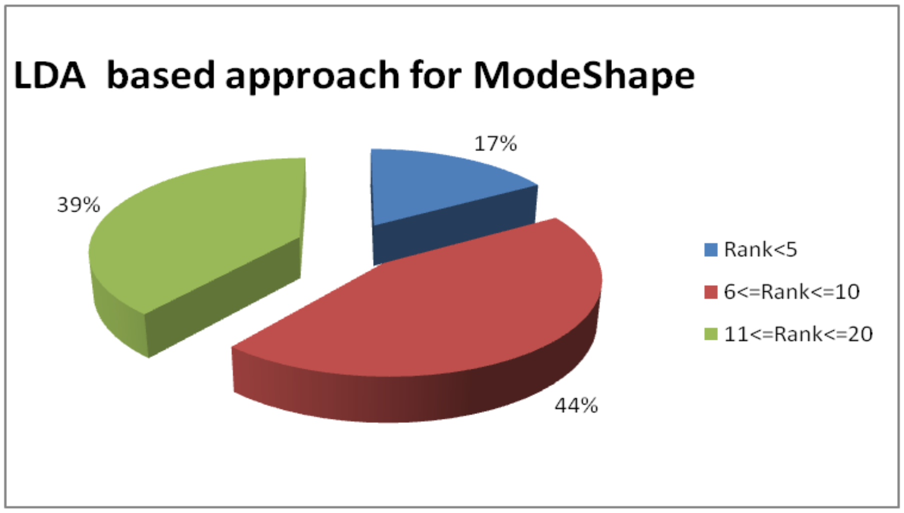


Figure 18: Rank of Relevant files using LDA for ModeShape

In PAM based approach for bug localization, 42% of bugs are located at Rank less than 5.

26% of bugs are located at rank between 6 to 10 and 32% of bugs are located at rank between 11 to 20. Figure19 illustrates the bugs located with the respective rank ranges.

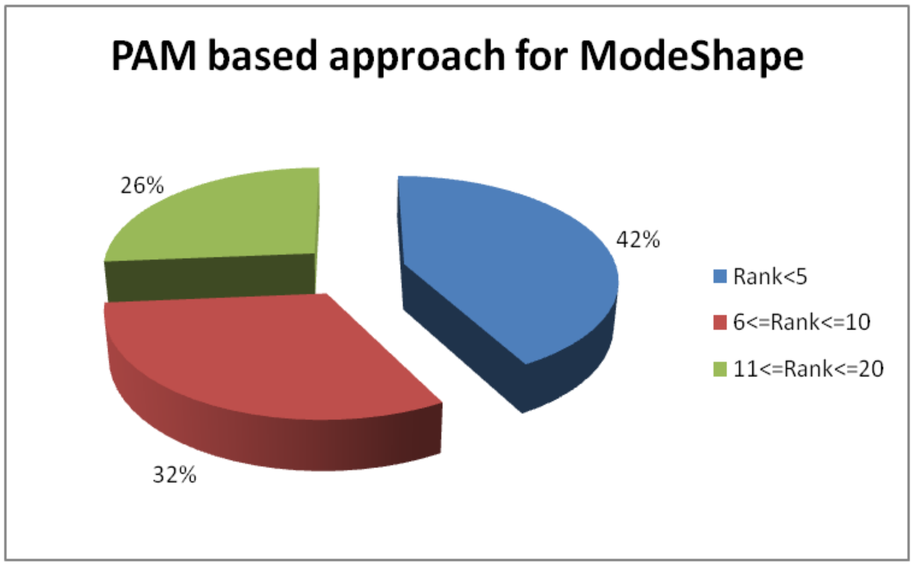


Figure 19: Rank of Relevant files using PAM for ModeShape

It can be clearly seen from the results that for ModeShape, PAM based approach has performed better than LDA based approach for bug localization, for both MAP and Ranking metrics. Figure 20 and figure 21 compares the performance of LDA and PAM based approach on both datasets using MAP and Rank metric respectively.

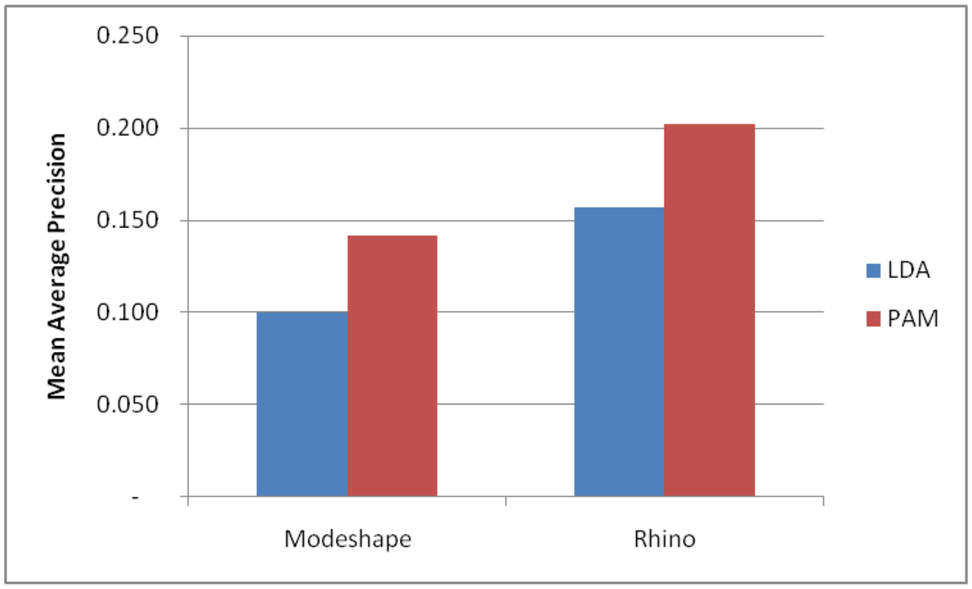


Figure 20: MAP based comparison on Rhino and ModeShape

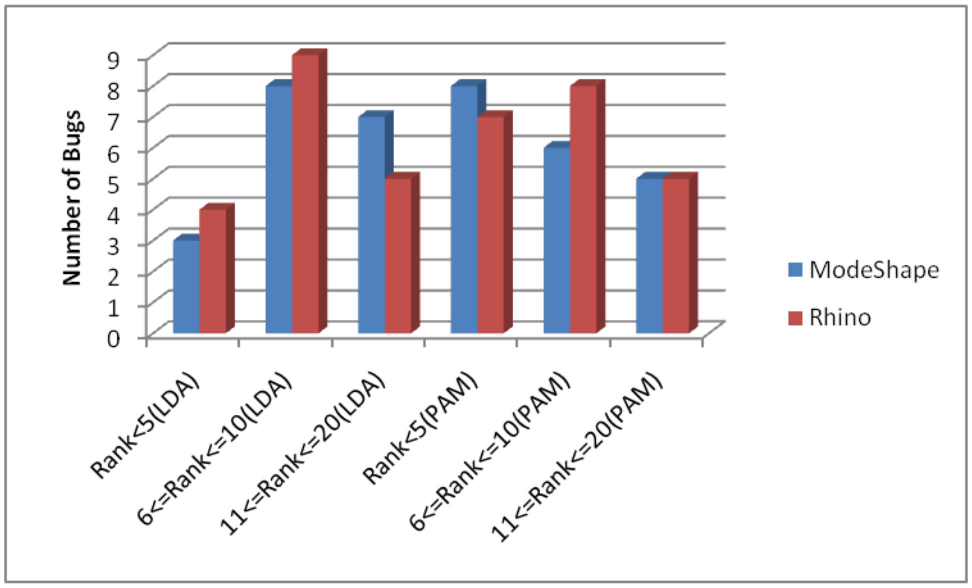


Figure 21: Rank based comparison on Rhino and ModeShape

In case of Rhino dataset, for one bug report only 10% of dataset is needed to be reviewed. In case of ModeShape dataset, for one bug report only 1.5 % of dataset is needed to be reviewed. For both the case studies, Rhino and ModeShape, PAM based bug localization technique has performed better than LDA based bug localization technique.

CONCLUSION

In this work, bug localization has been performed using PAM model. For the first time, PAM model has been used for the task of bug localization. PAM model based approach for bug localization has been compared with LDA model based approach for bug localization. MALLET library in java has been extended to incorporate PAM model based bug localization using proposed Inference engine. Data sets from two open source software viz. Rhino version 1.7 and ModeShape version3.1.1 have been used. For Rhino dataset, the value of MAP is 0.157 and 0.202 using LDA and PAM based approach respectively. Also percentage of bugs for which the relevant files retrieved are in top 5 position is 20% and 35% using LDA and PAM based approach respectively. For ModeShape dataset, the value of MAP is 0.100 and 0.142 using LDA and PAM based approach respectively. Also percentage of bugs for which the relevant files retrieved are in top 5 position is 17% and 42% using LDA and PAM based approach respectively. In case of Rhino dataset, for one bug report only 10% of dataset is needed to be reviewed. In case of ModeShape dataset, for one bug report only 1.5 % of dataset is needed to be reviewed. Thus, it can be concluded that performance of PAM based bug localization is better than LDA based bug localization.

SUGGESTIONS FOR FUTURE WORK

In this work, only lexical information has been focused. The performance of bug localization can be improved by focusing on both lexical and structural information. In future work, some new topic models can be used for the task of bug localization. Call graphs or any other such technique can be used to model the structural information as well for performing bug localization.

REFERENCES

Anthes, G. (Dec,2010). Topic models vs. unstructured data. *Communications of the ACM*, (pp. 16–18,).

Anvik, J., Hiew, L., & Murphy, G. C. (2006). Who should fix the bug? *28th international conference on Software engineering (ICSE '06),*, (pp. 361-370). Shanghai, China.

Asuncion, H. U., Asuncion, A. U., & Taylor, R. N. (2010). Software traceability with topic modeling. *Proceedings of the 32nd International Conference on Software Engineering*, (pp. 95-104).

Baldi, P. F., Lopes, C. V., Linsteda, E. J., & Bajracharya, S. K. (2008). A theory of aspects as latent topics. *ACM* , 543–562.

Blei, D. M., & Lafferty, J. D. (2009). Topic models. In *Text Mining: Classification, Clustering, and Applications* (pp. 71-94). London,UK: Chapman & Hall.

Blei, D. M., & McAuliffe, J. (2008). Supervised topic models. *Advances in Neural Information Processing System , 20*, 121-128.

Blei, D. M., Ng, A. Y., & Jordan, M. I. (2003). Latent Dirichlet Allocation. (J. Lafferty, Ed.) *Journal of Machine Learning Research , 3*, 993-1022.

Blei, D., Griffiths, T. L., Jordan, M. I., & Tenenbaum, J. B. (2004). Hierarchical topic models and the nested Chinese restaurant process. *Advances in neural information processing systems* .

Blei, David M. (2012). Probabilistic Topic Models. *Commuications of the ACM , 55*, 77-84.

Chang, J., & Blei, D. M. (2009). Relational topic models for document networks. *Proceedings of the 12th International Conference on Artificial Intelligence and Statistics*, *9*, pp. 81-89.

Chang, K. H., Bertacco, V., & Markov, I. L. (2005). Simulation-based bug trace minimization with BMC-based refinement. *2005 IEEE/ACM International conference on Computer-aided design (ICCAD '05)*, (pp. 1045-1051). San Jose, CA.

Deerwester, S. C., Dumais, S. T., Landauer, T. K., Furnas, F. W., & Harshman, R. A. (1990). Indexing by latent semantic analysis. *Journal of the American Society for Information Science , 41*, 391-407.

Gethers, M., & Poshyvanyk, D. (2010). Using relational topic models to capture coupling among classes in object-oriented software systems. *Proceedings of the 26th International Conference on Software Maintainence*, (pp. 1-10).

Grant, S., Cordy, J. R., & Skillicorn, D. (2008.). Automated concept location using independent component analysis. In15th Working Conference on Reverse Engineering., (pp. 138–142).

Griffiths, T. L., & Steyvers, M. (2004). Finding scientific topics. *National Academy of Sciences,*, (pp. 5228–5235).

Hanley, J. (2012, February). *History of Issue Tracking System.* Retrieved from blog.pmrobot.com: http://blog.pmrobot.com/2012/02/history-of-issue-tracking-systems.html

Hayes, C. J., Nichols, B., Kraft, N. A., & Anderson, M. D. (n.d.). Improving LSI-Based Bug Localization using Historical Patch data. *The University of Alabama McNair Journal* .

Hiemstra, D. (2009). Information Retrieval Models. In A. Goker, & J. Davies, *Information Retrieval: Searching in the 21st century.* John Wiley and Sons, Ltd.

Hoffman, T. (1999). Probabilistic Latent Semantic Indexing 22nd International Conference on Research and Development in Information Retrieval., (pp. 50–57).

http://www.wisegeek.com/what-is-debugging.htm. (n.d.). Retrieved from Wisegeek.com.

Katz, I. R., & Anderson, J. R. (1987). *Debugging: an analysis of bug- location strategies, Human Computer Interaction.*

Lal, S., & Sureka, A. (2012). A Static Technique for Fault Localization Using Character. *ISEC '12, Feb. 22-25* (pp. 109-118). Kanpur: ACM.

Li, W., & McCallum, A. (2006). Pachinko Allocation:DAG-Structured Mixture Models of Topic Correlations. *23 rd International Conference on Machine Learning.* Pittsburgh,PA.

Linstead, E., & Baldi, P. (2009). Mining the coherence of GNOME bug reports with statistical topic models. *Proceedings of the 6th Working Conference on Mining Software Repositories*, (pp. 99-102).

Linstead, E., Lopes, C., & Baldi, P. (2008). Proceedings of the 26th International Conference on Software evolution. *Proceedings of the 7th International Conference on Machine Learning and Applications*, (pp. 813-818).

Linstead, E., Rigor, P., Bajracharya, S., Lopes, C., & Baldi, P. (2007). Mining concepts from code with probabilistic topic models. *Proceedings of the 22nd International Conference on Automated Software Engineering*, (pp. 461-464).

Linstead, E., Rigor, P., Bajracharya, S., Lopes, C., & Baldi, P. (2007). Mining Eclipse developer contributions via author-topic models. *Proceedings of the 4th International Workshop on Mining Software Repositories*, (pp. 30-33).

Liu, Y., Poshyvanyk, D., Ferenc, R., Gyimothy, T., & Chrisochoides, N. (2009). Modeling class cohesion as mixture of latent topics. *Proceedings of the 25th International Conference on Software Maintenance*, (pp. 233-242).

Lukins, S. K., Kraft, N. A., & Etzkorn, L. H. (2008). Source code retrieval for bug localization. *2008 15th Working Conference on Reverse* (pp. 155-164). IEEE.

Maskeri, G., Sarkar, S., & Heafield, K. (2008). Mining business topics in source code using latent Dirichlet allocation. *Proceedings of the 1st conference on India software engineering conference*, (pp. 113-120).

McCallum, A. K. (2002). *MAchine Learning for Language Toolkit*. Retrieved from http://mallet.cs.umass.edu.

Mimno, D., Wallach, H. M., Naradowsky, J., Smith, D. A., & McCallum, A. (2009). Polylingual topic models. *Proceedings of the 2009 Conference on Empirical Methods in Natural Language Processing*, (pp. 880-889).

Paul, M. (2009.). *Cross-Collection Topic Models: Automatically Comparing and Contrasting Text.* University of Illinois at Urbana-Champaign, Urbana.

Poshyvanyk, D., & Marcus, A. (2007). Combining formal concept analysis with information retrieval for concept location in source code. *Proceedings of the 15th IEEE International Conference on Program Comprehension (ICPC)* (pp. 37-48). Banff, Alberta, Canada: IEEE.

Rakesh. (n.d.). *Bug Life Cycle*. Retrieved from www.softwaretestinghelp.com: http://www.softwaretestinghelp.com/?attachment_id=98

Ramage, D., Hall, D., Nallapati, R., & Manning, C. D. (2009). Labeled LDA: a supervised topic model for credit attribution in multi-labeled corpora. *Proceedings of the 2009 Conference on Empirical Methods in Natural Language Processing*, (pp. 248-256).

Rao, S., & Kak, A. (2011). Retrieval from software libraries for bug localization: a comparative study of generic and composite text models. *8th working conference on Mining software repositories, MSR '11,* (pp. 43-52). Honolulu,Hawaii: ACM.

Rhino. (n.d.). Retrieved from http://www.mozilla.org/rhino/.

Rosen-Zvi, M., Griffiths, T., Steyvers, M., & Smyth, P. (2004). The author-topic model for authors and documents. *Proceedings of the 20th conference on Uncertainty in artificial intelligence*, (pp. 487-494).

Sangeeta. (2011). *A Static Technique for Bug Localization Using Character N-Gram Based Information Retrieval Model.* Indraprastha Institute of Information Technology, Delhi, Delhi.

Savage, T., Dit, B., Gethers, M., & Poshyvanyk, D. (2010). TopicXP: exploring topics in source code using latent dirichlet allocation. *Proceedings of the 26th International Conference on Software Maintenance*, (pp. 1-6).

Shao, P. (2011). *Combining Information Retrieval Modules and Structural Information for Source Code Bug Localization and Feauture location.* Graduate School of University of Albama, Computer Science, Tuscaloosa, Alabama.

Sisman, B., & Kak, A. C. (2012). Incorporating Version Histories in Information Retrieval Based Bug Localization. *MSR* (pp. 50-59). Zurich: IEEE.

Steyvers, M., & Griffiths, T. (2007). Probabilistic topic models. In *Latent Semantic Analysis: A Road to Meaning.* Laurence Erlbaum.

Thomas, S. W. (2012). *Mining Software Repositories with Topic Models.* Queem's University, School of Computing, Ontario,Canada.

Thomas, S. W., Adams, B., Hassan, A. E., & Blostein, D. (2010). Validating the use of topic models for software evolution. *Proceedings of the 10th International Working Conference on Source Code Analysis and Manipulation*, (pp. 55-64).

Wei, X. (2007). *Topic Models in Information Retrieval.* University of Massachusetts Amherst, Computer Science.

Wei, X., & Croft, B. (2006). LDA-Based Document Models for Ad-hoc Retrieval. *29th Annual International ACM SIGIR Conference on Research & Development on Information Retrieval*, (pp. 178-185). Seattle,USA.

Wong, W. E., & Debroy, V. (2009). *A Survey of Software Fault Localization.* The University of Texas at Dallas, Department of Computer Science, Dallas.

Zachary, F. P. (2012). *Fault Localization Using Textual Similarities.* MCS Thesis, University of virginia.

Zhai, C. (October,2007). A Brief Review of Information Retrieval Models.

Zhou, J., Zhang, H., & Lo, D. (2012). Where Should the Bugs Be Fixed? *ICSE 2012* (pp. 14-24). Zurich, Switzerland: IEEE.