Mining Unstructured Software Repositories Using IR Models

by

Stephen W. Thomas

A thesis submitted to the

School of Computing

in conformity with the requirements for

the degree of Doctor of Philosophy

Queen's University

Kingston, Ontario, Canada

December 2012

Abstract

MINING SOFTWARE REPOSITORIES, which is the process of analyzing the data related to software development practices, is an emerging field which aims to aid development teams in their day to day tasks. However, data in many software repositories is currently unused because the data is *unstructured*, and therefore difficult to mine and analyze. Information Retrieval (IR) techniques, which were developed specifically to handle unstructured data, have recently been used by researchers to mine and analyze the unstructured data in software repositories, with some success.

The main contribution of this thesis is the idea that *the research and practice of using IR models to mine unstructured software repositories can be improved by going beyond the current state of affairs*. First, we propose new applications of IR models to existing software engineering tasks. Specifically, we present a technique to prioritize test cases based on their IR similarity, giving highest priority to those test cases that are most dissimilar. In another new application of IR models, we empirically recover how developers use their mailing list while developing software.

Next, we show how the use of advanced IR techniques can improve results. Using a framework for combining disparate IR models, we find that bug localization performance can be improved by 14–56% on average, compared to the best individual IR model. In addition, by using topic evolution models on the history of source code, we can uncover the evolution of source code concepts with an accuracy of 87–89%.

Finally, we show the risks of current research, which uses IR models as black boxes without fully understanding their assumptions and parameters. We show that *data duplication* in source code has undesirable effects for IR models, and that by eliminating the duplication, the accuracy of IR models improves. Additionally, we find that in the bug localization task, an unwise choice of parameter values results in an accuracy of only 1%, where optimal parameters can achieve an accuracy of 55%.

Through empirical case studies on real-world systems, we show that all of our proposed techniques and methodologies significantly improve the state-of-the-art.

Acknowledgments

First, I would like to offer deep and sincere gratitude to my co-supervisors, Ahmed E. Hassan and Dorothea Blostein. Leading by example, Ahmed pushed me to my limits. He provided keen insight into the academic world and helped me make sense of it all. He gave me resources, a sense of belonging, and encouragement to pursue my ideas. Dorothea brought experienced yet fresh perspective on all my work, incredible encouragement, and solid technical direction. She believed in me, helped me, and kept me on track. I am truly grateful for the one-two punch of Ahmed and Dorothea that was easily the most significant factor of my success.

I would also like to thank my committee members, Jim Cordy, Juergen Dingel, Prem Devanbu, Greg Lessard, and Pat Martin. Through expert, well-delivered critique as well as stimulating interaction, they helped me to shape my thesis and the ideas within.

I give thanks to my collaborators and coauthors: Bram Adams, Nicolas Bettenburg, Hadi Hemmati, Meiyappan Nagappan, and of course Ahmed E. Hassan and Dorothea Blostein. Draft after draft, experiment after experiment, they helped me navigate the complexities of statistics, $\LaTeX 2_\varepsilon$ formatting, deadlines, related research, R code, and everything else that is needed to release a scientific idea into the wild.

The Software Analysis and Intelligence Laboratory (SAIL), of which I was a member since January 2010, was paramount to my success and defined my PhD experience. The evolving cohort of undergraduates, Master's students, doctoral students, post doctoral researchers, and visiting scholars provided always-available interaction, feedback, motivation, and support. (And whether someone lives in the basement of the SAIL building or not, we may never know.)

I would like to thank my Master's supervisor and continued collaborator, Rick Snodgrass. Although he was not directly involved in this thesis, he has been a constant mentor in my academic life for more than five years. His sound advice, exceptional work ethic, brilliant ideas, and friendship have meant a great deal to me.

Finally, the love and support of my friends and family, both near and far, are what got me here in the first place, and keep me going to the next. A most special thanks goes to my wife, Tandy. Without her, I would be lost. Her love fuels my day.

Dedication

For Tandy and Addison, my special ladies

Table of Contents

List of Tables

List of Figures

Related Publications

In all chapters of this thesis, and all related publications, I contributed in the following ways: bringing the initial idea to the table; researching background material and related work; collecting the necessary data for experiments and analysis (authoring tools and scripts if necessary); conducting the experiments and analyses (authoring tools and scripts if necessary); and writing and polishing all sections of the chapter and paper (including making all figures and tables). My coauthors supported me in refining my initial ideas, pointing me towards additional related work if necessary, and providing feedback on drafts of the chapters and papers.

Parts of this thesis have been published as follows:

1. **Stephen W. Thomas.** *Mining Software Repositories Using Topic Models.* In: Proceedings of the 33rd International Conference on Software Engineering (Doctoral Symposium), pages 1138-1139, 2011. [Chapter 1.]

2. **Stephen W. Thomas**. *Mining Software Repositories with Topic Models.* Technical Report No. 2012-586, School of Computing, Queen's University. 39+iv pages. February 2012. [Chapter 2.]

3. **Stephen W. Thomas**, Hadi Hemmati, Ahmed E. Hassan, and Dorothea Blostein. *Static Test Case Prioritization Using Topic Models.* Empirical Software Engineering. 30 pages. Accepted July 8, 2012. To appear. [Chapter 3.]

4. **Stephen W. Thomas**, Nicolas Bettenburg, Dorothea Blostein, and Ahmed E. Hassan.

Talk and Work: Recovering the Relationship between Mailing List Discussions and Development Activity. Empirical Software Engineering. 24 pages. Second revision under preparation. [Chapter 4.]

5. **Stephen W. Thomas**, Meiyappan Nagappan, Dorothea Blostein, and Ahmed E. Hassan. *The Impact of Classifier Configuration and Classifier Combination on Bug Localization.* IEEE Transactions on Software Engineering. 16 pages. Submitted May 2012. [Chapters 5 and 8.]

6. **Stephen W. Thomas**, Bram Adams, Ahmed E. Hassan, and Dorothea Blostein. *Validating the Use of Topic Models for Software Evolution.* In: Proceedings of the 10th International Working Conference on Source Code Analysis and Manipulation, pages 55-64, 2010. [Chapter 6.]

7. **Stephen W. Thomas**, Bram Adams, Ahmed E. Hassan, and Dorothea Blostein. *Studying Software Evolution Using Topic Models.* Science of Computer Programming. 23 pages. Accepted August 20, 2012. To appear. [Chapter 6.]

8. **Stephen W. Thomas**, Bram Adams, Ahmed E. Hassan, and Dorothea Blostein. *Modeling the Evolution of Topics in Source Code Histories.* In: Proceedings of the 8th Working Conference on Mining Software Repositories, pages 173-182, 2011. [Chapter 7.]

List of Notation and Abbreviations

α: in LDA, a smoothing parameter for document-topic distributions

β: in LDA, a smoothing parameter for topic-term distributions

θ: the document-topic matrix of a corpus

θ_d: the topic membership vector of document d

ϕ: the topic-term matrix of a corpus

A: the term-document matrix of a corpus

C: a corpus

d: a document

K: in LDA, the number of topics; in LSI, the reduction factor

N: the number of terms in a document

V: the vocabulary of a corpus

w: a term

z: a topic

APFD: Average Percentage of Faults Detected

AOP: Aspect Oriented Computing

CVS: Concurrent Versions System

EM: Entity Metric

ICA: Independent Component Analysis

IR:	Information Retrieval
LDA:	Latent Dirichlet Allocation
LOC:	Lines of Code
LSA:	Latent Semantic Analysis
LSI:	Latent Semantic Indexing
MSR:	Mining Software Repositories
NLP:	Natural Language Processing
PCA:	Principal Component Analysis
PLSA:	Probabilistic Latent Semantic Analysis
PLSI:	Probabilistic Latent Semantic Indexing
SLOC:	Source Lines of Code
SUT:	System Under Test
SVD:	Singular Value Decomposition
SVN:	Subversion
TA:	Testing Ability
TCP:	Test Case Prioritization
UML:	Unified Modeling Language
VCS:	Version Control System
VSM:	Vector Space Model

Part I

Prologue

CHAPTER 1

Introduction

Publications based on this chapter: Thomas (2011)

SOFTWARE DEVELOPMENT presents enormous challenges for both developers and managers (Glass, 2003; Selby, 2005). One of the primary difficulties lies in the ever-increasing complexity of the ever-changing source code. Changes lead to complexity and bugs, which in turn lead to skyrocketing maintenance costs and unhappy customers (Erlikh, 2000; Hassan, 2009; Lehman, 1980; Moad, 1990).

Researchers in software engineering have attempted to improve software development by mining and analyzing software repositories, such as source code changes, email archives, bug databases, and execution logs (Godfrey et al., 2008; Hassan, 2008). Research shows that interesting and practical results can be obtained from mining these repositories, allowing developers and managers to better understand their systems and ultimately increase the quality of their products in a cost effective manner (Tichy, 2010). Particular success has been experienced with *structured* repositories, such as source code, execution traces, and change logs.

However, automated techniques to understand the *unstructured* textual data in software repositories are still relatively immature (Hassan, 2008), even though 80–85% of the data is unstructured (Blumberg and Atre, 2003; Grimes, 2008). Unstructured data is a current research challenge because the data is often unlabeled, vague, and noisy (Hassan, 2008). For example, the Eclipse bug database contains the following bug report titles:

– "`NPE caused by no spashscreen handler service available`" (#112600)

– "`Provide unittests for link creation constraints`" (#118800)

– "`jaxws unit tests fail in standalone build`" (#300951)

This data is *unlabeled* and *vague* because it contains no explicit links to the source code entity to which it refers, or even to a topic or task from some pre-defined ontology. Instructions such as "link creation constraints," with no additional information or pointers, are ambiguous at best. The data is *noisy* due to misspellings and typographical errors ("spashscreen"), unconventional acronyms ("NPE"), and multiple phrases used for the same concept ("unittests", "unit tests"). The sheer size of a typical unstructured repository (for example, Eclipse has received an average of 115 new bug reports a day for the last 10 years), coupled with its lack of structure, makes manual analysis extremely challenging and in many cases impossible. The end result is that this unstructured data is still waiting to be mined and analyzed.

Despite the challenges mentioned above, mining unstructured repositories has the potential to benefit software development teams in several ways. For example, linking emails to the source code entities that they discuss could provide developers access to the design decisions made about that entity. Determining which source code entities are related to a new bug report would significantly reduce the maintenance effort required to fix the bug. Automatically creating labels for source code entities would allow developers to more easily browse and understand the code, understand how certain concepts are changing over time, and uncover relationships between entities. All of these tasks would help decrease maintenance costs, increase software quality, and ultimately yield pleased, paying customers.

Advances in the field of Information Retrieval (IR), such as the development of *statistical topic models* (Blei and Lafferty, 2009; Blei et al., 2003; Griffiths et al., 2007), have helped make sense of unstructured data in other research communities, including the social sciences (Griffiths et al., 2007; Ramage et al., 2009) and computer vision (Barnard et al., 2003). IR models, such as the Vector Space Model (VSM) (Salton et al., 1975), Latent Semantic Indexing (LSI) (Deerwester et al., 1990) and latent Dirichlet allocation (LDA) (Blei et al., 2003), are models that automatically discover structure within an unstructured corpus of documents, using the statistical properties of its word frequencies. IR models can be used to index, search, cluster, summarize, and infer links within the corpus, all tasks that were previously manually performed or not performed at all.

In addition to discovering structure, IR models are promising for several reasons. The models require no training data, which makes them easy to use in practical settings (Blei et al., 2003). The models operate directly on the raw, unstructured text without expensive data acquisition or preparation costs. (The textual data is often preprocessed, for example by removing common English-language stop words and removing numbers and punctuation, but these steps are fast and simple (Marcus et al., 2004).) Most models, even generative statistical models like LDA, are fast and scalable to millions of documents in real time (Porteous et al., 2008). Some IR models are well equipped to handle both synonymy and polysemy, as explained in Section 2.2. Finally, all IR models can be applied to any text-based software repository, such as the identifier names and comments within source code, bug reports in a bug database, email archives, execution logs, and test cases.

Indeed, researchers are beginning to use IR models to mine software repositories. Recent studies focus on concept mining (e.g., Cleary et al., 2008; Grant et al., 2008; Marcus et al., 2004, 2005; Poshyvanyk and Marcus, 2007; Poshyvanyk et al., 2006; Revelle et al., 2010; van der Spek et al., 2008), constructing source code search engines (e.g., Bajracharya and Lopes, 2010; Tian et al., 2009), recovering traceability links between artifacts (e.g., Antoniol et al., 2008; Asuncion et al., 2010; de Boer and van Vliet, 2008; De Lucia et al., 2004, 2007; Hayes et al., 2006; Jiang et al., 2008; Lormans and Van Deursen, 2006; Lormans et al., 2006; Marcus and Maletic, 2003; McMillan et al., 2009), calculating source code metrics (e.g., Bavota et al., 2010; Gall et al., 2008; Gethers and Poshyvanyk, 2010; Kagdi et al., 2010; Linstead and Baldi, 2009; Liu et al., 2009; Marcus et al., 2008; Ujhazi et al., 2010), and clustering similar documents (e.g., Kuhn et al., 2005, 2007, 2008, 2010;

Lin et al., 2006; Maletic and Marcus, 2001; Maletic and Valluri, 1999).

Although recent studies have shown promising results, we performed a detailed analysis of the literature (Section 2.4) and found several limitations. In particular, we find that most studies to date:

– focus on only a limited number of software engineering tasks;
– use only basic IR models; and
– treat IR models as black boxes without fully understanding their underlying assumptions and parameter values.

In this thesis, we present techniques and empirical case studies that in aggregate help to alleviate these limitations and advance the state-of-the-art of using IR models in software engineering research and practice.

1.1 Thesis Statement

Our goal in this thesis it to encourage researchers and practitioners to go beyond the current state of affairs when using IR models to mine software repositories. We thus formulate our thesis statement as follows:

> *"The research and practice of using IR models to mine software repositories can be improved by (i) considering additional software engineering tasks, such as prioritizing test cases; (ii) using advanced IR techniques, such as combining multiple IR models; and (iii) better understanding the assumptions and parameters of IR models."*

1.2 Thesis Overview and Organization

We organize the thesis into five parts.

Part I provides introductory material and reviews the current literature in using IR models to mine software repositories.

Part II demonstrates ***new applications of IR models*** to existing software engineering tasks. By doing so, we hope to encourage researchers to move beyond the current, limited set of software engineering tasks, and explore how IR models can help tackle other tasks. We apply IR models to two new tasks. The first task, presented in Chapter 3, is test case prioritization, which aims to uncover an ordering of a test suite that can detect as many faults in the source code as early as possible. We apply an IR model to the linguistic content of the test suite to calculate the similarity between test cases, and give highest priority to those test cases which are most dissimilar. We find that such a technique outperforms existing coverage-based prioritization techniques. The second application, presented in Chapter 4, measures how developers use email to help them develop software. We apply a statistical topic model jointly to the source code history and mailing list archives of a software system. We define metrics that measure the amount of activity a certain topic experiences in the source code and mailing list over time, and use these metrics to quantify the information flow between the mailing list and source code. We find that there is indeed a strong relationship between the two repositories, and our results can be used to help managers guide documentation and training efforts and monitor project status, and can help developers recover past email discussions pertaining to a given source code entity.

Part III presents ***advanced IR techniques*** applied to unstructured software repositories. In Chapter 5 we present a framework to combine any number of disparate IR models. We conduct a case study in the context of bug localization, and find that model combination can significantly improve localization performance, even when the constituent models have poor performance. In Chapter 6, we use an advanced IR technique—the Hall topic evolution model—as a means to automatically analyze the evolution of source code concepts. We find that the Hall model is able to accurately describe the actual changes made by developers, giving managers and developers the ability to better monitor their source code and answer questions such as "Who is working on what concepts" and "What concepts changed since last week?"

Part IV examines the ***assumptions and parameters*** of IR models and argues that we should move away from treating IR models as black boxes. In Chapter 7, we show that current research violates an implicit assumption of IR models, specifically that data will not be duplicated over time. We propose a new model, called the Diff model, which properly satisfies the IR model's assumption by removing any duplication. We show that the Diff model produces more accurate topic evolutions and is therefore a better tool for software development teams. In Chapter 8, we show that data preprocessing steps and IR model parameters are critically important, even though they are often ignored by current research. In the context of bug localization, we perform a large empirical case study to investigate how sensitive the models are to these design decisions, and which design decisions are best.

Finally, Part V provides concluding remarks, summarizes the thesis contributions, and outlines future research directions.

1.3 Contributions of Thesis

The conceptual contribution of this thesis is the argument that we, as a community of software engineering researchers and practitioners, need to move beyond the current state of affairs in using IR models to mine unstructured software repositories, and move towards new applications, better IR techniques, and better understanding the parameters and assumptions of IR models. The technical contributions of this thesis focus on the development of tools and the invention of techniques to show how to realize our overarching goal of moving beyond the state of the art. The empirical contributions of this thesis are the application of all proposed techniques on several long-lived, real-world open source systems.

The main contributions of this thesis can be summarized as follows.

- **Performing software engineering applications that are novel for IR models, i.e.,**
  - proposing and evaluating a technique to prioritize test cases; and
  - proposing and evaluating a technique to analyze the interaction of source code and mailing lists.
- **Using advanced IR techniques on software repositories, i.e.,**
  - proposing and evaluating a framework for combining disparate IR models; and
  - describing and evaluating an advanced IR technique to analyze source code histories.

- **Going beyond the black box of off-the-shelf IR models, i.e.,**
  - proposing and evaluating a technique that overcomes the data duplication problem in large source code histories; and
  - analyzing the sensitivity of IR models to data preprocessing and IR model parameters.

Further contributions of this thesis include:

- A review of IR models and their current uses in software engineering (Sections 2.2–2.3)
- Publicly-available datasets corresponding to each case study in the thesis (available online (Thomas, 2012))

CHAPTER 2

Background and State of the Art

IN THIS CHAPTER, we first describe the field of mining software repositories. We then introduce information retrieval models, starting with the seminal work in the 1980s on the Vector Space Model. We then provide a chronological summary of previous research that has used IR models to mine software repositories. Finally, we analyze the trends and shortcomings of previous research.

2.1 Mining Software Repositories

Mining Software Repositories (MSR) is a field of software engineering research which aims to analyze and understand the data repositories related to software development. The main goal of MSR is to make intelligent use of these software repositories to support the decision-making process of software development (Godfrey et al., 2008; Hassan, 2004, 2008; Hassan and Holt, 2005).

Software development produces several types of repositories during its lifetime, detailed in the following paragraphs. Such repositories are the result of the daily interactions between the stakeholders, as well as the evolutionary changes to various software artifacts, such as source code, test cases, bug reports, requirements documents, and other documentation. These repositories offer a rich, detailed view of the path taken to realize a software system, but they must be transformed from their raw form into something usable (Godfrey et al., 2008; Hassan, 2008; Hassan and Xie, 2010; Tichy, 2010; Zimmermann et al., 2005). A prime example of mining software repositories is *bug prediction*. By mining the characteristics of source code entities (such as size, complexity, number of changes, and number of past bugs), researchers have shown how to accurately predict which entities are likely to have future bugs and therefore deserve more quality control resources.

2.1.1 Types of Software Repositories

We now describe the most common types of software repositories. These repositories contain a vast array of information about different facets software development, from human communication to source code evolution.

Source Code *Source code* is the executable specification of a software system's behavior (Lethbridge et al., 2005). The source code repository consists of a number of *documents* or *files* written in one or more programming languages. Source code documents are generally grouped into logical entities called *packages* or *modules*. While source code contains structured data (e.g., syntax, program semantics, control flow), it also contains unstructured data, such as comments, identifier names, and string literals. This unstructured portion of source code, even without the aid of the structured portion, has been shown to help determine the high-level functionality of the source code (Kuhn et al., 2007).

Bug and Vulnerability Databases A *bug database* (or *bug-tracking system*) maintains information about the creation and resolution of bugs, feature enhancements, and other software maintenance tasks (Serrano and Ciordia, 2005). Typically, when developers or users experience a bug in a software system, they make a note of the bug in the bug database in the form of an *issue*, which includes such information as what task they were performing when the bug occurred, how to reproduce the bug, and how critical the bug is to the functionality of the system. Then, one or more maintainers of the system investigate the issue, and if they resolve the issue, they close the issue. All of these tasks are captured in the bug database. Popular bug database systems include Bugzilla (Mozilla Foundation, 2012a) and Trac (Edgewall Software, 2012), although many exist.

A *vulnerability database* stores a list of information security vulnerabilities and exposures (Neuhaus and Zimmermann, 2010). The goal of a vulnerability database is to document and share data (e.g., current vulnerabilities being found in large systems) between communities and applications.

Mailing Lists and Chat Logs *Mailing lists* (or *discussion archives*), along with the *chat logs* (or *chat archives*) are archivals of the textual communication between developers, managers, and other project stakeholders (Shihab et al., 2010a). The mailing list is usually comprised of a set of time-stamped email messages, which contain a *header* (containing the sender, receiver(s), and time stamp), a *message body* (containing the text content of the email), and a set of *attachments* (additional documents sent with the email). The chat logs contain the record of the instant-messaging conversations between project stakeholders, and typically contain a series of time-stamped, author-stamped text messages (Bettenburg et al., 2009; Shihab et al., 2009a,b).

Revision Control Database A *revision control database* maintains and records the history of changes (or edits) to a repository of documents. Developers typically use revision control databases to maintain the edits to the source code of the system. Popular revision control databases (such as Concurrent Versions System (CVS) (Berliner, 1990), Subversion (SVN) (Pilato et al., 2008)), or Git (Bird et al., 2009b; Software Freedom Conservancy, 2012) allow developers to: checkout a copy of the global repository to their local file system; make local changes to existing documents, add new documents, delete existing documents, or alter the directory structure of the repository; and commit these local changes to the global repository.

Requirements and Design Documents *Requirements documents*, usually written in conjunction with (or with approval from) the customer, are documents that list the required behavior of the software system (Lethbridge et al., 2005). The requirements can be categorized as either *functional*, which specify the "*what*" of the behavior of the program, or *non-functional*, which describe the qualities of the software (e.g., reliability or accessibility).

Design documents are documents that describe the overall design of the software system, including architectural descriptions, important algorithms, and use cases. Design documents can take the form of diagrams (especially UML diagrams (Fowler and Scott, 2000)) or free-flowing text.

Execution Logs An *execution log* is a document that logs the output of a system during its execution of one or more predefined test cases. The log generally contains a listing of which methods were called at which times, the values of certain variables, and other details about the state of the execution. Execution logs are useful when debugging the performance of large-scale systems with thousands or millions of concurrent users, since individual bugs are difficult to recreate on demand.

Software System Repositories A *software system repository* is a collection of (usually open source) software systems. These collections often contain hundreds or thousands of systems whose source code can easily be searched and downloaded for use by interested parties. Popular repositories include SourceForge (Geeknet, 2012) and Google Code (Google, 2012).

2.1.2 Structured vs. Unstructured Data in Software Repositories

The term "unstructured data" is difficult to define and its usage varies in the literature (Bettenburg and Adams, 2010; Manning et al., 2008). For the purposes of this thesis, we adopt the definition given by Manning et al. (2008):

"Unstructured data is data which does not have clear, semantically overt, easy-for-a-computer structure. It is the opposite of structured data, the canonical example of which is a relational database, of the sort companies usually use to maintain product inventories and personnel records."

Unstructured data usually refers to natural language text, since such text has no explicit data model. Most natural language text indeed has latent structure, such as parts-of-speech, named entities, relationships between words, and word sense, that can be inferred by humans or advanced machine learning algorithms. However, in its raw, unparsed form, the text is simply a collection of characters with no structure and no meaning to a data mining algorithm. Examples of unstructured data in software repositories include: bug report titles and descriptions; source code linguistic data (i.e., identifier names, comments, and string literals); requirements documents; descriptions and comments in design documents; mailing lists and chat logs; and source control database commit messages.

Structured data, on the other hand, has a data model and a known form. Examples of structured data in software repository include: source code parse trees, call graphs, inheritance graphs; execution logs and traces; bug report metadata (e.g., author, severity, date); source control database commit metadata (e.g., author, date, list of changed files); and mailing list and chat log metadata.

2.2 Information Retrieval Models

"Information retrieval (IR) is finding material (usually documents) of an unstructured nature (usually text) that satisfies an information need from within large collections (usually stored on computers)."

—Manning et al. (2008, p. 1)

IR is used to find specific documents of interest in a large collection of documents. Usually, a user enters a query (i.e., a text snippet) into the IR system, and the system returns a list of documents related to the query. For example, when a user enters the query

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

(a) Document d_1.

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

(b) Document d_2.

There are numerous studies that examine whether or not cloned code is harmful to software systems. Yet, few of these studies study which characteristics of cloned code in particular lead to software defects (or faults).

(c) Document d_3.

Figure 2.1: A sample corpus of three documents.

"software engineering" into the Google IR system, it searches every webpage ever indexed and returns those that are somehow related to software engineering.

IR *models*—the internal workings of IR systems—come in many forms, from basic keyword-matching models to statistical models that take into account the location of the text in the document, the size of the document, the uniqueness of the matched term, and even whether the query and document contain shared topics of interest (Zhai, 2008).

Common Terminology

IR models share a common vernacular, which we summarize below. To make the discussion more concrete, we use a running example of a corpus of three simple documents shown in Figure 2.1.

term (word or token) w**:** a string of one or more alphanumeric characters.

In our example, we have a 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.

document d**:** an ordered set of N terms, $w_1, \ldots, w_N$.

In our example, we have 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.

query q**:** an ordered set of $|q|$ terms created by the user, $q_1, \ldots, q_{|q|}$.

In our example, a user might query for "defects" (with $|q|$=1 term) or "cloned code" (with $|q|$=2 terms).

corpus C: an unordered set of n documents, $d_1, \ldots, d_n$.

In our example, there is one corpus, which consists of $n = 3$ documents: d_1, d_2, and d_3.

vocabulary V: the unordered set of m unique terms that appear in a corpus.

In our example, the vocabulary consists of $m = 71$ unique terms across all three documents: *code, of, are, that, to, the, software,* $\ldots$.

term-document matrix A: an $m \times n$ matrix whose i^{th}, j^{th} entry is the weight of term w_i in document d_j (according to some weighting function, such as term-frequency).

In our example, we have

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

indicating that, for example, the term *code* appears in document d_1 with a weight of 3, and the term *are* appears in document d_2 with a weight of 2.

topic (concept) z: an m-length vector of probabilities over the vocabulary of a corpus.

In our example, we might have a topic

$$\begin{array}{r|cccccccc} & \textit{code} & \textit{of} & \textit{are} & \textit{that} & \textit{to} & \textit{the} & \textit{software} & \ldots \\ \hline z_1 = & 0.25 & 0.10 & 0.05 & 0.01 & 0.10 & 0.17 & 0.30 & \ldots \end{array}$$

indicating that, for example, when a term is drawn from topic z_1, there is a 25% chance of drawing the term *code* and a 30% chance of drawing the term *software*. (This example assumes a generative model, such as PLSI or LDA. See Section 2.2 for the full definitions.)

topic membership vector θ_d: For document d, a K-length vector of probabilities of the K topics.

In our example, we might have a topic membership vector

$$\begin{array}{r|ccccc} & z_1 & z_2 & z_3 & z_4 & \dots \\ \hline \theta_{d_1} = & 0.25 & 0.0 & 0.0 & 0.70 & \dots \end{array}$$

indicating that, for example, when a topic is selected for document d_1, there is a 25% chance of selecting topic z_1 and a 70% chance of selecting topic z_3 .

document-topic matrix θ (also called document-topic matrix D): an n by K matrix whose i^{th}, j^{th} entry is the probability of topic z_j in document d_i. Row i of θ corresponds to θ_{d_i}.

In our example, we might have a document-topic matrix

$$\theta = \begin{array}{c|ccccc|} & z_1 & z_2 & z_3 & z_4 & \dots \\ \hline d_1 & 0.25 & 0.0 & 0.0 & 0.70 & \dots \\ d_2 & 0.0 & 0.0 & 0.0 & 1.0 & \dots \\ d_3 & 0.1 & 0.4 & 0.2 & 0.0 & \dots \end{array}$$

indicating that, for example, document d_3 contains topic z_3 with probability 20%.

topic-term matrix ϕ (also called topic-term matrix T): a K by m matrix whose i^{th}, j^{th} entry is the probability of term w_j in topic z_i. Row i of ϕ corresponds to z_i.

In our example, we might have a topic-term matrix:

$$\phi = \begin{array}{c|cccccccc} & \textit{code} & \textit{of} & \textit{are} & \textit{that} & \textit{to} & \textit{the} & \textit{software} & \dots \\ \hline z_1 & 0.25 & 0.10 & 0.05 & 0.01 & 0.10 & 0.17 & 0.30 & \dots \\ z_2 & 0.0 & 0.0 & 0.0 & 0.05 & 0.2 & 0.0 & 0.05 & \dots \\ z_3 & 0.1 & 0.04 & 0.2 & 0.0 & 0.07 & 0.10 & 0.12 & \dots \\ \dots & \dots & & & & & & & \end{array}$$

Some common issues arise with any language model:

synonymy: Two terms w_1 and w_2, $w_1 \neq w_2$, are *synonyms* if they possess similar semantics.

homonymy: A term w is a *homonym* if it has multiple semantics.

We note that the term *semantic* is hard to define and takes on different meanings in different contexts. In the field of IR, often a manually-created oracle is used (e.g., WordNet (Miller, 1995)) to determine the semantics of a term (i.e., relationships with other terms).

The Boolean Model

The Boolean model treats each document as a set of words, and allows queries with single keywords coupled with Boolean expressions to combine keywords (i.e., `AND`, `OR`, and `NOT`). Using the three example documents in Figure 2.1, the query "faults `OR` defects" will match document d_1 (because it contains the word "faults") and d_3 (because it contains the word "defects"), but not d_2 (because it contains neither "faults" nor "defects"). The query "faults `AND` defects" will not match any documents, because none of them contain both words.

While simple to understand and implement, the Boolean model suffers from many drawbacks. First, too many documents may be matched if overly-general words are present in the query (e.g., "it" or "the"). Second, too few documents may be matched if the query is poorly formulated (e.g., "automobile" instead of "car"). Third, it is difficult to rank the matched documents by how relevant they might be, since all terms are weighted equally.

The Vector Space Model

The Vector Space Model (VSM) is a simple algebraic model based on the term-document matrix of a corpus (Salton et al., 1975). VSM represents documents by their column vector in the term-document matrix: a vector containing the weights of the words present in the document, and 0s otherwise. The similarity between two documents (or between a query and a document) is calculated by comparing the similarity of the two vectors in the term-document matrix. Example vector similarity measures include Euclidean distance, cosine distance, Hellinger distance, or KL divergence. In VSM, two documents will only be deemed

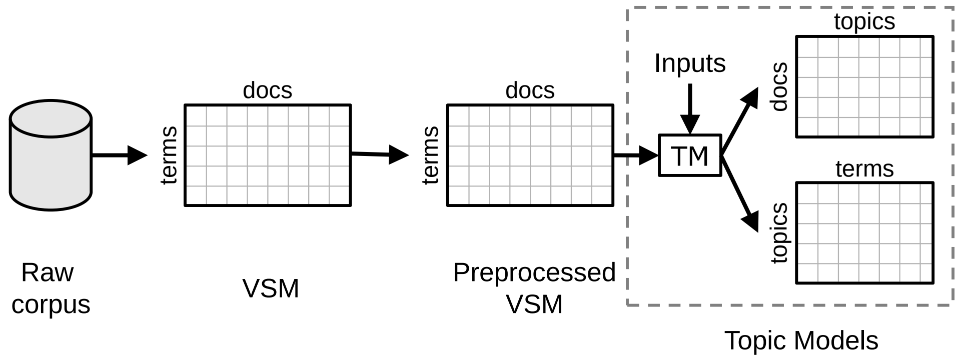


Figure 2.2: The path from a raw corpus to a basic IR model or a more advanced topic model (TM). Here, “Topic Models” includes LSI, ICA, PLSI, LDA, and all LDA variants.

similar if they contain at least one shared term; the more shared terms they have, and the higher the weight of those shared terms, the higher the similarity score will be.

VSM improves over the Boolean model in two ways. First, it allows for term weighting schemes to be used, such as tf-idf (term frequency, inverse document frequency) weighting. Weighting schemes help to downplay the influence of common terms in the query and provide a boost to documents that match rare terms in the query. Another improvement is that the relevance between the query and a document is based on vector similarity measures, which is more flexible than the strict Boolean model.

Topic Models

A *topic model* (or *latent topic model*) is an IR model designed to automatically extract *topics* from a corpus of text documents (Anthes, 2010; Blei and Lafferty, 2009; Steyvers and Griffiths, 2007; Zhai, 2008). Here, a topic is 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*}. Due to the nature of language use, the terms that constitute a topic are often semantically related (Blei et al., 2003).

Topic models were originally developed 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. By representing documents by the lower-dimensional topics, as opposed to terms, topic models (i) uncover *latent* semantic relationships and (ii) allow faster analysis on text (Zhai, 2008). Figure 2.2 shows the general process of creating basic IR models and more advanced topic models from a raw corpus; we describe several topic modeling techniques below.

Latent Semantic Indexing *Latent Semantic Indexing* (LSI) (or *Latent Semantic Analysis* (LSA)) is an information retrieval model that extends the VSM by reducing the dimensionality of the term-document matrix by means of *Singular Value Decomposition* (SVD) (Deerwester et al., 1990). During the dimensionality reduction phase, terms that are related (in terms of co-occurrence) will be grouped together into topics[1]. This noise-reduction technique has been shown to provide increased performance over VSM in terms of dealing with polysemy and synonymy (Baeza-Yates and Ribeiro-Neto, 1999).

SVD is a factorization of the original term-document matrix A that reduces the dimensionality of A by isolating the *singular values* of A (Salton and McGill, 1983). Since A is likely to be very sparse, SVD is a critical step of the LSI approach. SVD decomposes A into three matrices: $A = TSD^T$, where T is an m by $r = rank(A)$ term-topic matrix, S is the r by r singular value matrix, and D is the n by r document-topic matrix.

LSI augments the reduction step of SVD by choosing a reduction factor, K, which is typically much smaller than the rank of the original term-document matrix r. Instead of reducing the input matrix to r dimensions, LSI reduces the input matrix to K dimensions. There is no perfect choice for K, as it is highly data- and task-dependent. In the literature, typical values range between 50–300.

As in VSM, terms and documents are represented by row and column vectors, respectively, in the term-document matrix. Thus, two terms (or two documents) can be compared by some distance measure between their vectors (e.g., cosine similarity) and queries can

[1]The creators of LSI call these reduced dimensions “concepts”, not “topics”. However, to be consistent with other topic modeling approaches, we will use the term “topics”.

by formulated and evaluated against the matrix. However, because of the reduced dimensionality of the term-document matrix after SVD, these measures are more equipped to deal with noise in the data.

Independent Component Analysis *Independent Component Analysis* (ICA) (Comon, 1994) is a statistical technique used to decompose a random variable into statistically independent components (i.e., dimensions). Although not generally considered a topic model, it has been used in similar ways to LSI to model source code documents in a K-dimensional conceptual space.

Like LSI, ICA reduces the dimensionality of the term-document matrix to help reduce noise and associate terms. However, unlike LSI, the resulting dimensions in ICA are statistically independent of one another, which helps capture more variation in the underlying data (Grant and Cordy, 2009).

Probabilistic LSI *Probabilistic Latent Semantic Indexing* (PLSI) (or *Probabilistic Latent Semantic Analysis* (PLSA)) (Hofmann, 1999, 2001) is a generative model that addresses the statistical unsoundness of LSI. Hofmann argues that since LSI uses SVD in its dimension-reduction phase, LSI is implicitly making the unqualified assumption that term counts will follow a Gaussian distribution. Since this assumption is not verified, LSI is "*unsatisfactory and incomplete*" (Hofmann, 1999).

To overcome this assumption, PLSI defines a generative latent-variable model, where the latent variables are topics in documents. At a high level, a generative model has the advantages of being evaluable with standard statistical techniques, such as model checking, cross-validation, and complexity control; LSI could not be evaluated with any of these techniques. And since the latent variables are topics in documents, PLSI is also well-equipped to more readily handle polysemy and synonymy.

The generative model for each term in the corpus can be summarized with the following steps.

– Select a document d_i with probability $P(d_i)$.

– Select a topic z_k with probability $P(z_k|d_i)$.

– Generate a term w_j with probability $P(w_j|z_k)$.

Given the observations in a dataset (i.e., terms), one can perform inference against this model to uncover the topics $z_i, ..., z_k$. We refer interested readers to the original articles (Hofmann, 1999, 2001).

As was shown in subsequent articles (e.g., Blei et al., 2003; Zhai, 2008), the generative model of PLSI suffers from at least two critical problems. First, since d is used as an index variable in the first step, the number of parameters that need to be estimated grows linearly with the size of the corpus, which can lead to severe over-fitting issues. Second, since the z_k vectors are only estimated for documents in the training set, they cannot be easily applied to new, unseen documents.

Latent Dirichlet Allocation *Latent Dirichlet Allocation* (LDA) is a popular probabilistic topic model (Blei et al., 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 is 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 and Lafferty, 2009).

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. This generative model is formulated as follows:

– Choose a topic vector $\theta_d \sim \text{Dirichlet}(\alpha)$ for document d.

– For each of the N terms w_i:

  – Choose a topic $z_k \sim \text{Multinomial}(\theta_d)$.

– 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.

The 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 for.

Like any generative model, the task of LDA is that of *inference*: given the terms in the documents, what topics did they come from (and what are the topics)? LDA performs inference with *latent variable models* (or *hidden variable models*), which are machine learning techniques devised for just this purpose: to associate observed variables (here, terms) with latent variables (here, topics). A rich literature exists on latent variable models (Bartholomew, 1987; Bishop, 1998; Loehlin, 1987); for the purposes of this thesis, we omit the details necessary for computing the posterior distributions associated with such models. It is sufficient to know that such methods exist and are being actively researched.

For the reasons presented above, it is argued that LDA's generative process gives it a solid footing in statistical rigor—much more so than previous topic models (Blei et al., 2003; Griffiths and Steyvers, 2004; Steyvers and Griffiths, 2007). As such, LDA may be better suited to discover the latent relationships between documents in a large text corpus.

Table 2.1 shows example topics discovered by LDA from version 7.5.1 of the source code of JHotDraw (Gamma, 2012), a framework for creating simple drawing applications. For each topic, the table shows an automatically-generated two-word topic label, the top (i.e., highest probable) words for the topic, and the top three matching Java classes in JHotDraw. The topics span a range of concepts, from opening files to drawing Bezier paths. The discovered topics intuitively make sense and the top-matching classes match our expectations—there seems to be a natural match between the "Bezier path" topic and the `CurvedLinear` and `BezierFigure` classes.

Label	Top words	Top 3 matching classes
file filter	*file uri chooser urichoos save filter set jfile open*	`JFileURIChooser`, `URIUtil`, `AbstractSaveUnsavedChangesAction`
tool bar	*editor add tool draw action button bar view creat*	`DrawingPanel`, `ODGDrawingPanel`, `PertPanel`
undoabl edit	*edit action undo chang undoabl event overrid*	`NonUndoableEdit`, `CompositeEdit`, `UndoRedoManager`
connect figur	*figur connector connect start end decor set handl*	`ConnectionStartHandle`, `ConnectionEndHandle`, `Connector`
bezier path	*path bezier node index mask point geom pointd*	`CurvedLiner`, `BezierFigure`, `ElbowLiner`

Table 2.1: Example topics from JHotDraw source code version 7.5.1. The labels are automatically-generated based on the most popular bigram in the topic.

Topic Evolution Models

Several advanced techniques have been proposed to extract the *evolution* of a topic in a time-stamped corpus—how the usage of a topic (and sometimes the topic itself) changes over time as the terms in the documents are changed over time. Such a model is usually an extension to a basic topic model that accounts for time in some way. We call such a model a *topic evolution model*.

Initially, the Dynamic Topic Model (Blei and Lafferty, 2006) was introduced. This model represents time as a discrete Markov process, where topics themselves evolve according to a Gaussian distribution. This model thus penalizes abrupt changes between successive time periods, discouraging rapid fluctuation in the topics over time.

The Topics Over Time (TOT) (Wang and McCallum, 2006) model represents time as a continuous beta distribution, effectively removing the penalty on abrupt changes from the Dynamic Topic Model. However, the beta distribution is still rather inflexible in that it assumes that a topic evolution will have only a single rise and fall during the entire corpus history.

The Hall model (Hall et al., 2008) applies LDA to the entire collection of documents at the same time and performs post hoc calculations based on the observed probability of each document in order to map topics to versions. Linstead et al. (2008b) and Thomas et al. (2010b) also used this model on a software system's version history. The main advantage of this model is that no constraints are placed on the evolution of topics, providing the necessary flexibility for describing large changes to a corpus.

The Link model, proposed by Mei and Zhai (2005) and first used on software repositories by Hindle et al. (2009), applies LDA to each version of the repository *separately*, followed by a post-processing phase to link topics across versions. Once the topics are linked, the topic evolutions can be computed in the same way as in the Hall model. The post-processing phase must iteratively link topics found in one version to the topics found in the previous. This process inherently involves the use of similarity thresholds to determine whether two topics are similar enough to be called the same, since LDA is a probabilistic process and it is not guaranteed to find the exact same topics in different versions of a corpus. As a result, at each successive version, some topics are successfully linked while some topics are not, causing past topics to "die" and new topics to be "born". Additionally, it is difficult to allow for gaps in the lifetime of a topic.

Applying IR to Source Code

Before IR models are applied to source code, several preprocessing steps are generally taken in an effort to reduce noise and improve the resulting models.

- Characters related to the syntax of the programming language (e.g., "`&&`", "`->`") are removed; programming language keywords (e.g., "`if`", "`while`") are removed.
- Identifier names are split into multiple parts based on common naming conventions, such as camel case (`oneTwo`), underscores (`one_two`), dot seperators (`one.two`), and capitalization changes (`ONETwo`)).
- Common English-language stopwords (e.g., "the", "it", "on") are removed. See Table 2.2 for a full list.
- Word stemming is applied to find the root of each word (e.g., "changing" becomes "chang"), typically using the Porter algorithm (Porter, 1980).
- In some cases, the vocabulary of the resulting corpus is pruned by removing words that occur in, for example, over 80% or under 2% of the documents (Madsen et al., 2004).

The main idea behind these steps is to capture the semantics of the developers' intentions, which are thought to be encoded within the identifier names and comments in the source code (Poshyvanyk et al., 2007). The rest of the source code (i.e., special syntax, language keywords, and stopwords) are just noise and will not be beneficial to the results of IR models. We have provided our preprocessing tool online (Thomas, 2012).

a able about above according accordingly across actually after afterwards again against all allow allows almost alone along already also although always am among amongst an and another any anybody anyhow anyone anything anyway anyways anywhere apart appear appreciate appropriate are around as aside ask asking associated at available away awfully b be became because become becomes becoming been before beforehand behind being believe below beside besides best better between beyond both brief but by c came can cannot cant cause causes certain certainly changes clearly co com come comes concerning consequently consider considering contain containing contains corresponding could course currently d definitely described despite did different do does doing done down downwards during e each edu eg eight either else elsewhere enough entirely especially et etc even ever every everybody everyone everything everywhere ex exactly example except f far few fifth first five followed following follows for former formerly forth four from further furthermore g get gets getting given gives go goes going gone got gotten greetings h had happens hardly has have having he hello help hence her here hereafter hereby herein hereupon hers herself hi him himself his hither hopefully how howbeit however i ie if ignored immediate in inasmuch inc indeed indicate indicated indicates inner insofar instead into inward is it its itself j just k keep keeps kept know knows known l last lately later latter latterly least less lest let like liked likely little look looking looks ltd m mainly many may maybe me mean meanwhile merely might more moreover most mostly much must my myself n name namely nd near nearly necessary need needs neither never nevertheless new next nine no nobody non none noone nor normally not nothing novel now nowhere o obviously of off often oh ok okay old on once one ones only onto or other others otherwise ought our ours ourselves out outside over overall own p particular particularly per perhaps placed please plus possible presumably probably provides q que quite qv r rather rd re really reasonably regarding regardless regards relatively respectively right s said same saw say saying says second secondly see seeing seem seemed seeming seems seen self selves sensible sent serious seriously seven several shall she should since six so some somebody somehow someone something sometime sometimes somewhat somewhere soon sorry specified specify specifying still sub such sup sure t take taken tell tends th than thank thanks thanx that thats the their theirs them themselves then thence there thereafter thereby therefore therein theres thereupon these they think third this thorough thoroughly those though three through throughout thru thus to together too took toward towards tried tries truly try trying twice two u un under unfortunately unless unlikely until unto up upon us use used useful uses using usually uucp v value various very via viz vs w want wants was way we welcome well went were what whatever when whence whenever where whereafter whereas whereby wherein whereupon wherever whether which while whither who whoever whole whom whose why will willing wish with within without wonder would would x y yes yet you your yours yourself yourselves z zero

Table 2.2: The stopword list we use in our empirical case studies throughout the thesis. We obtain the list from the MALLET tool (McCallum, 2012). The list contains a total of 521 terms.

2.3 State of the Art

In this section, we describe and evaluate related work that uses IR models to mine software repositories and perform some software engineering task. We organize the work into subsections by software engineering task. We provide a brief description of each task, followed by a chronological presentation of the relevant articles.

2.3.1 Concept/Feature 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 (Rajlich and Wilde, 2002). 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.

LSI-based Techniques

LSI was first used for the concept location task by Marcus et al. (2004), who developed a technique to take a developer query and return a list of related source code documents. The authors showed that LSI provides better results than existing methods (i.e., regular expressions and dependency graphs) and is easily applied to source code, due to the flexibility and light-weight nature of LSI. The authors also noted that since LSI is applied only to the comments and identifiers of the source code, it is language-independent and thus accessible for any system.

Marcus et al. (2005) demonstrated that concept location is needed in the case of Object-Oriented (OO) programming languages, contrary to previous beliefs. The authors compared

LSI with two other techniques, namely regular expressions and dependency graphs, for locating concepts in OO source code. The authors concluded that all techniques are beneficial and necessary, and each possesses its own strengths and weaknesses.

Poshyvanyk et al. (2006) combined LSI and Scenario Based Probabilistic ranking of execution events for the task of feature location in source code. The authors demonstrated that using the two techniques, when applied together, outperform either of the techniques individually.

Poshyvanyk and Marcus (2007) combined LSI and Formal Concept Analysis (FCA) to locate concepts in source code. LSI is first used to map developer queries to source code documents, then FCA is used to organize the results into a concept lattice. The authors found that this technique works well, and that concept lattices are up to four times more effective at grouping relevant information than simple ranking methods.

Cleary et al. (2008) compared several IR (e.g., VSM, LSI) and NLP techniques for concept location. After an extensive experiment, the authors found that NLP techniques do not offer much of an improvement over IR techniques, which is contrary to results in other communities.

van der Spek et al. (2008) used LSI to find concepts in source code. The authors considered the effects of various preprocessing steps, such as stemming, stopping, and term weighting. The authors manually evaluated the resulting concepts with the help of domain experts.

Grant et al. (2008) used ICA, a conceptually similar model to LSI, to locate concepts in source code. The authors argued that since ICA is able to identify statistically independent signals in text, it can better find independent concepts in source code. The authors showed the viability of ICA to extract concepts through a case study on a small system.

Revelle and Poshyvanyk (2009) used LSI, along with static and dynamic analysis, to tackle the task of feature location. The authors combined the different techniques in novel ways. For example, textual similarity was used to traverse the static program dependency graphs, and dynamic analysis removed textually-found methods that were not executed in a scenario. The authors found that no technique outperformed all others across all case studies.

Revelle et al. (2010) performed data fusion between LSI, dynamic analysis, and web mining algorithms (i.e., HITS and PageRank) to tackle the task of feature location. The authors found that combining all three techniques significantly outperforms any of the individual methods, and outperforms the state-of-the-art in feature location.

LDA-based Techniques

Linstead et al. (2007a) were the first to use LDA to locate concepts in source code in the form of LDA topics. Their technique 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.

Linstead et al. (2007b) applied a variant of LDA, the Author-Topic model (Rosen-Zvi et al., 2004), to source code to extract the relationship between developers (authors) and source code topics. Their 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 technique.

Maskeri et al. (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 technique is able to successfully extract business topics, implementation topics, and cross-cutting topics from source code.

Baldi et al. (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 technique to a large set of open-source systems to identify the global set of topics, as well as perform a more detailed analysis of a few specific systems. The authors found that latent topics with high scattering metrics are indeed those that are typically classified as aspects in the AOP community.

Savage et al. (2010) introduced a topic visualization tool, called Topic$_{XP}$, which supports interactive exploration of discovered topics located in source code.

2.3.2 Traceability Recovery and Bug Localization

An often-asked question during software development is: "*Which source code document(s) implement requirement X?*" *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 at all!). 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. *Bug localization* is a special case of traceability recovery in which traceability links between bug reports and source code are sought.

LSI-based Techniques

Marcus and Maletic (2003) were the first to use LSI to recover traceability links between source code and documentation (e.g., requirements documents). The authors applied LSI to the source code identifiers and comments and the documentation, then computed similarity scores between each pair of documents. A user could then specify a similarity threshold to determine the actual links. The authors compared their work to a VSM-based recovery technique and found that LSI performs at least as good as VSM in all case studies.

De Lucia et al. (2004) integrated a traceability recovery tool, based on LSI, into a software artifact management system called ADAMS. The authors presented several case studies that use their LSI-based technique to recover links between source code, test cases, and requirements documents. In subsequent work, De Lucia et al. (2006) proposed an incremental technique for recovering traceability links. In this technique, a user semi-automatically interacts with the system to find an optimal similarity threshold between documents (i.e., a threshold that properly discriminates between related and unrelated documents). The authors claimed that a better threshold results in fewer links for the developer to consider, and thus fewer chances for error, making human interaction a necessity.

Hayes et al. (2006) evaluated various IR techniques for generating traceability links between various high- and low-level requirements, concentrating on the tf-idf and LSI models. The authors implemented a tool called RETRO to aid a requirements analyst in this task. The authors concluded that, while not perfect, IR techniques provide value to the analyst.

Lormans and Van Deursen (2006) evaluated different linking strategies (i.e., thresholding techniques) for traceability recovering using LSI by performing three case studies. The authors concluded that LSI is a promising technique for recovering links between source code and requirements documents and that different linking strategies result in different results. However, the authors observed that no linking strategy is optimal under all scenarios. In subsequent work, Lormans (2007) introduced a framework for managing evolving requirements (and their traceability links) in a software development cycle. Their technique uses LSI to suggest candidate links between artifacts.

Lormans et al. (2006) used LSI for constructing *requirement views*, which are different views of requirements. For example, one requirement view might display only requirements that have been implemented. The authors implemented their tool, called ReqAnalyst, and used it on several real-world case studies.

De Lucia et al. (2007) were the first to perform a human case study, which evaluated the effectiveness of using LSI for recovering traceability links during the software development process. The authors concluded that LSI is certainly a helpful step for developers, but that its main drawback is the inevitable trade off between precision and recall.

Jiang et al. (2008) proposed an incremental technique to maintaining traceability links as a software system evolves over time. The authors' technique, called incremental LSI, uses links (and the LSI matrix) from previous versions when computing links for the current version, thus saving computation effort.

de Boer and van Vliet (2008) developed a tool to support auditors in locating documentation of interest. The tool, based on LSI, suggests to the auditor documents that are related to a given query, as well as documents that are semantically related to a given document. Such a process gives the auditor, who is unfamiliar with the documentation, a guide to make it easier to explore and understand the documentation of a system.

Antoniol et al. (2008) introduced a tool called Reuse or Rewrite (ReORe) to help stakeholders decide if they should update existing code (for example, to introduce new functionalities) or completely rewrite from scratch. ReORe achieves this by using a combination of static (LSI), manual, and dynamic analysis to create traceability links between existing requirements and source code. The stakeholders can then review the recovered traceability links to decide how well the current system implements the requirements.

McMillan et al. (2009) used both textual (via LSI) and structural (via Evolving Interoperation Graphs) information to recover traceability links between source code and requirements documents. The authors performed a case study on a small but well-understood system, `CoffeeMaker`. The authors demonstrated that combining textual and structural information modestly improves traceability results in most cases.

Comparison Studies

While the majority of researchers only evaluate their technique with respect to a single topic model, a few have directly compared the performance of multiple topic models.

Lukins et al. (2008, 2010) used LDA for bug localization. The authors first build an LDA model on the source code at the method level, using the standard preprocessing steps. Then, given a bug report, the authors compute the similarity of the text content of the bug report to all source code documents. They then return the top ranked source code documents. By performing case studies on Eclipse and Mozilla (on a total of 3 and 5 bug reports, respectively), the authors find that LDA often outperforms LSI. We note that the authors use manual query expansion, which may influence their results.

Nguyen et al. (2011) introduced a new topic model based on LDA, called BugScout, in an effort to improve bug localization performance. BugScout explicitly considers past bug reports, in addition to identifiers and comments, when representing source code documents, using the two data sources concurrently to identify key technical concepts. The authors applied BugScout to four different systems and found that BugScout improves performance by up to 20% over LDA applied only to source code.

Rao and Kak (2011) compared several IR models for bug localization, including VSM, LSI, and LDA, as well as various combinations. The authors performed a case study on a small dataset, iBUGS (Dallmeier and Zimmermann, 2007), and concluded that simpler IR models often outperform more sophisticated models.

Capobianco et al. (2009) compared the ability of four different techniques (Vector Space Model, LSI, Jenson-Shannon, and B-Spline) to recover traceability links between source code, test cases, and UML diagrams. The authors found that the B-Spline method outperforms VSM and LSI, and is comparable to the Jenson-Shannon method.

Oliveto et al. (2010) compared the effectiveness of four IR techniques for traceability recovery: Jenson-Shannon, VSM, LSI, and LDA. The authors showed that LDA provides unique dimensionality compared to the other four techniques.

Asuncion et al. (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 system. 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.

Often, the state of the art in bug prediction is advanced either by the introduction of new metrics or by the use of a previously unexplored statistical model (e.g., Kamei et al. (2010), Nguyen et al. (2010), Shihab et al. (2010b)). An entire suite of metrics have thus far been introduced, counting somewhere in the hundreds. Additionally, dozens or hundreds of statistical models have been applied with varying degrees of success.

The majority of metrics are measured directly on the code (e.g., code complexity, number of methods per class) or on the code change process (methods that are frequently changed together, number of methods per change). However, researchers have used topic models to introduce *semantic* or *conceptual* metrics, which are mostly based on the comments and keywords in the source code.

LSI-based Metrics

Marcus et al. (2008) introduced a new class cohesion metric, called the Conceptual Cohesion of Classes (C3), for measuring the cohesion of a program entity. The metric is based on the semantic information in the class, such as identifier names and comments, and is computed using LSI. Highly cohesive entities are thought to follow better design principles and are shown to correlate negatively with program faults. Bavota et al. (2010) used the C3 metric in developing an technique to support the automatic refactoring of so-called blob classes (i.e., classes that contain too much functionality and thus have a low cohesion score). Kagdi et al. (2010) used a similar metric, the conceptual similarity between pairs of source code methods, as a part of a novel change impact analysis technique.

Gall et al. (2008) extensively evaluated a suite of semantic metrics that are computed on the design and requirements documents and on the source code of a system throughout the development process. Some of the metrics are based on LSI. Through three case studies, the authors found significant correlation between metrics measured on design and requirements documents and the same metrics measured source code, providing strong evidence of the semantic similarity of these documents. The authors argued that tracking such metrics can help in the detection of problematic or suspect design decisions early in the software development process.

Ujhazi et al. (2010) defined two new conceptual metrics that measure the coupling and cohesion of methods in software systems. Both metrics are based on a method's representation in an LSI subspace. The authors compared their new metrics to an existing suite of metrics (including those of Marcus et al. (2008)) and found that the new metrics provide statistically significant improvements compared to previous metrics.

LDA-based Metrics

Linstead and Baldi (2009) applied LDA to the bug reports in the GNOME system 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 is better).

Liu et al. (2009) applied LDA to source code methods in order to compute a 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.

Gethers and 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 Statistical Debugging and Root Cause Analysis

Andrzejewski et al. (2007) performed statistical debugging with the use of Delta LDA, a variant of LDA. *Statistical debugging* is the task of identifying a problematic piece of code, given a log of the execution of the code. Delta LDA is able to model two types of topics: usage topics and bug topics. Bug topics are those topics that are only found in the logs of failed executions. Hence, the authors were able to identify the pieces of code that likely caused the bugs.

Bose and Suresh (2008) used LSI as a tool for root cause analysis (RCA), i.e., identifying the root cause of a software failure. The authors built and executed a set of test scenarios that excersized the system's methods in various sequences. Then, the authors used LSI to build a method-to-test co-occurrence matrix, which clustered tests that execute similar functionalities, helping to characterize the different manifestations of a fault.

Zawawy et al. (2010) presented a framework for reducing the size and complexity of execution logs so that the manual work performed by a log analyst is reduced during RCA. The reduction is accomplished by filtering the log by performing SQL queries and LSI queries. The authors demonstrated that LSI leads to fewer false positives and higher recall during the filtering process.

2.3.5 Software Evolution and Trend Analysis

Analyzing and characterizing how a software system changes over time, or the *software evolution* (Lehman, 1980) of a system, has been of interest to researchers for many years. Both *how* a software system changes (e.g., it grows rapidly every twelfth month) and *why* a software system changes (e.g., a bug fix) can help yield insights into the processes used by a specific software system as well as software development as a whole.

Linstead et al. (2008b) applied LDA to several versions of the source code of a system 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.

In a similar effort, Thomas et al. (2010b) 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 system documentation, providing evidence that topic evolution models provide a good summary of the software history.

Hindle et al. (2009, 2010) applied LDA to commit log messages in order to see what topics are being worked on by developers at any given time. The authors applied LDA to the commit logs in a 30 day period, and then linked successive periods together using a topic similarity score (i.e., two topics are linked if they share 8 out of their top 10 terms). The authors found LDA to be useful in identifying developer activity trends.

Neuhaus and Zimmermann (2010) used LDA to analyze the Common Vulnerabilities and Exposures (CVE) database, which archives vulnerability reports from many different sources. The authors' goal was to find the trends of each vulnerability, in order to see which are increasing and which are decreasing. The authors found that their results are mostly comparable to an earlier manual study on the same dataset.

2.3.6 Document Clustering

Document clustering is the task of grouping related documents together, usually to enhance program understanding or reduce a developer's searching effort (Kuhn et al., 2005, 2007). Documents can be clustered using any of several possible attributes, including their semantic similarity or dependency graphs.

Maletic and Marcus (2001); Maletic and Valluri (1999) first applied LSI to cluster source code documents. The authors claimed that such a clustering can improve program comprehension during the maintenance and evolutionary phases of the software development cycle. The authors found that LSI produces useful clusters and, since LSI is automated, can be of significant value to developers.

In a similar effort, Kuhn et al. (2005, 2007) introduced a tool named HAPAX for clustering source code documents. The authors extended the work by Maletic and Marcus (2001) by visualizing the resulting clusters and providing each cluster with a name based on all the words in the class, not just the class names.

Lin et al. (2006) introduced a tool called Prophecy that allows developers to search the Java API for groups of related functionalities. The authors applied LSI to the Javadocs of the Java API to find similarities in their functionalities. A developer can then search the LSI index to yield a cluster of related classes.

Kuhn et al. (2008, 2010) built a two dimensional map of a software system, where the positions of entities and distances between entities are based on their vocabularies. LSI is used to reduce the dimensionality of the document-term matrix so that similar documents can be closely aligned on the map. This *software cartography* can help developers understand the layout and relationships of their source code.

2.3.7 Organizing and Searching Software Repositories

Kawaguchi et al. (2006) presented a tool called MUDABlue for automatically organizing large collections of open-source software systems (e.g., SourceForge and Google Code) into related groups, called software categories. MUDABlue applies LSI to the identifier names found in each software system. The authors demonstrated that MUDABlue can achieve recall and precision scores above .80, compared with manually created tags of the systems.

Tian et al. (2009) developed LACT, a technique to categorize systems based on their underlying topics. This work is similar in nature to Kawaguchi et al. (2006), except this work employs LDA instead of LSI. The authors compared their technique to MUDABlue and concluded that the techniques are comparable in effectiveness.

Linstead et al. (2008a,c) introduced and used an Internet-scale repository crawler, Sourcerer, to analyze a large set of software systems. The authors applied LDA and the Author-Topic model to extract the concepts in source code and the developer contributions in source code, respectively. The authors also defined new techniques for searching for code, based on the extracted topic model. Sourcerer can be used to analyze existing systems (i.e., view most popular identifier names and LDA topics) as well as search for modules which contain desired functionality.

Poshyvanyk and Grechanik (2009) proposed a technique called S^3 for searching, selecting, and synthesizing existing systems. The technique is intended for developers wishing to find code snippets from an online repository matching their current development needs.

The technique builds a dictionary of available API calls and related keywords, based on online documentation. Then, developers can search this dictionary to find related code snippets. LSI is used in conjunction with Apache Lucene to provide the search capability.

2.3.8 Other Tasks

Marcus and Maletic (2001) were the first to detect high-level clones (Bellon et al., 2007; Rahman et al., 2012; Roy et al., 2009) of source code methods by computing the semantic similarity between pairs of methods. The authors used LSI to cluster related methods together in *concept space* (i.e., a K-dimensional representation of a document, based on the document's topic memberships), and tight clusters represents code clones. Despite low levels of precision, the authors argued that this technique is cheap and can therefore be used in conjunction with existing clone detection techniques to enhance the overall results.

Grant and Cordy (2009) used ICA to detect method clones. The authors argued that since ICA can identify more distinct signals (i.e., topics) than LSI, then the conceptual space used to analyze the closeness of two methods will be of higher effectiveness. The authors performed a small case study on the Linux `kernal` package, but do not compare their results to LSI.

Ahsan et al. (2009) aimed to create an automatic bug triaging system, which determines which developer should address a given bug report. The authors extracted the textual content from the titles and summaries of a system's bug reports and applied LSI to obtain a reduced term-document matrix. Then, various classifiers mapped each bug report to a developer, trained on previous bug reports and related developers. In the best case, this technique achieved 45% classification accuracy.

Bajracharya and Lopes (2009, 2010) applied LDA to a usage log of a popular code search engine (Koders) to analyze the user queries over time. Their goal was to determine which topics are the most popular search topics, and whether the search engine provides users with the features that they need to identify the code they want. They found LDA to be an effective tool for such a task.

Dit et al. (2008) measured the cohesion of the content of a bug report by applying LSI to the entire set of bug reports and then calculating a similarity measure on each comment within a single bug report. The authors compared their metrics to human-generated analysis of the comments and find a high similarity.

Grant and Cordy (2010) tackled the challenge of choosing the optimal number of topics to input into LDA when analyzing source code. The authors' technique varied the number of topics and used a heuristic to determine how well a particular choice is able to identify two pieces of code located in the same document. The authors concluded with general guidelines and case studies.

Wu et al. (2008) tackled the challenge of building a semantic-based Web-service discovery tool. Their technique, built on LSI, allows the automatic discovery of Web services based on concepts, rather than keywords.

2.4 Research Trends

In this section, we identify and describe the research trends in the area of mining unstructured repositories using IR models. We define a set of attributes that allow us to characterize each article presented in Section 2.3. Additionally, we define six facets of related attributes, summarized in Table 2.3.

First and foremost, we are interested in which *IR model* was primarily used in the study: LSI, LDA or some other model. (Note that if an article evaluates its proposed technique, which uses IR model X, against another technique, which uses IR model Y, we only mark the article as using model X. However, if the main purpose of an article is to compare various IR models, we mark the article with all IR models considered.) Second, we are interested in the *SE task* that was being performed. We include a range of tasks to allow a fine-grained view of that literature. Third, we document the *repository* being used in the article. Fourth, we are interested in how the authors of the article *evaluated* their technique, as some IR models are known to be difficult to objectively evaluate. Fifth, we are interested in how the corpus was *preprocessed*, as there are several proposed techniques. Finally, we are interested in which IR model *tool* was used in the article, along with which parameter values were chosen for the tool, and how they were chosen.

We processed each of the articles in our article set and assigned attribute sets to each. The results allow the articles to be summarized and compared along our six chosen facets.

The results are shown in Tables 2.4 and 2.5. Table 2.4 shows our first four facets: which IR model was used, which software engineering task was being performed, which repository was used, and how the authors evaluated their technique. Table 2.5 shows our last two facets: which preprocessing steps were taken, and what IR modeling tools and parameters were used. We now analyze the research trends of each facet.

2.4.1 Facet 1: Which IR Models Were Used?

The majority of articles that we surveyed (62%) use LSI as the primary IR model. This majority is likely due to LSI's earlier introduction than LDA (1990 vs. 2003) as well as its

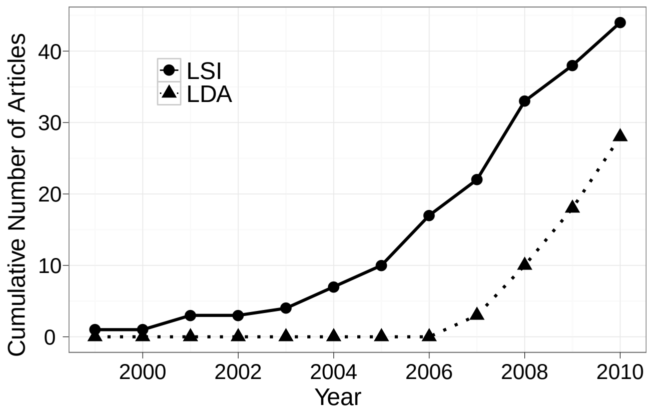


Figure 2.3: Trends of LSI and LDA use.

relative simplicity, speed, and ease of implementation.

Still, 37% of the articles used LDA or an LDA variant, indicating that LDA is indeed a popular choice. In fact, as Figure 2.3 illustrates, the use of LDA is increasing rapidly since its introduction into the software engineering field in 2006.

Most research uses LSI, and little research combines the results of different IR models to improve their overall performance.

2.4.2 Facet 2: Which Software Engineering Task Was Supported?

The most popular software engineering tasks in the articles we surveyed are concept location (31% of articles) and traceability link recovery (25% of articles). Concept location is an ideal task for IR models, since many researchers (e.g., Baldi et al. (2008)) believe that the topics discovered by a topic model are essentially equivalent (or can be directly mapped) to the programming concepts in the source code.

Traceability link recovering is another task well-suited for IR models, since the goal of traceability recovering is to find the textual similarity between pairs of documents. Thus, using the document similarity metrics defined on the topic membership vectors of two documents is a direct implementation of traceability link recovery.

The tasks in the "*other*" category include bug triaging (Ahsan et al., 2009), search engine usage analysis (Bajracharya and Lopes, 2009, 2010), auditor support for exploring the implementation of requirements (de Boer and van Vliet, 2008), analyzing bug report quality (Dit et al., 2008), estimating the number of topics in source code (Grant and Cordy, 2010), clone identification (Grant and Cordy, 2009; Marcus and Maletic, 2001), finding related code on the web (Poshyvanyk and Grechanik, 2009), and web service discovery (Wu et al., 2008).

Most research performs concept location or traceability link recovery.

2.4.3 Facet 3: Which Repositories Were Mined?

The overwhelming majority (77%) of the articles mine the source code repository, with the second most being requirements or design documents (23%). One possible reason for the popularity of mining source code is that source code is usually the only repository that is (easily) available to researchers who study open source systems. Requirements and design documents are not created as often for open source systems, and if they are, they are rarely accessible to researchers. Email archives are usually only available for large systems, and when they are available, the archives are not usually in a form that is suited for analysis without complicated preprocessing efforts, due to their unstructured nature. Execution logs are most useful for analyzing ultra-large scale systems under heavy load, which is difficult for researchers to simulate. Bug reports, although gaining in popularity, are typically only kept for large, well-organized projects. In addition, we found that most articles only considered a single snapshot of the repository, even when the repositories history was available.

The majority of prior research mines only the source code of the system, and only mines a single snapshot of the repository.

2.4.4 Facet 4: How Were the IR Models Evaluated?

The most typical evaluation method used is task-specific (51% of articles surveyed). This result makes sense, because most researchers are using IR models as a tool to accomplish some software engineering task. Hence, the researchers evaluate how well their technique performed at the given task, as opposed to how well the IR model fit the data, as is typically done in the IR community.

Perhaps surprising is that 14% of articles performed a manual evaluation of the IR modeling results—an evaluation technique that is difficult and time consuming. This may be due to the seemingly "black-box" nature of IR models—documents are input, results are output, and the rest is unknown. In articles that used topic models such as LDA, manual evaluation is deemed to be the only way to be sure that the discovered topics make sense for software repositories. This may also be due to the limited ground truth available for many software engineering tasks.

Although articles in the IR and natural language processing communities tend to only use statistical means to evaluate topic models, only a single article we surveyed used statistical evaluation techniques (Maskeri et al., 2008). This is perhaps a result of the software engineering community being task-focused, as opposed to model-focused. For example, a traceability recovery technique is often evaluated on its accuracy of connecting two documents that are known to be related, as opposed to being evaluated on the model fit on the corpus, which does not directly indicate how accurate the discovered traceability links are.

Most prior research uses task-specific or manual evaluation of the IR models.

2.4.5 Facet 5: How Was the Data Preprocessed?

Of the articles that analyzed a source code repository, 54% mention that they include the identifiers, 42% mention that they include the comments, and 14% mention that they include string literals. The relatively high percentages for identifiers and comments seems to follow the idea that the semantics of the developers' intention is captured by their choice of identifier names and comments (Poshyvanyk et al., 2007).

The majority of articles that analyzed source code created "documents" at the method level (33%), with a close second being the class level (25%). 9% of the articles left the choice of method or class as an input to their tool and reported results on both. 29% of the articles did not specify the level of granularity used.

The majority of articles that analyzed source code chose to tokenize terms (53%). One reason for this is that identifier names in source code are often written in a form that lends itself to tokenization (e.g., `camelCase` and `under_score`). By tokenizing these identifier names, the terms are being broken down into their base form and generating a larger likelihood of finding meaningful topics.

To further reduce the size of the vocabulary and increase the effectiveness of IR models, 27% of articles report stemming words, 49% report removing stop words, and 11% report pruning the vocabulary by removing overly- and/or underly-used terms.

In general, many articles were unclear as to how they preprocessed the textual data, even though this step may have a large impact on the results of IR models. For example, 30% of the articles we surveyed did not mention the document granularity of their technique (e.g., an entire class or an individual method) and 65% of the articles did not indicate whether they used word stemming, a common preprocessing step in the IR community.

Data preprocessing techniques are not well documented and are not consistent across current research.

2.4.6 Facet 6: Which Tools Were Used, and What Parameter Values Were Used?

For LDA-based articles, GibbsLDA (Phan et al., 2008) was the most frequently reported tool used (6 times). The only other tool that was used in more than one article was Dragon (Zhou et al., 2007). For LSI-based articles, no tool was used by more than on article.

Not reporting the value of K was the norm, with 49% of articles giving no indication of their choice. Of those that did, values ranged between 5 and 500, with the most frequent values being between 100 and 200.

43% of the articles that did specify the value of K did not specify *why* that particular value was used. Of the articles that did specify why, 54% came to an optimal value by testing a range of K values and evaluating each in some way (usually task-specific). A single article used an expert's opinion on what the number of topics should be ("*The number of topics for each program are chosen according to domain expert advice.*" (Andrzejewski et al., 2007)) although the actual value was not reported.

The reporting of other input parameters, such as α, β, and the number of sampling iterations (in the case of LDA) was even more scarce. 77% of the articles that used LDA did not indicate the number of iterations sampled. Of those that did, values ranged between 1000 and 3500, with 3000 being the norm.

Key study design decisions are not well documented. For instance, 49% of the articles we studied did not report the value of K (topics or reduction factor) used in the topic model, even though it is well known that this choice greatly affects the output of the IR model, and thus the results of the study.

2.4.7 Conclusions

The trends identified above reveal many potentials for advancement of the state of the art. First, only the software engineering tasks of concept location or traceability linking are typically addressed, leaving many tasks under explored. In Part II of this thesis, we go beyond these typical tasks and present two software engineering tasks that have not been previously performed with IR models.

Second, most research only uses basic IR models, such as LSI or VSM. In addition, most research only uses a single IR model, even though research in other communities indicates that combining multiple models can improve overall performance (Misirli et al., 2011). We address these problems in Part III by showing how more advanced IR models and combining the results of multiple models can help to enhance performance.

Finally, research is inconsistent as to which data preprocessing steps are performed, and most articles lack any justification as to why some steps are performed and others are not. In addition, parameter values are often not reported, and when they are, they are not justified or consistent with previous research. Most research rarely explores the sensitivity of IR models to their parameters. We tackle these problems in Part IV of the thesis.

Facet	Attribute	Description
IR Model	LSI	uses LSI
	LDA	uses standard LDA
	Other	uses ICA, PLSI, or a variant of LDA
SE Task	doc. clustering	performs a clustering of documents
	concept loc.	concept/feature location or aspect-oriented programming
	metrics	derives source code metrics (usually, but not always, for bug prediction)
	trend/evolution	analyzes/predicts source code evolution
	traceability	uncovers traceability links between pairs of artifacts (including bug localization)
	bug predict./debug	predicts bugs/faults/defects in source code, uses statistical debugging techniques, or performs root cause analysis
	org./search coll.	operates on collections of systems (search, organize, analyze)
	other	any other SE task, including bug triaging and clone detection
Repository	source code	uses source code, revision control repository, or software system repository
	email	uses email, chat logs, or forum postings
	req./design	uses requirements or design documents
	logs	uses execution logs or search engine logs
	bug reports	uses bug reports or vulnerability reports
Evaluation	statistical	uses IR modeling statistics, like log likelihood or perplexity
	task specific	uses an task specific method (e.g., classification accuracy)
	manual	performs a manual evaluation
	user study	conducts a user study
Preprocessing	identifiers	includes source code identifiers
	comments	includes source code comments
	string literals	includes string literals in source code
	tokenize	splits camelCase and under_scores
	stem	stems the terms in the corpus
	stop	performs stop word removal
	prune	removes overly common or overly rare terms from vocabulary
Tool Use	tool	name of the used IR model implementation
	K value	for LDA and LSI, the value chosen for the number of topics, K
	K justif.	justification given for the chosen K
	iterations	number of sampling iterations run (if LDA or LDA variant)

Table 2.3: The final set of attributes we collected on each article.

	IR model			task								repository					evaluation			
	LSI	LDA	other	doc. clustering	concept loc.	metrics	trend/evolution	traceability	bug pred./debug	org./search coll.	other	source code	email	req./design	logs	bug reports	statistical	task specific	manual	user study
Ahsan et al. (2009)	o	.	.	.	.	.	.	.	.	.	o	.	.	.	.	o	.	o	.	.
Andrzejewski et al. (2007)	.	o	.	.	.	.	.	.	o	.	.	.	.	.	o	.	.	o	.	.
Antoniol et al. (2008)	o	.	.	.	.	.	.	o	.	.	.	o	.	o	.	.	.	.	o	.
Asuncion et al. (2010)	.	o	.	.	.	.	.	o	.	.	.	o	o	o	.	o	.	o	.	.
Bajracharya and Lopes (2009)	.	o	.	.	.	.	.	.	.	.	o	.	.	.	o	.	.	.	.	.
Bajracharya and Lopes (2010)	.	o	.	.	.	.	.	.	.	.	o	.	.	.	o	.	.	.	.	.
Baldi et al. (2008)	.	o	.	.	o	.	.	.	.	o	.	o	.	.	.	.	.	.	.	.
Bavota et al. (2010)	o	.	.	.	.	o	.	.	.	.	.	o	.	.	.	.	.	o	.	.
Bose and Suresh (2008)	o	.	.	o	.	.	.	.	o	.	.	.	.	.	o	.	.	o	.	.
Capobianco et al. (2009)	o	.	.	.	.	.	.	o	.	.	.	o	.	o	.	.	.	o	.	.
Cleary et al. (2008)	o	.	.	.	o	.	.	.	.	.	.	o	.	.	.	.	.	o	.	.
de Boer and van Vliet (2008)	o	.	.	.	.	.	.	o	.	.	o	.	.	o	.	.	.	.	.	.
De Lucia et al. (2004)	o	.	.	.	.	.	.	o	.	.	.	o	.	o	.	.	.	o	.	.
De Lucia et al. (2006)	o	.	.	.	.	.	.	o	.	.	.	o	.	o	.	.	.	o	.	.
De Lucia et al. (2007)	o	.	.	.	.	.	.	o	.	.	.	o	.	o	.	.	.	o	.	.
Dit et al. (2008)	o	.	.	.	.	.	.	.	.	.	o	.	.	.	.	o	.	o	o	.
Gall et al. (2008)	o	.	.	.	.	o	.	.	.	.	.	o	.	o	.	.	.	.	.	.
Gethers and Poshyvanyk (2010)	.	o	.	.	.	o	.	.	.	.	.	o	.	.	.	.	.	o	.	.
Grant et al. (2008)	.	.	o	.	o	.	.	.	.	.	.	o	.	.	.	.	.	.	.	.
Grant and Cordy (2009)	.	.	o	.	.	.	.	.	.	.	o	o	.	.	.	.	.	.	.	.
Grant and Cordy (2010)	.	o	.	.	.	.	.	.	.	.	o	o	.	.	.	.	.	o	.	.
Hayes et al. (2006)	o	.	.	.	.	.	.	o	.	.	.	.	.	o	.	.	.	o	.	.
Hindle et al. (2009)	.	o	.	.	.	.	o	.	.	.	.	.	.	.	o	.	.	.	.	.
Hindle et al. (2010)	.	o	.	.	.	.	o	.	.	.	.	.	.	.	o	.	.	.	.	.
Jiang et al. (2008)	o	.	.	.	.	.	.	o	.	.	.	o	.	o	.	.	.	o	.	.
Kagdi et al. (2010)	o	.	.	.	.	o	.	.	.	.	.	o	.	.	.	.	.	o	.	.
Kawaguchi et al. (2006)	o	.	.	.	.	.	.	.	.	o	.	o	.	.	.	.	.	.	.	.
Kuhn et al. (2005)	o	.	.	o	.	.	.	.	.	.	.	o	.	.	.	.	.	.	.	.
Kuhn et al. (2007)	o	.	.	o	.	.	.	.	.	.	.	o	.	.	.	.	.	.	.	.
Kuhn et al. (2008)	o	.	.	o	.	.	o	.	.	.	.	o	.	.	.	.	.	o	.	.
Kuhn et al. (2010)	o	.	.	o	.	.	o	.	.	.	.	o	.	.	.	.	.	.	.	.
Lin et al. (2006)	o	.	.	o	.	.	.	.	.	.	.	o	.	.	.	.	.	o	o	.
Linstead et al. (2007a)	.	o	.	.	o	o	.	.	.	o	.	o	.	.	.	.	.	.	.	.
Linstead et al. (2007b)	.	o	.	.	o	.	.	.	.	.	o	o	.	.	.	.	.	.	.	.
Linstead et al. (2008c)	.	o	.	o	o	o	.	.	.	o	o	o	.	.	.	.	.	.	.	.
Linstead et al. (2008a)	.	o	.	o	o	o	.	.	.	.	o	o	.	.	.	.	.	.	.	.
Linstead et al. (2008b)	.	o	.	.	o	.	o	.	.	.	.	o	.	.	.	.	.	.	.	.
Linstead and Baldi (2009)	.	o	.	.	.	o	.	.	.	.	.	.	.	.	.	o	.	.	.	.
Linstead et al. (2009)	.	o	.	.	o	.	o	.	.	.	o	o	.	.	.	.	.	.	.	.
Liu et al. (2009)	.	o	.	.	.	o	.	.	o	.	.	o	.	.	.	.	.	o	.	.

continued on next page

continued from previous page

	IR model			task								repository					evaluation			
	LSI	LDA	other	doc. clustering	concept loc.	metrics	trend/evolution	traceability	bug pred./debug	org./search coll.	other	source code	email	req./design	logs	bug reports	statistical	task specific	manual	user study
Lormans and Van Deursen (2006)	o	.	.	.	.	.	.	o	.	.	.	o	.	o	.	.	.	o	.	.
Lormans et al. (2006)	o	.	.	.	.	.	.	o	.	.	.	.	.	o	.	.	.	o	.	.
Lormans (2007)	o	.	.	.	.	.	o	o	.	.	.	.	.	o	.	.	.	o	.	.
Lukins et al. (2008)	.	o	.	.	.	.	.	o	.	.	.	o	.	.	.	o	.	o	.	.
Lukins et al. (2010)	.	o	.	.	.	.	.	o	.	.	.	o	.	.	.	o	.	o	.	.
Maletic and Valluri (1999)	o	.	.	o	.	.	.	.	.	.	.	o	.	.	.	.	.	.	.	.
Maletic and Marcus (2001)	o	.	.	o	.	.	.	.	.	.	.	o	.	.	.	.	.	.	o	.
Marcus and Maletic (2001)	o	.	.	.	.	.	.	.	.	.	o	o	.	.	.	.	.	.	o	.
Marcus and Maletic (2003)	o	.	.	.	.	.	.	o	.	.	.	o	.	o	.	.	.	o	.	.
Marcus et al. (2004)	o	.	.	.	o	.	.	.	.	.	.	o	.	.	.	.	.	o	o	.
Marcus (2004)	o	.	.	o	o	o	.	o	.	.	.	o	.	o	.	.	.	.	.	.
Marcus et al. (2005)	o	.	.	.	o	.	.	.	.	.	.	o	.	.	.	.	.	.	o	.
Marcus et al. (2008)	o	.	.	.	.	o	.	.	o	.	.	o	.	.	.	.	.	o	.	.
Maskeri et al. (2008)	.	o	.	.	o	.	.	.	.	.	.	o	.	.	.	.	o	.	.	.
McMillan et al. (2009)	o	.	.	.	.	.	.	o	.	.	.	o	.	o	.	.	.	o	o	o
Neuhaus and Zimmermann (2010)	.	o	.	.	.	.	o	.	.	.	.	.	.	.	.	o	.	.	.	.
Oliveto et al. (2010)	o	o	.	.	.	.	.	o	.	.	.	o	.	.	.	.	.	o	.	.
Ossher et al. (2009)	.	o	.	.	o	.	o	.	.	o	.	o	.	.	.	.	.	.	.	.
Poshyvanyk et al. (2006)	o	.	.	.	o	.	.	.	.	.	.	o	.	.	.	.	.	o	.	.
Poshyvanyk and Marcus (2007)	o	.	.	.	o	.	.	.	.	.	.	o	.	.	.	.	.	o	.	.
Poshyvanyk et al. (2007)	o	.	.	.	o	.	.	.	.	.	.	o	.	.	.	.	.	o	.	.
Poshyvanyk and Grechanik (2009)	o	.	.	.	o	.	.	.	.	.	o	o	.	.	.	.	.	.	.	.
Revelle and Poshyvanyk (2009)	o	.	.	.	o	.	.	.	.	.	.	o	.	.	.	.	.	o	.	.
Revelle et al. (2010)	o	.	.	.	o	.	.	.	.	.	.	o	.	.	.	.	.	o	.	.
Savage et al. (2010)	.	o	.	.	o	.	.	.	.	.	.	o	.	.	.	.	.	.	.	.
Thomas et al. (2010b)	.	o	.	.	.	.	o	.	.	.	.	o	.	.	.	.	.	.	o	.
Tian et al. (2009)	.	o	.	.	.	.	.	.	.	o	.	o	.	.	.	.	.	.	.	.
Ujhazi et al. (2010)	o	.	.	.	.	o	.	.	o	.	.	o	.	.	.	.	.	o	.	.
van der Spek et al. (2008)	o	.	.	.	o	.	.	.	.	.	.	o	.	.	.	.	.	.	o	.
Wu et al. (2008)	o	.	.	.	.	.	.	.	.	.	o	.	.	.	o	.	.	.	.	.
Zawawy et al. (2010)	o	.	.	.	.	.	.	.	o	.	.	.	.	.	o	.	.	o	.	.
Percentage 'o'	62	37	3	15	31	17	14	25	8	8	6	77	1	23	11	10	1	51	14	1

Table 2.4: Article characterization results of facets 1–4. The attributes are described in Table 2.3.

	preprocessing								tools			
	identifiers	comments	strings	granularity	tokenize	stem	stop	prune	tool	K value	justif. of K	iterations
Ahsan et al. (2009)	?	?	?	bug report	?	N	Y	Y	MATLAB	50-500	vary	-
Andrzejewski et al. (2007)	-	-	-	?	?	?	?	?	own	?	expert	2000
Antoniol et al. (2008)	?	?	?	method	Y	Y	Y	?	own	?	?	-
Asuncion et al. (2010)	?	?	?	?	?	Y	Y	?	own	?	?	?
Bajracharya and Lopes (2009)	-	-	-	query	Y	?	N	?	Dragon	50-500	vary	?
Bajracharya and Lopes (2010)	-	-	-	query	Y	?	N	?	Dragon	50-500	vary	?
Baldi et al. (2008)	Y	N	N	class	Y	?	Y	?	?	125	vary	?
Bavota et al. (2010)	?	?	?	method	?	?	?	?	?	?	?	-
Bose and Suresh (2008)	-	-	-	?	?	?	?	?	?	?	?	-
Capobianco et al. (2009)	?	?	?	class	?	?	Y	?	?	?	?	-
Cleary et al. (2008)	Y	Y	Y	?	Y	Y	Y	Y	?	300	?	-
de Boer and van Vliet (2008)	-	-	-	req.	N	?	Y	?	?	5	?	-
De Lucia et al. (2004)	Y	?	?	?	Y	?	Y	?	own	?	?	-
De Lucia et al. (2006)	?	?	?	?	?	N	?	?	own	?	?	-
De Lucia et al. (2007)	Y	N	N	?	Y	?	Y	Y	own	?	?	-
Dit et al. (2008)	?	?	?	bug report	Y	?	Y	?	?	300	?	-
Gall et al. (2008)	Y	Y	?	class	?	?	?	?	own	?	?	-
Gethers and Poshyvanyk (2010)	Y	Y	?	class	Y	?	?	?	lda-r	75, 125, 225	vary	?
Grant et al. (2008)	Y	Y	Y	method	?	?	?	Y	own	10	?	-
Grant and Cordy (2009)	Y	N	Y	method	?	?	?	Y	?	?	?	-
Grant and Cordy (2010)	Y	N	?	method	Y	?	?	?	GibbsLDA	50-300	vary	?
Hayes et al. (2006)	-	-	-	?	?	Y	Y	?	own	10-100	vary	-
Hindle et al. (2009)	-	-	-	commit msg	?	?	Y	Y	lda-c	20	vary	?
Hindle et al. (2010)	-	-	-	commit msg	?	?	?	?	?	?	?	?
Jiang et al. (2008)	?	?	?	?	?	?	?	?	own	?	?	-
Kagdi et al. (2010)	Y	Y	?	method	?	?	?	?	?	?	?	-
Kawaguchi et al. (2006)	Y	N	N	system	?	?	?	Y	own	?	?	-
Kuhn et al. (2005)	Y	Y	?	class/method	Y	Y	Y	?	?	200-500	?	-
Kuhn et al. (2007)	Y	Y	?	class	Y	Y	Y	?	own	15	?	-
Kuhn et al. (2008)	?	?	?	class	?	?	?	?	own	?	?	-
Kuhn et al. (2010)	Y	Y	Y	class	?	?	?	?	own	50	?	-
Lin et al. (2006)	N	Y	N	class	?	Y	Y	?	own	?	?	-
Linstead et al. (2007a)	?	?	?	class	Y	?	Y	?	own	100	vary	3000
Linstead et al. (2007b)	?	?	?	?	Y	?	Y	?	TMT	100	vary	3000
Linstead et al. (2008c)	?	?	?	?	Y	?	Y	?	?	100	vary	3000
Linstead et al. (2008a)	?	?	?	?	Y	?	Y	?	?	100	vary	3000
Linstead et al. (2008b)	Y	?	?	class	Y	?	Y	?	?	100	vary	?
Linstead and Baldi (2009)	?	?	?	bug report	Y	?	Y	?	?	100	vary	3500
Linstead et al. (2009)	?	?	?	?	?	?	?	?	?	?	?	?
Liu et al. (2009)	Y	Y	?	method	Y	?	Y	?	GibbsLDA	100	?	1000

continued on next page

continued from previous page

	preprocessing								tools			
	identifiers	comments	strings	granularity	tokenize	stem	stop	prune	tool	K value	justif. of K	iterations
Lormans and Van Deursen (2006)	?	?	?	?	?	Y	Y	?	TMG	?	?	-
Lormans et al. (2006)	-	-	-	?	?	?	?	?	own	?	?	-
Lormans (2007)	-	-	-	?	?	?	?	?	own	?	?	-
Lukins et al. (2008)	Y	Y	Y	method	Y	Y	Y	?	GibbsLDA	100	?	?
Lukins et al. (2010)	Y	Y	?	method	N	N	N	?	GibbsLDA	100	vary	?
Maletic and Valluri (1999)	Y	Y	Y	class/method	?	?	?	?	?	250	vary	-
Maletic and Marcus (2001)	Y	Y	Y	class/method	?	?	?	?	?	350	?	-
Marcus and Maletic (2001)	Y	Y	Y	class/method	?	?	?	?	own	350	?	-
Marcus and Maletic (2003)	Y	Y	Y	class	Y	?	?	?	?	?	?	-
Marcus et al. (2004)	Y	Y	N	?	Y	?	?	?	?	?	?	-
Marcus (2004)	Y	Y	?	class/method	Y	?	?	?	?	?	?	-
Marcus et al. (2005)	Y	Y	N	?	?	?	?	?	?	?	?	-
Marcus et al. (2008)	Y	Y	N	method	?	?	?	?	?	?	?	-
Maskeri et al. (2008)	Y	Y	?	class	Y	Y	Y	?	own	30	?	?
McMillan et al. (2009)	Y	?	?	method	Y	Y	Y	?	own	25-75	?	-
Neuhaus and Zimmermann (2010)	-	-	-	report	N	Y	Y	?	?	40	?	?
Oliveto et al. (2010)	?	?	?	?	?	Y	Y	Y	?	250	vary	?
Ossher et al. (2009)	?	?	?	?	?	?	?	?	?	?	?	?
Poshyvanyk et al. (2006)	Y	Y	?	method	?	?	?	?	?	?	?	-
Poshyvanyk and Marcus (2007)	Y	Y	N	method	Y	N	N	N	?	?	?	-
Poshyvanyk et al. (2007)	Y	Y	N	method	Y	N	N	N	?	500	?	-
Poshyvanyk and Grechanik (2009)	?	?	?	?	?	?	?	?	?	?	?	-
Revelle and Poshyvanyk (2009)	Y	?	?	method	?	?	?	?	?	?	?	-
Revelle et al. (2010)	Y	Y	Y	method	Y	Y	?	?	?	?	?	-
Savage et al. (2010)	Y	Y	?	class	Y	Y	Y	?	JGibbLDA	input	?	input
Thomas et al. (2010b)	Y	Y	?	class	Y	Y	Y	?	MALLET	45	previous	?
Tian et al. (2009)	Y	Y	?	system	Y	?	Y	?	GibbsLDA	40	vary	?
Ujhazi et al. (2010)	Y	Y	N	method	Y	Y	Y	?	?	?	?	-
van der Spek et al. (2008)	Y	Y	N	method	Y	N	Y	?	SVDLIBC	input	vary	-
Wu et al. (2008)	-	-	-	log	Y	Y	Y	?	JAMA	?	?	-
Zawawy et al. (2010)	-	-	-	log	?	Y	Y	?	?	?	?	-
Percentage ‘Y’	54	42	14	-	48	27	49	11	-	-	-	-
Percentage ‘N’	1	7	15	-	4	8	7	3	-	-	-	-
Percentage ‘?’	27	32	52	30	48	65	44	86	48	49	70	25

Table 2.5: Article characterization results of facets 5 and 6. The attributes are described in Table 2.3. A ‘Y’ means the article stated that it included this attribute or performed this step; a ‘N’ means the article stated that it did not include this attribute or perform this task; a ‘?’ means the article did not state either way; and a ‘-’ means this attribute is not applicable to this article.

Part II

New Applications of IR Models in Software Engineering

In Section 2.4, we found that most research to date uses IR models to perform the software engineering tasks of either concept location or traceability linking. In this part of the thesis, we present additional software engineering tasks that can be solved using basic IR models: test case prioritization, and measuring the information flow between mailing lists and source code. Our use of IR models in solving these tasks is novel, and shows that research in software engineering can move beyond the limited applications of IR models to achieve new and exciting results.

– **Chapter 3: Prioritizing test cases using topic models**. Typically, test cases are prioritized by maximizing the amount of source code that is covered, known as the code coverage of the test cases. Such prioritization techniques ignore the linguistic data present in test cases, i.e., the identifier names, comments, and string literals that help to determine the functionality of the test cases. In this chapter, we prioritize test cases by applying LDA, a statistical IR model, to the linguistic data of the test cases. We use the resultant topics to determine the similarity between test cases, giving highest priority to those test cases that are most dissimilar. Through case studies on two open source systems, Apache Ant and Apache Derby, we find that our technique outperforms traditional code coverage techniques by up to 30% in the average percentage of faults detected.

– **Chapter 4: Measuring the interaction between mailing lists and source code**. Understanding how distributed development teams communicate over email, and how email discussions affect source code implementation, can help managers guide documentation and training efforts and monitor project status. In addition, developers need to recover past email discussions pertaining to a given source code entity. In this chapter, we apply LDA jointly to the source code and mailing list of a system to uncover the shared topics and their behavior over time. We find that the discovered topics are sometimes more active in either the source code or mailing list, are sometimes equally active in both, and are sometimes inactive altogether. Through case studies on two open source systems, PostgreSQL and Apache HTTP Server, we find that, for example, topics are first heavily discussed in the mailing list for a short period of time before being implemented in the source code, confirming previous intuition of distributed software development practices. Our technique can also be used to determine which topics are being implemented in a certain time period, or which topics were being discussed on the mailing list.

CHAPTER 3

Prioritizing Test Cases Using Topic Models

Software development teams use test suites to test changes to their source code. In many situations, the test suites are so large that executing every test for every source code change is infeasible, due to time and resource constraints. Development teams need to prioritize their test suite so that as many distinct faults as possible are detected early in the execution of the test suite. We consider the task of static black-box test case prioritization (TCP), where test suites are prioritized without the availability of the source code of the system under test (SUT). We propose a new static black-box TCP technique that represents test cases using a previously unused data source in the test suite: the linguistic data of the test cases, i.e., their identifier names, comments, and string literals. Our technique applies topic models to the linguistic data to approximate the functionality of each test case, allowing our technique to give high priority to test cases that test different functionalities of the SUT. We compare our proposed technique with existing static black-box TCP techniques in a case study of multiple real-world open source systems: several versions of Apache Ant and Apache Derby. We find that our static black-box TCP technique outperforms existing static black-box TCP techniques, and has comparable or better performance than two existing execution-based TCP techniques.

Publications based on this chapter: Thomas et al. (2012c)

3.1 Motivation

SOFTWARE DEVELOPMENT TEAMS typically create large test suites to test their source code (Ali et al., 2009). These test suites can grow so large that it is cost prohibitive to execute every test case for every new source code change (Rothermel et al., 2001). For example, at Google, developers make more than 20 source code changes per minute, resulting in 100 million test cases executed per day (Kumar, 2010). In these situations, developers need to *prioritize* the test suite so that test cases that are more likely to detect undetected faults are executed first. To address this challenge, researchers have proposed many automated test case prioritization (TCP) techniques (Elbaum et al., 2002; Rothermel et al., 2001). Most of these TCP techniques use the *execution* information of the system under test (SUT): the dynamic run-time behavior, such as statement coverage, of each test case (Chen et al., 2011; Simao et al., 2006). While execution information is a rich information source for the TCP technique, execution information may be unavailable for several reasons (Zhang et al., 2009):

– Collecting execution information can be cost prohibitive, both in terms of time and resources.

– For large systems, execution information will be quite large, making storage and maintenance costly.

– Execution information must be continuously updated as source code and test cases evolve.

To address situations in which the execution information is not available, researchers have proposed TCP techniques based on *specification models* of the tests (Hemmati et al., 2012; Korel et al., 2007). These models describe the expected behavior of the SUT and test suite. However, specification models are also sometimes not available, for similar reasons: the models may be cost prohibitive to manually generate, or the maintenance of the models as source code and test cases evolve may be cost prohibitive.

To address situations when neither the execution information nor specification models are available, researchers have recently developed *static* TCP techniques. In particular, Zhang et al. (2009) and Mei et al. (2011) propose techniques based on the static call graph of the test cases. Additionally, Ledru et al. (2011) treat each test case as a string of characters, and prioritize test cases by using a simple string edit distance to determine the similarity between test cases. In these techniques, the goal is to give high priority to test cases that are highly dissimilar (e.g., because they invoke different methods, or have high string distances), thereby maximizing test case diversity and casting a wide net for detecting unique faults (Hemmati et al., 2010b, 2011). While static TCP techniques do not have as much information to work with as those based on execution information or specification models, static techniques are less expensive and are lighter weight, making them applicable in many practical situations.

However, existing static TCP techniques make little or no use of an important data source embedded within test cases: their *linguistic data*, i.e., the identifier names, comments, and string literals that help to determine the functionality of the test cases (Kuhn et al., 2007). In this chapter, we propose a new static TCP technique that uses the unstructured linguistic data of test cases to help differentiate their functionality. Our technique uses LDA to create topics from the linguistic data, and prioritizes test cases that contain different topics. The main advantage of our technique, compared to existing black-box static TCP techniques, is that topics abstractly represent test cases' functionality, which is robust to trivial differences in the test's source code and can capture more information than call graphs alone.

3.2 Background

Test case prioritization (TCP) is the task of ordering the test cases within the test suite of a system under test (SUT), with the goal of maximizing some criteria, such as the fault detection rate of the test cases (Wong et al., 1997). Should the execution of the test suite be interrupted or stopped for any reason, the more important test cases (with respect to the criteria) have been executed first. More formally, Rothermel et al. (2001) define the TCP task as follows.

Definition 1 (Test case prioritization) Given: T, a test suite; PT, the set of permutations of T; and f, a performance function from PT to the real numbers. Find: $T' \in PT$ s.t. $(\forall T'' \in PT)(T'' \neq T')[f(T') \geq f(T'')]$.

In this definition, PT is the set of all possible prioritizations of T and f is any function that determines the performance of a given prioritization. The definition of performance can vary, as developers will have different goals at different times (Rothermel et al., 2001). Developers may first wish to find as many faults as possible; they may later wish to achieve maximal code coverage. In these scenarios, the task definition of TCP is the same, but the performance function f being optimized changes. Researchers have proposed and evaluated many techniques to solve the TCP task, based on a range of data sources and prioritization algorithms; Yoo and Harman have conducted a thorough survey of such techniques (Yoo and Harman, 2010). In general, each TCP technique tries to achieve optimal prioritization based on the following steps (Hemmati et al., 2010a).

1. Encode each test case by what it *covers*, i.e., what it does, based on the elements in the available dataset. For example, the elements may be the program statements of the SUT that the test case covers.

2. Prioritize test cases using a distance maximization algorithm, based on maximizing either coverage or diversity.

   (a) *Maximize coverage.* Prioritize test cases so that the maximum number of elements are covered. The intuition behind this strategy is that by maximizing element coverage, the technique tests as much of the SUT as possible and increasing its chances of fault detection.

   (b) *Maximize diversity.* First, determine the *similarity* (equivalently *distance*) between test cases, for some definition of similarity (e.g., intersection of statements covered). Then, prioritize test cases by maximizing their dissimilarity. The intuition for this strategy is that pairs of test cases that are similar will likely detect the same faults, and therefore only one needs to be executed.

   Researchers have proposed many distance maximization algorithms, including greedy algorithms, clustering, and genetic algorithms (Hemmati et al., 2010a).

In this chapter, we categorize TCP techniques primarily based on step 1 above, i.e., how the technique encodes the test cases into elements, resulting in the following five categories.

Definition 2 (White-box execution-based prioritization) Prioritization based on the dynamic execution behavior of the test cases, with access to the source code of the SUT. Test cases are encoded by which source code elements they execute. Requires the source code of the SUT and test cases to be instrumented, compiled, and executed.

Definition 3 (Black-box execution-based prioritization) Prioritization based on the dynamic execution behavior of the test cases, using the execution logs of the source code. Test cases are encoded using the contents of their execution logs. Does not require the actual source code of the SUT, but instead assumes that execution log files are available for each test case.

Definition 4 (Grey-box Model-based prioritization) Prioritization based on specification models (e.g., state diagrams) of the SUT and test cases. Test cases are encoded by which paths they take on the specification model. Does not require execution information. Requires specification models to be created and maintained.

Definition 5 (White-box static prioritization) Prioritization based on the source code of the SUT and test cases. Test cases are encoded based on some aspect of the static snapshot of the source code of the SUT and test cases. Does not require execution information or specification models. Requires access to the source code of the SUT and test cases.

Definition 6 (Black-box static prioritization) Prioritization based on the source code of the test cases. Test cases are encoded based on some aspect of the source code of the test cases. Does not require execution information, specification models, or source code of the SUT. Only requires source code of the test cases.

Table 3.1 categorizes related work, which we now summarize.

3.2.1 White-box Execution-based Prioritization

Most existing TCP techniques are white-box execution-based (also called *dynamic coverage-based*): they use the dynamic execution information of the test cases to prioritize them (Yoo and Harman, 2010). Many of these technique aim to maximize source code coverage. For

Table 3.1: A two dimensional classification of TCP techniques, based on the data available (execution information, specification models, or static source code) and maximization strategy (coverage-based or diversity-based).

Max. strategy	Execution-based		Model-based	Static	
	White-box	Black-box	Grey-box	White-box	Black-box
Coverage	e.g., Elbaum et al. (2002); Feldt et al. (2008); Jones and Harrold (2003)	Sampath et al. (2008)	Korel et al. (2007)	Zhang et al. (2009)	this thesis
Diversity	Ramanathan et al. (2008); Simao et al. (2006); Yoo et al. (2009)	–	Hemmati et al. (2010a,b, 2012)	–	Ledru et al. (2009, 2011); Mei et al. (2011), this thesis

example, Wong et al. (1997) present a coverage-based technique to prioritize test cases in the context of regression testing, taking into account the changes in the source code between versions and giving high priority to those test cases that likely test the changed portions of the source code. Rothermel et al. (2001) present a family of coverage-based techniques, all based on statement-level execution information, and define the Average Percentage of Fault-Detection (APFD) metric that is widely used today for evaluating the effectiveness of TCP techniques. Many subsequent studies also use statement-level execution information in the source code as the basis for prioritization cases (Feldt et al., 2008; Jiang et al., 2009; Jones and Harrold, 2003; Leon and Podgurski, 2003; Masri et al., 2007; McMaster and Memon, 2006), differing mainly in coverage metric definition or maximization algorithm.

Other work prioritizes test cases by maximizing test case diversity. For example, Yoo et al. (2009) use a clustering algorithm to differentiate test case's execution profiles. Simao et al. (2006) build a feature vector for each test case (which can include any desired aspect of the test case), and then use a neural network to identify test cases with the most dissimilar features. Ramanathan et al. (2008) build a graph of test cases similarity and experiment with different graph orders for test case prioritization. In building the graph, the authors consider standard execution information as well as additional heuristics, such as memory operations.

3.2.2 Black-box Execution-based Prioritization

It is possible for execution-based TCP techniques to not require access to the source code of the SUT. For example, Sampath et al. (2008) focus on prioritizing test cases for web systems, where logs of user behavior (i.e., data requests sent to the web server) are available to the prioritization algorithm. The user data in the logs contains the dynamic execution scenarios for the user, which captures the execution information of the SUT without actually requiring access to its source code. Using the data in the logs, Sampath et al. consider a number of strategies to prioritize test cases based, most of which try to maximize the method coverage of the SUT.

To the best of our knowledge, there have been no proposed black-box execution-based techniques based on maximizing the diversity of test cases. However, this could be achieved by, for example, clustering the elements in the user data logs of a web system, and then selecting test cases from different clusters.

3.2.3 Grey-box Model-based Prioritization

For situations in which the execution behavior of the SUT is not available, due to time, budget, or resource constraints, researchers have proposed techniques based on the specification models of the source code (Hemmati et al., 2010a,b, 2012; Korel et al., 2007). Specification models represent the expected behavior of the source code (via e.g., a UML state diagram) and test cases (e.g., paths on the state diagram). Specification models relate each test case to an execution path in the state model. By definition, all model-based TCP techniques are gray-box, in that they require knowledge of the internal data structures and architecture of the SUT, but do not explicitly require the source code of the SUT during the prioritization of the test suite.

For example, Korel et al. (2007) compute the difference between two versions of the specification model of a SUT, and give high priority to those test cases whose models intersect with the difference. In effect, the authors are maximizing the coverage of the model in the new version. The authors also propose a set of heuristics to enhance this basic technique.

Hemmati et al. (2010a,b) explore a number of diversity-based algorithms which operate on the similarity between test cases' paths in the state model, and give high priority to those test cases whose paths are most dissimilar. The authors consider a wide range of maximization algorithms including genetic algorithms, clustering algorithms, and greedy algorithms.

In these model-based TCP techniques, the specification models of the SUT and test cases are required. Developers must create and maintain these specification models of the source code and test cases, which may not always be feasible due to the time and labor required.

3.2.4 White-box Static Prioritization

If both the execution behavior and specification models of the SUT are unavailable, then the TCP technique must rely only on static snapshots of the source code of the SUT and/or the test cases themselves (Ledru et al., 2011). We call this *static* TCP, which is our focus in this chapter. Among static techniques, some require access to the source code of the SUT (white-box) and others do not (black-box).

An example of a white-box static TCP technique is the *call graph-based* technique, proposed by Zhang et al. (2009) and later extended by Mei et al. (2011). The call graph-based technique uses the static call graph of test cases to approximate their dynamic source code coverage. After statically extracting the call graph of each test case, the authors define a Testing Ability (TA) metric that measures the number of source code methods that a test invokes, taking into account the methods invoked by the already-prioritized test cases. (In their work, Zhang et al. refer to this technique as "JuptaA", while Mei et al. refer to it as

“JUPTA”.) The authors define the TA metric by

$$\text{TA}(T_i, PS) = \sum_{m \in \text{Rel}(T_i, PS)} \text{FCI}(m)$$

where T_i is a test case under consideration, PS is the set of already-prioritized test cases, Rel(T_i, PS) is a function that returns the set of methods relevant to T_i but not covered by the already-prioritized test cases PS, and the function FCI(m) represents the probability that method m contains a fault. Zhang et al. and Mei et al. treat all methods equally, and therefore FCI(m)=1 for all m.

The call graph-based technique is coverage-based and prioritizes test cases using a greedy algorithm that iteratively computes the TA metric on each unprioritized test case. Initially, the test case with the highest TA metric is added to PS. Then, at each iteration, the greedy algorithm adds to PS the test case with the highest TA value. In this way, the greedy algorithm attempts to maximize the number of source code methods that are covered by the prioritized test suite.

In the above white-box static TCP technique, the maximization strategy is based on maximizing code coverage. To the best of our knowledge, no white-box static TCP technique has been proposed that maximizes based on test case diversity.

3.2.5 Black-box Static Prioritization

An example of a black-box static TCP technique is the *string-based* technique, proposed by Ledru et al. (2009, 2011). The string-based technique treats test cases as single, continuous strings of text. The technique uses common string distance metrics, such as the Levenshtein edit distance, between test cases to determine their similarity. The intuition is that if two test cases are textually similar, they will likely test the same portion of source code and therefore detect the same faults.

To measure the distance between two strings (i.e., test cases), Ledru et al. consider several distance metrics, including Euclidean, Manhattan, Levenshtein, and Hamming. The authors find that the Manhattan distance has the best average performance for fault detection.

To maximize diversity between strings, Ledru et al. use a greedy algorithm, similar to that of the call graph-based technique, that always prioritizes the test case which is furthest from the set of already-prioritized test cases. To do so, Ledru et al. define a distance measure between a single test case and a set of test cases. For a test case T_i, the set of already-prioritized test cases PS, and a distance function $d(T_i, T_j)$ which returns the distance between T_i and T_j, the authors define the distance between T_i and PS to be:

$$\text{AllDistance}(T_i, PS, d) = \min\{d(T_i, T_j) \mid T_j \in PS, j \neq i\}. \tag{3.1}$$

The authors choose the min operator because it assigns high distance values to test cases which are most different from all other test cases.

Similar to the greedy algorithm in the call graph-based technique, the greedy algorithm in the string-based technique iteratively computes the AllDistance metric for each unprioritized test case, giving high priority to the test with the highest AllDistance value at each iteration.

Despite its simplicity, this basic technique shows encouraging results that inspire us to consider the textual information to help differentiate test cases.

In this chapter, during the evaluation of our proposed technique (which we present next), we also implement a black-box static TCP technique that maximizes test cases by maximizing coverage, rather than diversity. Our implementation is a modification of the white-box call graph-based technique, and is described further in Section 3.4.1.

3.3 Proposal

We propose a new black-box static TCP technique that takes advantage of an additional data source in the test scripts: the developer knowledge contained in the identifier names, comments, and string literals (Kuhn et al., 2007), collectively called *linguistic data*. We

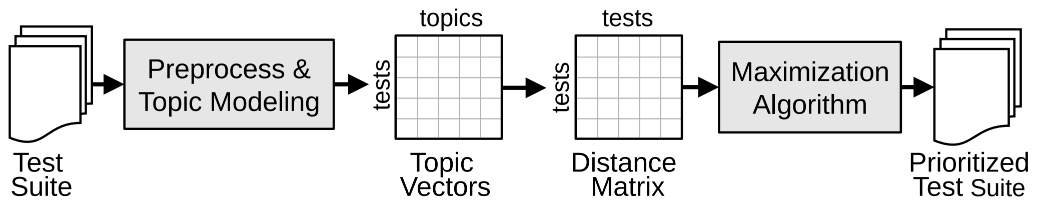


Figure 3.1: Overview of our proposed topic-based TCP technique.

apply a topic modeling technique (Section 2.2) to the test case linguistic data to abstract test cases into topics, which we use to compute the similarity between pairs of test cases. The motivation for using topic modeling stems from recent research that found that the topics discovered from linguistic data are good approximations of the underlying *business concerns*, or functionality, in the source code (Kuhn et al., 2007; Maskeri et al., 2008). We hypothesize that when two test cases contain the same topics, the test cases are similar in functionality and will detect the same faults.

Our proposed TCP technique performs the following steps (see Figure 3.1).

1. Preprocess the test suite (i.e., set of individual test cases) to extract the linguistic data (i.e., identifier names, comments, and string literals) for each test case.

2. Apply topic modeling to the preprocessed test suite, resulting in a vector of topic memberships for each original test case (i.e., a vector describing the probability that the test case is assigned to each topic).

3. Define the distance between pairs of test cases based on their topic membership vectors. Many standard distance metrics are applicable (Blei and Lafferty, 2009), including Manhattan distance and Kullback-Leibler (Kullback and Leibler, 1951) distance.

4. Finally, prioritize test cases by maximizing their average distances to already-prioritized test cases, using any of several distance maximization algorithms (e.g., greedy, clustering, or genetic algorithms).

The key benefit of using topic modeling is that it uses a test script's linguistic data to provide an approximation of the business concerns of the test script. Business concerns often cannot be deduced from a test script's call graph alone. For example, consider how the call graph-based technique treats the following two example test scripts.

```
/* Read in the default color data for the menu bar, pull-down menus,
 * mouse hovers, and button down indicators. */
String defaultColors = readFileContents("default_color_options.dat");
...
```

```
/* Read in the terrain map data for Tucson, AZ. The file contains elevation
 * points of the terrain in a 10x10 meter grid. */
String terrain = readFileContents("tucson_map.dat");
...
```

Despite their obvious differences in functionality (which can be seen from the comments and identifier names), the call graph-based technique will consider these two test snippets to be identical, since they both make a single method call to the same method (i.e., `readFileContents()`). In this case, the linguistic data serves to help identify that the test snippets perform different functions, even if their call graphs are similar.

Additionally, while the string-based technique does use the comments and identifier names during the calculation of string distances, it does so in an overly rigid manner. Consider the following three example test scripts, which illustrate the effect of identifier names on the similarity measure computed by the string-based technique:

```
/* Test 1 */ int printerID=getPrinter();
/* Test 2 */ int printerID; int x; x=getPrinter(); printerID=x;
/* Test 3 */ int compiler=getCompiler();
```

Here, tests 1 and 2 use similar identifier names and are identical in terms of execution semantics, but they are considered relatively dissimilar due to the number of edits required to change one string into the other, compared to the strings' lengths. On the other hand, tests 1 and 3 have a higher string similarity, even though they have no common identifier names nor functional similarities.

We compare our proposed technique to two existing black-box static TCP techniques in a detailed case study on several real-world systems. We find that our topic-based technique increases the average fault detection rate over existing techniques by as much 31%, showing improved performance across all studied systems.

3.4 Case Studies

We are interested in the following research question.

Is our proposed topic-based TCP technique more effective than existing black-box static TCP techniques?

In this research question, we relate the term *effectiveness* to the standard *APFD* evaluation measure (Rothermel et al., 2001). To answer this question, we perform a detailed case study on multiple real-world software systems. We now describe the details and design decisions of our study.

3.4.1 Techniques Under Test

We implement and test the following static black-box TCP techniques.

- **Black-box call graph-based**. To provide a fair comparison with our proposed topic-based technique, we have implemented a black-box version of the call graph-based technique. Our implementation uses TXL (Cordy, 2006) to extract the static call graph of the test cases without relying on the source code. Essentially, our implementation extracts a list of method names invoked by each test case, but does not continue down the call graph further into the source code, as we assume the source code is unavailable. We then prioritize the test cases using the same greedy algorithm used by the white-box call graph-based technique described in Section 3.2.4.
- **String-based**. We have implemented the black-box static technique proposed by Ledru et al. (2011), which uses the string edit distance between test cases to maximize test case diversity.
- **Topic-based**. This technique, proposed in Section 3.3, uses an abstracted representation of test cases, based on topic modeling, to maximize test case diversity.

As a baseline technique, we implement and test the random technique.

- **Random**. This technique randomizes the order of the test cases.

3.4.2 Systems Under Test

We obtain data from the Software-artifact Infrastructure Repository (SIR), a public repository containing fault-injected software systems of various sizes, domains, and implementation languages (Do et al., 2005). The SIR is a superset of the popular Siemens test suite. The dataset for each system includes the fault-injected source code, test cases, and fault matrices (i.e., a description of which faults are detected by which test cases) for several versions of the system. The faults were manually injected by humans in an attempt to provide realistic and representative errors made by programmers. We choose this repository because it contains many real-world, open-source systems and many previous TCP studies have used it, providing an opportunity for fair comparison.

Table 3.2 summarizes the five systems under test that we use from SIR. The *failure rate* of each system indicates the portion of test cases that detect at least one fault. We selected these five systems based on the following criteria.

– The test cases are written in a high-level programming language (e.g., unit tests).

– The fault matrix contains at least five faults.

– The fault matrix contains at least ten test cases.

The first criterion is required since we use static TCP techniques (which are based on analysis of the source code of the test cases); see Section 3.5.2 for a discussion. The last two criteria ensure that the datasets are large enough to draw reasonable conclusions from our results.

Both Ant and Derby are maintained by the Apache Foundation (Apache Foundation, 2012c), and both are written in the Java programming language. Ant is a command line tool that helps developers manage the build system of their software (Apache Foundation, 2012b). Derby is a lightweigth relational database management system (Apache Foundation, 2012d). We consider the versions of Derby as different systems under test because the injected faults are not propagated between versions. Each version is injected with new faults in new locations in the source code. Additionally, test cases are added and removed between versions. (An exception is Derby v4, which is very similar to Derby v3, so we only

Table 3.2: Systems under test (from the Software-artifact Infrastructure Repository (Do et al., 2005)).

System	Version	Release date	SLOC (K)	No. of faults	No. of tests	Failure rate
Ant v7	1.5.3	4/2003	90	6	105	17%
Derby v1	10.1.2.1	11/2005	420	7	98	21%
Derby v2	10.1.3.1	6/2006	416	9	106	34%
Derby v3	10.2.1.6	10/2006	517	16	120	22%
Derby v5	10.3.1.4	8/2007	571	26	53	64%

include Derby v3 in our case study.) Thus, for the purposes of our case study, we treat each version of Derby as an independent system.

These datasets provide us with test oracles on which to evaluate the techniques under test. Specifically, the datasets include a *fault matrix* which indicates which faults each test is able to detect. As the fault matrix is unknown in practice, we only use the fault matrix to evaluate the relative performance of each TCP technique; the fault matrices are not accessible to any of the TCP techniques during the prioritization process.

3.4.3 Data Preprocessing

We preprocess the text of the test cases for the topic-based techniques in the standard way when applying topic modeling techniques to source code (Section 2.2). Specifically, we split identifier names, remove stop words, and stem those that remain. We did not preprocess the test cases in the string-based technique, to be consistent with the technique of Ledru et al. (2011).

3.4.4 Distance Metrics and Maximization Algorithms

For our case study, we fix the distance metric to be used by both string-based and topic-based techniques. We use the Manhattan distance metric, because it is applicable to both TCP techniques and Ledru et al. found it to be optimal for string-based TCP (Ledru et al., 2011).

For distance maximization, we implement the greedy algorithm used by previous research (Elbaum et al., 2002). Namely, our implementation first uses the AllDistance metric (Equation 3.1) to determine the test case that is most dissimilar from the set of all test cases, and adds it to the (initially empty) set of already-prioritized test cases, PS. Then, our implementation iteratively finds the test case that is most dissimilar (again, based on the AllDistance metric) to all the test cases in PS, and adds it to PS. If two or more test cases have the same dissimilar value, the tie is broken randomly. The implementation continues in this way until all test cases have been prioritized.

3.4.5 Topic Modeling Technique

Our proposed topic-based TCP technique can use any of several underlying topic modeling techniques. In this case study, we use the well-known LDA model (Blei et al., 2003).

We use the `lda` package (Chang, 2012) of the R programming environment (Ihaka and Gentleman, 1996) as our LDA implementation. As LDA is a generative statistical model that uses machine learning algorithm to approximate the ideal set of topics, some randomness is involved. Given a different random number seed, LDA may produce a slightly different set of topics (Blei et al., 2003). The `lda` package uses collapsed Gibbs sampling (Porteous et al., 2008) as the machine learning algorithm to approximate topics.

To use LDA, we must specify four parameters: the number of topics, K; the document-topic smoothing parameter, α; the topic-word smoothing parameter, β; and the number of Gibbs sampling iterations to execute, II. For a given corpus, there is no provably optimal choice for K (Wallach et al., 2009). The choice is a trade-off between coarser-grained topics (smaller K) and finer-grained topics (larger K). Setting K to extremely small values results in topics that approximate multiple concerns (imagine only a single topic, which will contain all of the concerns in the corpus!), while setting K to extremely large values results in topics that are too fine and nuanced to be meaningful. Common values in the software engineering literature range from 5 to 500 topics, depending on the number of documents, granularity desired, and task to perform (Griffiths et al., 2007). We seek topics of medium granularity, and thus use $K = N/2.5$ topics per system (rounded to the nearest integer), where N is the number of documents in the SUT. Thus, we use K = 42, 39, 42, 48, and 21 for the five systems listed in Table 3.2, respectively. For α, β, and the number of iterations II, we use the defaults in the `lda` package (which are α=0.1, β=0.1, and II=200) across all five systems, as these values are not dependent on the number of documents in the SUT. (We investigate parameter sensitivity in Section 3.5.2.)

3.4.6 Random Samples of Performance

To mitigate the effect of randomization and to capture the variation in the performance of each TCP technique, we run each technique several times with different random number seeds. Doing so allows us to quantify the expected behavior of each TCP technique, as well as calculate statistics on the variability of each technique.

In particular, for the topic-based technique, we first run the topic modeling technique 30 times, each with the same parameters (i.e., K, α, β, and number of iterations) but a different initial random seed. This results in a set of 30 topic models (i.e., the topics themselves as well as the topic membership vectors for each test case), each showing a slight variation. We compute the distance matrices for each of the 30 topic models. We then execute the greedy maximization algorithm 30 times for each distance matrix, which causes ties to broken in different ways.

For string-based techniques, there is no randomization when computing the distance matrix, because only deterministic processes are being executed; the process of computing the distance between the raw strings is deterministic. Instead, we apply the greedy algorithm 900 times, randomly breaking ties in different ways, in order to collect an equal number of samples as the topic-based technique. Similarly, for the call-graph technique, we apply the greedy algorithm 900 times, randomly breaking ties in different ways. For the random prioritization technique, we compute 900 random prioritizations.

3.4.7 Analysis and Evaluation Metrics

We use a standard metric to evaluate the performance of the different techniques: *average percentage of fault-detection* (APFD) (Rothermel et al., 2001). APFD captures the average of the percentage of faults detected by a prioritized test suite. APFD is given by

$$APFD = 100 * \left(1 - \frac{TF_1 + TF_2 + \cdots + TF_m}{nm} + \frac{1}{2n}\right), \tag{3.2}$$

where n denotes the number of test cases, m is the number of faults, and TF_i is the number of tests which must be executed before fault i is detected. As a TCP technique's effectiveness increases (i.e., more faults are detected with fewer test cases), the APFD metric approaches 100. (We note that APFD is bounded in the range of $[100*(1-(1/2n)), 100*(1-(3/(2n))]$.)

To compare and contrast the APFD values of the different TCP techniques, we use three metrics. The first, %Δ, is the percent difference between the mean APFD results of two techniques:

$$\%\Delta(\mu_1, \mu_2) = \frac{\mu_2 - \mu_1}{\mu_1}, \tag{3.3}$$

where μ_1 and μ_2 are the mean APFD results of two TCP techniques. Second, we use the Mann-Whitney U statistical test (also known as the Mann-Whitney-Wilcoxon test) to determine if the difference between the APFD results are statistically significant. The Mann-Whitney U test has the advantage that the sample populations need not be normally distributed (non-parametric). If the p-value of the test is below a significance threshold, say 0.01, then the difference between the two techniques is considered statistically significant.

While the Mann-Whitney U test tells us whether two techniques are different, it does not tell us how much one technique outperforms another, i.e., the *effect size*. To quantify the effect size between the techniques, our third metric is the Vargha-Delaney A measure (Vargha and Delaney, 2000), which is also robust to the shape of the distributions under comparison (Arcuri and Briand, 2011). The A measure indicates the probability that one technique will achieve better performance (i.e., higher APFD) than another technique. When the A

measure is 0.5, the two techniques are equal. When the A measure is above or below 0.5, one of the techniques outperforms the other.

3.5 Results and Discussion

We wish to evaluate the effectiveness of topic-based TCP with respect to other state-of-the-art static techniques. To do so, we compare the APFD results (Equation 3.2) of the topic-based technique (TOPIC) against three baselines: 1) the random TCP technique (RANDOM); 2) the call graph-based technique (CALLG); and 3) the string-based technique (STRG).

3.5.1 Results

Table 3.3 and Figure 3.2 present and compare the APFD results for the five systems under test. Figure 3.2 shows boxplots for each technique, which summarize the distribution of the 900 random samples that we collected for each technique. Table 3.3 shows the mean APFD values along with comparisons between techniques using the Mann-Whitney U test (to determine if the techniques are statistically different) and the Vargha-Delaney A measure (to quantify how different the techniques are). We use the figure and table to make the following observations.

First, we find in Table 3.3 that the differences between techniques are almost always statistically significant according to the Mann-Whitney U test (i.e., the resulting p-values are $<$0.01), with one exception (Ant v7: TOPIC compared to CALLG). This indicates that the techniques capture different aspects of the tests and that the techniques result in significantly different prioritizations. In the exceptional case, TOPIC and CALLG exhibit equivalent effectiveness, as the p-value is $\geq$0.01.

Second, TOPIC always outperforms RNDM, as determined by higher mean APFD values and a effect sizes $\geq$ 0.5. In the best case, TOPIC outperforms RNDM by 20%; on average TOPIC outperforms RNDM by 7.2%.

Third, TOPIC outperforms CALLG in all but one system under test. In Ant v7, the

Table 3.3: Mean APFD results (μ) and comparisons for each technique: random (RNDM), call-graph (CALLG), string (STRG), and topic (TOPIC). Each technique is compared to TOPIC using percent change in mean (%Δ), the p-value of the Mann-Whitney U test (p-val), and the Vargha-Delaney A measure (A); see Section 3.4.7 for definitions of these metrics. We bold the best technique(s) for each SUT.

	RNDM	CALLG	STRG	TOPIC									
					vs. RNDM			vs. CALLG			vs. STRG		
	μ	μ	μ	μ	%Δ	p-val	A	%Δ	p-val	A	%Δ	p-val	A
Ant v7	70.6	**85.5**	77.2	**84.8**	20.1	<0.001	0.94	-0.9	0.515	0.49	9.8	<0.001	0.94
Derby v1	90.7	91.8	90.4	**93.7**	3.4	<0.001	0.64	2.1	<0.001	0.77	3.7	<0.001	0.86
Derby v2	91.5	87.6	90.3	**94.6**	3.3	<0.001	0.69	8.0	<0.001	0.97	4.8	<0.001	0.91
Derby v3	92.7	93.4	93.5	**94.0**	1.3	<0.001	0.55	0.6	<0.001	0.67	0.5	<0.001	0.67
Derby v5	89.1	73.8	82.1	**96.4**	8.1	<0.001	0.86	30.5	<0.001	1.00	17.4	<0.001	1.00

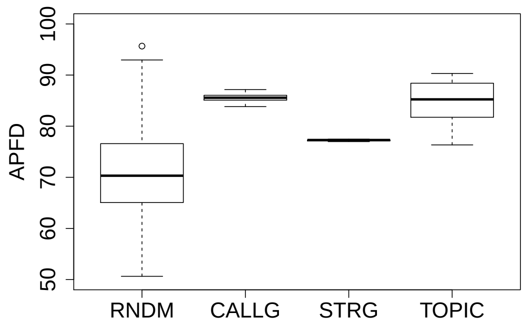
performance of TOPIC is not statistically different from the performance of CALLG: the p-value is $\geq$0.01 and the effect size is very close to 0.5. However, in each of the other four systems under test, TOPIC outperforms CALLG by up to 31% and by 10.3% on average.

Finally, TOPIC always outperforms STRG. In the best case, TOPIC outperforms STRG by 17%; TOPIC outperforms STRG by 7.2% on average.

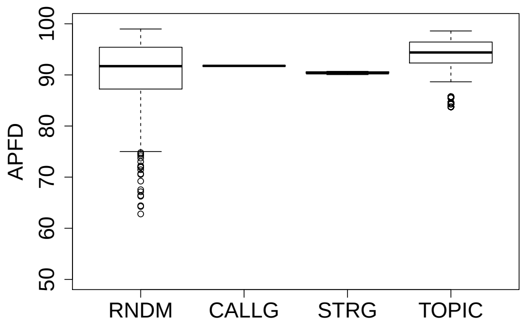
We conclude that our proposed topic-based TCP technique is more effective than state-of-the-art static techniques when applied to the studied systems.

3.5.2 Discussion

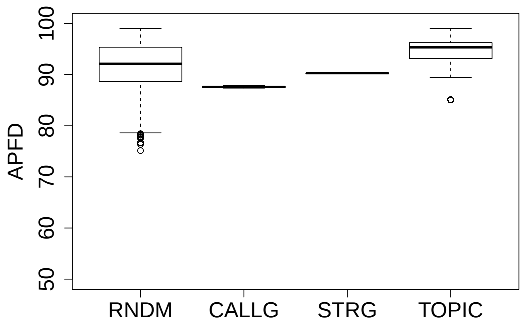
We now discuss our results to better understand the characteristics of our proposed topic-based technique. We start by providing a detailed example from our case study. We then compare the topic-based technique against a well known, execution-based technique, leveraging the fact that both techniques were tested on similar datasets. We then examine how the characteristics of each SUT's fault matrix affect the results of the tested TCP techniques. We then examine how sensitive the results of the topic-based technique is to its parameters. We then investigate whether the string-based technique is sensitive to the preprocessing steps. We outline practical advantages and disadvantages of the techniques. Finally, we enumerate potential threats to the validity of our case study.



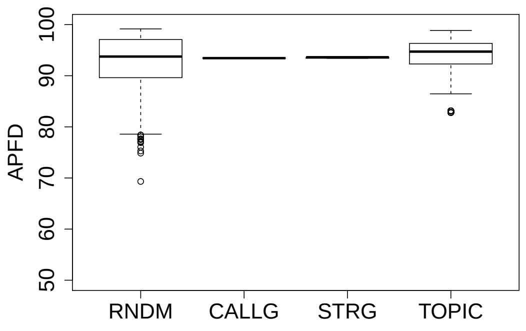
(a) Ant v7.



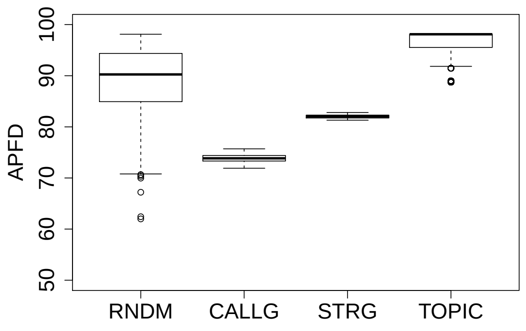
(b) Derby v1.



(c) Derby v2.



(d) Derby v3.



(e) Derby v5.

Figure 3.2: APFD results for each system under test (different subplots) and each technique (different boxes): random (RNDM), call graph (CALLG), string (STRNG), and topic (TOPIC). The boxplot show the distributions of 900 random repetitions of each technique (Section 3.4.6).

Comparison of Similarity Measures: An Example

To understand our technique in more depth, we present an example from our case study. Figure 3.3 contains an example from version v1 of the Derby DBMS. In this example, we compare one test script, `metadataMultiConn.java`, against all other test scripts in the suite using the similarity measures from the three static TCP techniques: topics, call graphs, and string distances. `metadataMultiConn.java` is a test script that tests the metadata columns of a DBMS using multiple connections. Our desire is to have a similarity measure that rates `metadataMultiConn.java` as highly similar to test scripts that also test metadata columns, such as `odbc_metadata.java`, `dbMetaDataJdbc30.java`, or `metadata.java`.

Figure 3.3 contains a snippet of `metadataMultiConn.java` along with snippets of three other test scripts, the closest for each of the three static TCP techniques. Table 3.4 shows the actual distances between the test scripts. According to the topic-based similarity measure, `metadataMultiConn.java` is closest to the `metadata.java` test script. As `metadata.java` also primarily exists to test metadata columns, the similarity with `metadataMultiConn.java` meets our desired similarity requirement.

However, with either the call-graph based or string-based similarity measures, `metadata.java` is only the 26th or 64th most similar to `metadataMultiConn.java`, respectively, indicating that these measures were unable to capture the similarity between these two scripts. Additionally, these measures did not rate any of our other desired metadata-related files as highly similar to `metadataMultiConn.java`. Using the string-based similarity measure, an unrelated test script named `backupRestore1.java` is considered the closest to `metadataMultiConn.java`, even though they share no obvious funtionalities. (`backupRestore1.java` creates a table in memory, backs it up on the hard disk, and restores it into memory.) The call graph-based similarity measure considers `executeUpdate.java` to be the most similar to `metadataMultiConn.java`, even though `executeUpdate.java` deals primarily with testing Derby's `Statement::executeUpate()` method, and has nothing to do with metadata columns nor multiple database connections.

Table 3.4: The distances between four example test scripts in Derby: `metadataMulti-Conn.java` (a), `metadata.java` (b), `backupRestore1.java` (c), and `executeUpdate.java` (d). Figure 3.3 contains snippets of these test scripts. We also show the rank of each distance, out of 98 total test scripts.

	Topic-based		String-based		Call graph-based	
	Dist.	Rank	Dist.	Rank	Dist.	Rank
(a) vs. (b)	0.401	1	293892	26	44	64
(a) vs. (c)	1.837	41	131250	1	37	44
(a) vs. (d)	1.952	57	400366	56	20	1

In this example, topic-based similarity better determine the similarity between pairs of test cases than existing string-based or call-graph based measures.

Comparison to Execution-based Results

While not the primary focus of our chapter, we now wish to compare our results with those of popular white-box execution-based TCP techniques to gain an understanding of the performance differences. The most popular white-box execution-based techniques are based on basic block coverage, where a block can be defined as an individual statement or a method (Elbaum et al., 2002; Yoo and Harman, 2010). Specifically, we consider the results of Zhang et al. (2009), who have reported the APFD results for the Ant system under test for two variations of their white-box execution-based technique, both based on method coverage information, which is consistent with our technique. Their first technique, called Method-total (MTT), orders tests cases based on the coverage of methods. Their second technique, called Method-additional (MAT), orders test cases based on the coverage of methods not yet covered.

We note that the mean APFD results reported for CALLG by Zhang et al. (87.8%) is quite similar to the results we report in this chapter (85.5%), encouraging us that our implementation of their technique is accurate, and that the comparisons above are fair.

The studies performed by Zhang et al. and this chapter are slightly different. Zhang et al. test 8 versions of Ant, and compute only one random sample per version. The results

```
public class metadataMultiConn {
    ...
  Connection conn = ij.startJBMS();
  conn.setAutoCommit(autoCommit);
  Connection conn1 = getConnection(args, false);
  metadataCalls(conn1);
  Connection conn2= getConnection(args, false);
  metadataCalls(conn2);
  ...
  DatabaseMetaData dmd = conn.getMetaData();
  ...
  DatabaseMetaData dmd = conn1.getMetaData();
  ...
}
```

(a) `metadataMultiConn.java`

```
/** Test of database meta-data. This program simply calls each of the meta-data
  * methods, one by one, and prints the results. */
public class metadata extends metadata_test {
  ...
  ResultSetMetaData rsmd = s.getMetaData();
  ...
  for (int i=1; i<=numCols; i++) {
    System.out.print("[" + rsmd.getColumnTypeName(i) + "]");
  }
  ...
```

(b) `metadata.java`

```
public class backupRestore1{
  ...
  Connection conn = ij.startJBMS();
  conn.setAutoCommit( false);
  ...
  //just open to a file in existing backup, so that rename will fail on next backup
  rfs = new RandomAccessFile("extinout/mybackup/wombat/service.properties", "r");
  ...
  PreparedStatement insStmt = conn.prepareStatement(...);
  insStmt.setBinaryStream( 2, new ByteArrayInputStream( blob1), blob1.length);
  insStmt.setAsciiStream( 3, new ByteArrayInputStream( clob1), clob1.length);
  ...
}
```

(c) `backupRestore1.java`

```
class executeUpdate{
  ...
  Statement stmt = conn.createStatement();
  int rowCount = stmt.executeUpdate("create table exup(a int)");
  if (rowCount != 0)
    System.out.println("FAIL - non zero return count on create table");
  else
    System.out.println("PASS - create table");
  ...
}
```

(d) `executeUpdate.java`

Figure 3.3: Four abbreviated test scripts from Derby v1. Full versions are online (Thomas, 2012). Table 3.4 shows the static TCP similarities between them.

they report are the mean APFD value across the 8 versions. We, on the other hand, collect 900 samples from a single version of Ant, and report the mean of these 900 samples. (We only tested a single version Ant, because the remaining versions did not meet the size requirements we imposed in Section 3.4.2.) Even though this is not an exact comparison, it can still shed some light in the comparison of the techniques. Zhang et al. present the mean APFD value for MTT and MAT to be 64.7% and 87.5%, respectively. Our results for the mean APFD value for TOPIC is 84.1%. Thus, TOPIC outperforms one execution-based technique, and has similar performance to another. Given that the execution-based techniques have much more information available, we find these results encouraging.

Our topic-based TCP technique has similar or better performance to execution-based techniques, even though the topic-based technique requires less information.

Effects of the Characteristics of Fault Matrices on Results

A fault matrix represents which faults each test case can detect. Rows represent test cases and columns represent faults: if entry i, j is 1, then test case T_i detects fault f_j. Fault matrices are not known in advance by practitioners (hence the need for TCP techniques), but each system has a fault matrix nonetheless.

The success of any TCP technique depends on the characteristics of the system's underlying fault matrix (Rothermel et al., 2002). If, for example, all test cases can detect all faults, then developers can use any TCP technique (even random!) and achieve identical results. Similarly, if each test case can detect exactly one fault, and each fault is detected by exactly one test case (an unsorted diagonal fault matrix), again all possible TCP techniques will achieve identical results, assuming that faults have equal importance. Prioritization techniques become necessary when the fault matrix is between these two extremes.

The characteristics of the fault matrices in the systems under test vary widely, which helps explain the differences in APFD results. Table 3.5 quantifies various characteristics of each fault matrix: the number of faults and tests; the failure rate; the average number of faults detected by each test case; the average number of test cases that detect each fault; the percentage of tests that detect no faults; the percentage of tests that detect at least half of the faults; and the percentage of tests that detect all of the faults.

Ant v7 has the lowest failure rate, lowest average number of faults per test case, and lowest average number of test cases that detect each fault. Additionally in Ant v7, no test cases can detect more than 50% of the faults, and 83% of the tests do not detect any faults at all. Thus, detecting faults in Ant v7 is relatively difficult, and performance by all techniques is generally the worst for Ant v7, compared to the other systems under test. Additionally, RNDM has worse performance for Ant v7 than any other system under test, since random guessing is likely to choose tests that detect no faults.

Derby v5 has the largest percentage of tests that can detect all of faults, the smallest percentage of tests that detect no faults, and the largest average number of faults detected per test case. These three characteristics combined indicate that faults are relatively easy to detect, compared to other systems. Thus, TCP techniques become less important, as the probability of selecting a valuable test case at random is high. Indeed, RNDM has significantly better performance than the CALLG and STRG techniques in this system. However, TOPIC is still able to outperform RNDM, highlighting that TOPIC is valuable even in the extreme case of easy-to-detect faults.

In the other three systems under test (Derby v1, Derby v2, and Derby v3), the fault matrices have more middle-of-the-road characteristics: they lie somewhere between the extreme cases of Ant v7 (faults are harder to detect) and Derby v5 (faults are easier to detect). In these systems, we found that the performance differences between the techniques under test were the smallest. Even still, TOPIC consistently outperformed the other techniques, indicating that TOPIC is the best choice even for systems with average characteristics.

TCP performance depends on the system's underlying fault matrix. However, our proposed topic-based technique is robust to all types of fault matrices: those where faults are easy to detect, hard to detect, or some middle ground.

Table 3.5: Characteristics of the fault matrices of the systems under test.

	Faults	Tests	Failure rate (%)	Avg. faults per test	Avg. tests per fault	% tests detecting X% faults		
						$X=0$	$X=50$	$X=100$
Ant v7	6	105	17.1	0.21	3.67	82.9	0.0	0.0
Derby v1	7	98	21.4	0.69	9.62	78.6	9.2	5.1
Derby v2	9	106	34.0	0.92	10.89	66.0	7.5	3.8
Derby v3	16	120	21.7	1.66	12.47	78.3	10.0	6.7
Derby v5	26	53	64.2	4.37	8.90	35.8	13.2	7.5

Run Time

To compare run times of the three static techniques, we ran a single instance of each technique on modest hardware (Ubuntu 9.10, 2.8 GHz CPU, 64GB RAM) and modest software (in R, the `lda` package for LDA, the `man` package for Manhattan distance, and our own implementation of the greedy algorithms) for our largest system under test, Derby v3. The topic-based technique ran in 23.1 seconds (22.9 to extract topics, <1 to compute the distance matrix, and <1 to run the greedy algorithm). The string-based technique ran in 29.6 seconds (29.15 to compute the distance matrix and <1 to run the greedy algorithm). Computing the distance matrix is longer for the string-based technique because the length of each string is much longer than the topic vectors. The call graph-based technique ran in 168.5 seconds (168.4 to extract the call graph of all tests using TXL and <1 to run the greedy algorithm).

The run times of all techniques, even with these unoptimized implementations, is trivial compared to the run time of most test cases (Hemmati et al., 2012; Rothermel et al., 2001). Further, the techniques scale: LDA can run on millions of documents in real time (Porteous et al., 2008) and many parts of all three techniques can be parallelized, such as extracting call graphs and computing distance matrices.

Parameter Sensitivity of LDA

In our experiment, the topic-based TCP technique used the LDA topic model to automatically create topics that are used to compare test cases. As described in Section 3.4.5, LDA depends on four input parameters: the number of topics, K; document-topic and topic-word smoothing parameters, α and β; and the number of sampling iterations, II. In our experiment, we fixed these values based on the characteristics of the SUTs. In this section, we investigate how sensitive our results are to the chosen parameter values.

For each of the four parameters, we compare two values to the baseline value used in our case study: some value smaller than the baseline value; and some value larger than the baseline value. For K, we consider $K'/2$ and $K' * 2$, where K' is the original value of K used in our case study. For α, we consider $\alpha'/2$ and $\alpha' * 2$. For β, we consider $\beta'/2$ and $\beta' * 2$. Finally, for II, we consider $II'/10$ and $II' * 10$. We keep constant all other settings and design decisions from our original case study (see Section 3.4).

Table 3.6 summarizes the values tested for each parameter and shows the results. We find that changing the values of the four parameters sometimes results in a change in mean APFD, but never by a large magnitude. Some parameter changes result in an increased mean APFD, and some parameter changes result in a decreased mean APFD. Some results are statistically significant, while others are not. We make two conclusions. First, although there is some variability in the results, and more work is needed to fully quantify the parameter space, we find that parameter values do not play a pivotal role in the results of the topic-based TCP technique. Second, the results of our original case study are not biased by showing only the results of the best possible parameter values.

Our topic-based TCP technique is not particularly sensitive to parameter values.

Effects of Preprocessing on String-based TCP

In our experiment, we implemented the string-based technique exactly as described by Ledru et al. (2011) in an effort to provide a fair comparison. In particular, we computed the string edit distance between two test scripts using their raw, unprocessed text. In our topic-based technique, before we compute the distance between test scripts, we perform a text preprocessing step that removes programming language punctuation and keywords, splits

Table 3.6: Results of the parameter sensitivity analysis. For each SUT, we show the baseline mean APFD, which are a result of the parameter values used in the original case study. We then change the value of each parameter and report the resulting mean APFD, along with a comparison with the baseline mean APFD, using the p-value of the Mann-Whitney U test (p-val), and the Vargha-Delaney A measure (A); see Section 3.4.7 for definitions of these metrics.

	Lower				Higher		
Value change	μ	p-val	A	Value change	μ	p-val	A
Ant v6 (Baseline: μ=84.8 using K=42, α=0.1, β=0.1, and II=200)							
K=21	83.6	<0.001	0.41	K=84	84.3	<0.001	0.45
α=0.05	83.0	<0.001	0.38	α=0.20	85.2	0.912	0.50
β=0.05	83.0	<0.001	0.36	β=0.20	83.8	<0.001	0.44
II=20	82.4	<0.001	0.32	II=2000	83.9	<0.001	0.45
Derby v1 (Baseline: μ=93.7 using K=39, α=0.1, β=0.1, and II=200)							
K=19	91.0	<0.001	0.30	K=78	94.0	0.774	0.50
α=0.05	92.0	<0.001	0.32	α=0.20	93.5	0.652	0.51
β=0.05	92.1	<0.001	0.41	β=0.20	92.3	<0.001	0.40
II=20	92.3	<0.001	0.39	II=2000	89.4	<0.001	0.17
Derby v2 (Baseline: μ=94.6 using K=42, α=0.1, β=0.1, and II=200)							
K=21	91.2	<0.001	0.23	K=84	95.3	<0.001	0.58
α=0.05	89.8	<0.001	0.11	α=0.20	94.5	0.008	0.54
β=0.05	94.1	0.353	0.51	β=0.20	93.3	<0.001	0.42
II=20	91.7	<0.001	0.29	II=2000	88.9	<0.001	0.09
Derby v3 (Baseline: μ=94.0 using K=48, α=0.1, β=0.1, and II=200)							
K=24	87.4	<0.001	0.16	K=96	92.1	<0.001	0.32
α=0.05	91.0	<0.001	0.21	α=0.20	94.7	<0.001	0.58
β=0.05	91.4	<0.001	0.33	β=0.20	93.2	0.222	0.48
II=20	91.5	<0.001	0.36	II=2000	87.7	<0.001	0.10
Derby v5 (Baseline: μ=96.4 using K=21, α=0.1, β=0.1, and II=200)							
K=10	93.1	<0.001	0.23	K=42	96.0	<0.001	0.44
α=0.05	90.4	<0.001	0.10	α=0.20	90.6	<0.001	0.09
β=0.05	95.8	<0.001	0.39	β=0.20	94.3	<0.001	0.32
II=20	95.2	<0.001	0.36	II=2000	94.7	<0.001	0.33

Table 3.7: Mean APFD results for the string-based technique without and with text preprocessing. We compare the two using the p-value of the Mann-Whitney U test (p-val), and the Vargha-Delaney A measure (A); see Section 3.4.7 for definitions of these metrics.

	Without preprocessing	With preprocessing	p-val	A
Ant v6	77.22	77.21	0.264	0.486
Derby v1	90.38	90.38	0.549	0.508
Derby v2	90.30	90.30	0.392	0.512
Derby v3	93.52	93.51	0.031	0.472
Derby v5	82.05	82.06	0.516	0.509

compound identifiers, removes stopwords, and performs word stemming, as is common with topic models (see Section 2.4). To determine whether the difference in performance between the string-based technique and topic-based technique is due to the techniques themselves, and not the text preprocessing step, we perform a simple experiment. Namely, we execute the string-based technique using the preprocessed test scripts (i.e., those used by the topic-based technique), and compare the results to those of the string-based technique using the raw test scripts. We keep constant all other settings and design decisions from our original case study (see Section 3.4).

Table 3.7 shows the results. We use the Mann-Whitney U test and the Vargha-Delaney A measure (see Section 3.4.7) to compare the two versions of the string-based technique. In all five SUTs, the p-value of the Mann-Whitney U test is greater than 0.01, indicating that there is not a significant difference between the two versions of the string-based technique. In addition, the A measure is always close to 0.5, further indicating that the two versions of the technique are comparable. Thus, we conclude that the performance of the string-based TCP technique is not dependent on the text preprocessing step, and that the success of the topic-based technique is not due solely to its text preprocessing.

The performance of the string-based TCP technique is not dependent on the text preprocessing step, and that the success of the topic-based technique is not due solely to its text preprocessing.

Practical Advantages of Static Techniques

Compared to existing execution-based techniques (based on code coverage information) and model-based techniques (based on specification models of the source code), static techniques enjoy several practical advantages.

First, there is no need to collect coverage information, a process that can be time-, money-, and resource-expensive due to the need to instrument and execute the entire test suite. Similarly, there is no need to create specification models of the SUT, also a process that can be time- and labor-intensive.

Second, there is no need to store coverage information on disk, which can become quite large and cumbersome for large systems with many tests. The data needed for static techniques are the test cases themselves, which are already being stored on disk.

Finally, with black-box static techniques, there is no need to maintain the coverage information or specification models as the source code and test cases evolve. Since black-box static techniques work directly on the test cases themselves, these techniques are independent of source code or model changes.

Scope of Static Techniques

Despite the advantages of static techniques over execution-based or model-based techniques, static techniques may not be appropriate for some TCP tasks.

First, static techniques work best on tests written in a high-level programming language. Static techniques will not perform well with short tests, or those written in command form. For example, the test cases of the Unix `bash` shell are specified as short scripts, such as `eval $1=\"\${$1:-$2}\"`. These types of tests do not have enough linguistic data or an extractable call graph for static techniques to gather data and diversify test cases.

Second, static techniques are not currently equipped to deal with input file or command line arguments to the tests. For example, say a single test case is executed against multiple different input files (e.g., matrices or configuration files) which cause the test case to exhibit vastly different behavior. Here, static TCP techniques will only operate on the text in the test case itself, and not in its associated input files, and thus the TCP technique may miss a valuable characteristic of the test case.

Another consideration of the call graph-based and topic-based techniques (but not string-based) is that these techniques treat test cases as unordered bags of words. Thus, these techniques cannot capture sequences of operations in test cases. For example, if one test case opened a connection to a database, inserted rows, and then closed the connection, while another test case opened a connection to a database, closed the connection, and then inserted rows (intentionally creating an error), a bag of words model would not differentiate the two: in the case of the call graph technique, the same methods are being called and therefore the test cases would be identical; in the case of the topic-based technique, the same topics are present in the test cases, and the distance metric would not detect a difference.

3.5.3 Potential Threats to Validity

Our case study provides an initial evaluation of a promising static TCP technique, and we are encouraged by the results. However, we note the following internal and external threats to the validity of our study.

Internal Validity. One potential threat to the validity of our results is our limited access to large, high-quality datasets. In particular, some of the SUTs exhibit characteristics that may not be representative of real-world systems. For example, for some SUTs, multiple test cases were able to detect all the faults in the system (Section 3.5.2). However, our results are still based on carefully selected systems from the widely-used SIR repository (Do et al., 2005).

The topic-based technique requires the topic modeling parameters—K, α, β, and the number of iterations—to be specified beforehand. However, there is currently no method for determining the optimal values of each parameter for any given dataset (Wallach et al., 2009), and empirically estimating the optimal values is not feasible without performing a large number of case studies. This problem is shared by topic modeling techniques in other communities, such as analyzing scientific literature (Griffiths and Steyvers, 2004) and bug localization (Lukins et al., 2010). Some research has proposed heuristics for determining the number of topics in source code (Grant and Cordy, 2010). In addition, as we found in Section 3.5.2, our results are not particularly sensitive to the exact values of the parameters. Still, further research is required to fully understand the parameter space.

Finally, the topic-based technique is based on a machine learning algorithm, which inherently involves randomness to infer the topics from the test scripts. As a result, different random number seeds may yield slightly different results. We mitigated this effect in our case study by executing 30 iterations of the topic-based technique, each with a different random number seed, and reporting the average of the results. Practitioners can mitigate the effects of randomness by using a sufficiently large number of Gibbs sampling iterations.

External Validity. Despite testing as many systems as possible from the publicly-available SIR repository, we still have only studied a limited set of systems. Our systems were all written in Java, were medium sized, and did not cover all possible system domains and testing paradigms. We therefore cannot quantify with any certainty how generalizable our results will be to other systems.

3.6 Conclusion

Many test case prioritization (TCP) techniques require the execution behavior or specification model of each test case. In this chapter, we considered the situation in which these information sources are not available, so-called *static* TCP. Further, we considered *black-box* static TCP, in which the source code of the system under test is not available. To this end, we proposed a new *topic-based* black-box static TCP technique, which uses topic modeling to abstract each test case into higher-level topics, based on the test cases' unstructured linguistic data. Using these topics, we calculated the dissimilarity between pairs of test cases, and gave high priority to those test cases that were most dissimilar, thereby diversifying our prioritization and casting a wide net for detecting unique faults.

Given the unstructured linguistic data in the test cases, there is a spectrum of text analysis methods that can be used to differentiate test cases. On the shallow extreme, one could use the simplest text comparison possible i.e., string equivalence, to compare test cases. While this technique is fast and simple, it has the potential to miss important aspects of the test cases and be misled by trivialities in the test scripts. On the deep extreme of the spectrum, one could use full natural language processing, complete with grammars, parse trees, and parts-of-speech tagging. This technique is powerful, but requires training data, is inefficient, and is error-prone, making the automation of TCP difficult. Somewhere in the middle of the spectrum is topic modeling, which uses a bag-of-word model and word co-occurrences to approximate the functionality from each test case. We feel that topic modeling is a good trade-off for comparing test cases, because it is fast and unsupervised, yet still offers strong discrimination between test cases.

In a detailed case study of five real-world systems, we compared our topic-based TCP technique to three baseline TCP techniques: random prioritization; a black-box version of call graph-based prioritization, and string-based prioritization. We found that the proposed topic-based prioritization outperforms existing black-box static techniques by up to 31%, and is always at least as effective as the baseline techniques. These results indicate that making use of the linguistic data in test cases is an effective way to statically prioritize test cases. Further, our technique enjoys the advantage of being lightweight, in that it does not require the execution behavior or specification models of the system under test, and instead operates directly on the test scripts themselves.

Although our technique can stand on its own, our technique can also complement existing techniques. The heart of a TCP technique is the similarity measure that is used to assess similarity of test cases. More sources of information are better: in the future one could combine the strengths of existing TCP techniques by combining the various sources of information to make a more informed similarity measure. In this scenario, the contribution of this chapter (i.e., a technique to extract useful information from the linguistic content of test cases) could be used to enhance *any* TCP technique, since even execution-based and model-based techniques always have the source code of test cases available.

In future test case prioritization work, we could fine tune our topic-based technique by investigating the effect of using different distance metrics between test cases, such as the Kullback-Leibler and Hellinger distance metrics. Likewise, we could consider additional distance maximization algorithms, such as hill climbing, genetic algorithms, and simulated annealing. We could consider other IR models, such as the Vector Space Model and Latent Semantic Indexing, for determining the similarities between test cases. Finally, we could perform additional case studies, containing additional SUTs as well as execution-based and model-based TCP techniques, to further verify our results.

CHAPTER 4

Measuring the Interaction Between Mailing Lists and Source Code

In this chapter, we consider a second new application of IR models in software engineering. We tackle the challenge of understanding how developers use their mailing list while developing software, which can provide insights into the development process, as well as pave the way for new tools to better manage the design decisions of the system. We present an IR-based technique to discover high-level topics in the mailing list and source code histories of a software system. Analysis of the extracted topics, and their interactions over time, allows us to discover and characterize relationships between the two repositories. Case studies on two open-source systems (Apache HTTP Server and PostgreSQL) show that topics can be used to establish a high-level relationship between the mailing list and source code. Our technique can thus be used to document the decisions and design processes behind software changes and makes this data accessible to practitioners and researchers.

Publications based on this chapter: Thomas et al. (2012b)

4.1 Motivation

CONSIDER THE TYPICAL life cycle of an open source software system. Brian comes up with a new idea, hacks together a prototype, and uses it for his own purpose. As his tool seems to be useful for a wider range of tasks, Brian uploads it to a public code repository with a dedicated mailing list. Soon, feature requests and bug reports trickle in via email, exposing critical flaws in the system's original design. Bernard and Beverley soon contribute small bug fixes. Beverley proposes novel restructurings to the architecture, whereas Bernard has ideas on how to scale up Brian's system. Based on the mailing list discussions of the three developers and other users, Brian's tool undergoes several iterations of restructuring and maintenance. After heated discussions on choosing the right XML parser, plans are made for an official 2.0 release. Right after their new release, the whole process repeats again.

When trying to understand the design of a software system like Brian's system, practitioners and researchers usually turn to the source code, as it is the most concrete and executable specification of a software system. Various complex mining techniques exist to abstract this concrete specification into a more semantic model of the *concerns* (conceptual units) that collaborate to implement the system (Robillard and Murphy, 2002). These techniques especially recover functional concerns such as the main algorithm, caching logic, and synchronization.

However, as illustrated in our fictitious example, the majority of the human thought process that went into the development iterations of Brian's tool is not directly encoded into the source code. Rather, it was only captured in mailing list discussions, bug repositories or even IRC chats. Although concern mining techniques might be able to reconstruct some of the intermediate design decisions (Adams et al., 2010) by analyzing several releases of Brian's tool, these techniques are still unable to record deprecated design options, nonfunctional topics like performance and reliability, and more process-related topics such as testing and release management.

Our overarching research goal in this chapter is to recover this relationship between the mailing list (where design decisions are discussed and made) and the source code (where design decisions are implemented). Even though mailing lists are usually accessible to the project developers, the sheer size of this repository (often with thousands of emails from hundreds of users) makes it impractical to manually read, organize, or gain insight from. Tools are thus needed to aid developers in recovering the conceptual links between mailing lists and source code.

Recent work has proposed recovering traceability links between emails and source code, based on keywords and regular expressions that look for class names in emails (Bacchelli et al., 2009; Baysal and Malton, 2007). Although these techniques are useful for certain

```
if (options.username)
{
    /*
     * The \001 is a hack to support the deprecated -u option which
     * issues a username prompt. The recommended option is -U
     * followed by the name on the command line.
     */
    if (strcmp(options.username, "\001") == 0)
        username = simple_prompt("User name: ", 100, true);
    else
        username = pg_strdup(options.username);
}
```

Figure 4.1: Example code snippet from PostSQL.

scenarios, we also need a technique to find more *conceptual* links that often can not be obtained by regular expressions alone.

Suppose Brian, our fictional developer, is now working on the popular open source DBMS, PostgreSQL. Brian is making a change to the `psql/startup.c` file, which handles global configuration options when the database is invoked. Figure 4.1 shows a curious excerpt from the code that mentions a hack to get around a deprecated command line option. Brian wonders why this option was deprecated in the first place, and why it is still supported with a hack. Surely, there is a reason and history why this is the case; how can he learn more about it?

On December 9, 2007, a message titled "*whats the deal with -u?*" was sent to the PostgreSQL developer mailing list. What followed was a 21-message discussion that examined in detail the reasons for deprecating the `-u` option, whether it was redundant with the `-U` and`-W` options, and what the typical use-cases were for all three of these options. This discussion would not only answer Brian's questions about the design decisions regarding why the `-u` option was deprecated, but would also give him a list of developers who are familiar with the issue (i.e., the developers involved in this mailing list discussion). However, the discussion never mentions the phrase `startup.c` or any of the function names in `startup.c`, so current linking techniques, based on keyword matching, would not be able to find this discussion for Brian.

In this chapter, we hypothesize that topics are shared between source code and mailing lists, and that we can recover this relationship using LDA. We further hypothesize that, by analyzing the shared topics between source code and mailing lists, we can begin to understand the life cycle of topics, from planning to eventual development. This motivates our research questions:

RQ1: How similar are the topics that are discussed in the mailing list and implemented in the source code? Intuitively, we expect that mailing list discussions primarily talk about the source code. However, it is not obvious that the two repositories will share topics, which are automatically discovered from the data based on the co-occurrences of words. It is possible, for example, that the source code uses a different vocabulary than the mailing list (e.g., "class" instead of "object"), which would mean that the topics would not be similar and therefore we could not use them for further analysis. We therefore wish to quantify the similarity of the topics in the two repositories.

RQ2: What is the temporal relationship between topic activity in the mailing list and topic activity in the source code? We wish to investigate the knowledge flow between the mailing list and source code. We hypothesize that topics will enter certain states at different points in time, such as a *discussion/planning* state, when the topic is active on the mailing list for some time before it is changed in the code, followed by an *implementation* state, when there is less discussion but more code changes. We wish to characterize and quantify this relationship and interaction.

4.2 Proposal

As conceptualized in Figure 4.2 and depicted in Figure 4.3, our technique to discover and evaluate topics shared between a project's mailing list and source code consists of mining data from the project's repositories and preprocessing the data so that LDA can be applied (Step A); applying LDA to the preprocessed data (Step B); and computing metrics on the output of LDA to analyze the results (Step C). In the following we discuss each step of our technique in more detail and outline implementation specific design choices.

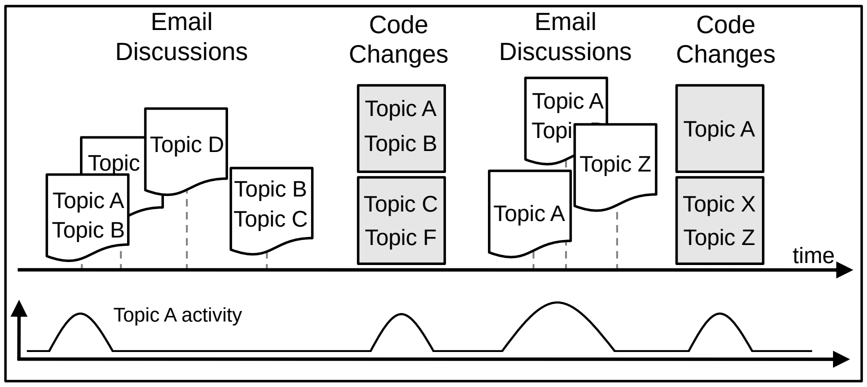


Figure 4.2: Our model of the interaction between emails and source code. Emails discuss certain topics, and later those topics are changed in the source code. The activity for a topic can be measured over time across both repositories.

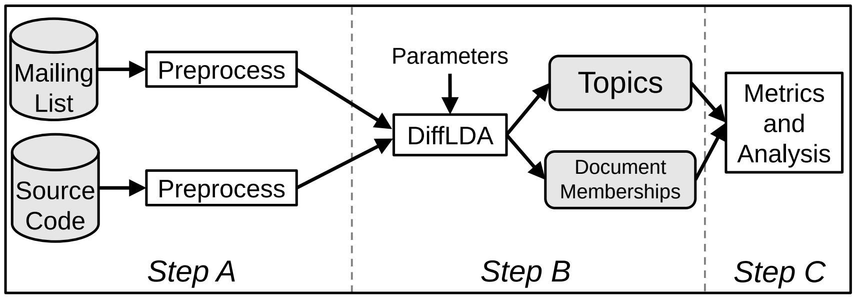


Figure 4.3: An overview of our technique.

4.2.1 Data Preprocessing

We collect mailing list data using an in-house tool, which downloads and processes mailing list archives in mbox format (Robles et al., 2009). We then collect the standard release versions of the source code from the system archives.

The preprocessing step for the mailing list involves various cleansing operations (Bettenburg et al., 2009). We first remove noise from each message, including personal signatures, quotes of previous emails, and other lines of little contribution to the content of the discussion, such as “On May 3rd, Brian wrote:”. We then remove copies of messages that are incorrectly sent multiple times by the mail server, remove attachments, and reconstruct discussion threads. We remove English language stop words, such as “the” and “it”, and finally stem each word into its base form.

The preprocessing step for the code (Section 2.2) involves extracting identifier names and comments, splitting, removing stopwords, and pruning rare and common words.

4.2.2 Applying LDA

We discover topics by applying the LDA model to the mail documents and source code documents at the same time. This process results in a set of topics, as well as a description of which topics are in which documents.

To apply LDA to the source code histories, we use the Diff model (Thomas et al., 2011), which computes LDA on the changes (diffs) between versions of a document instead of the versions themselves. The Diff model has been shown to reduce data duplication issues found in source code histories, which degrade the effectiveness of the topics that LDA discovers (Thomas et al., 2011). We combine the output of the Diff model with the preprocessed mail documents into one large corpus, which we use as input into the LDA model.

4.2.3 Metric Computation and Analysis

We are interested in the characteristics of each discovered topic, and how the mailing list interacts with the source code through these topics. We calculate metrics of the general behavior or use of a topic, such as how often a topic is mentioned in the mailing list, the total presence of a topic in the source code, and when a topic was added, removed, or changed in the source code.

To describe our analysis, we briefly introduce some notation. We say that there is a total of $n = n_c + n_m$ source code and mail documents input into LDA. LDA discovers K topics, $z_1, \ldots, z_K$, where each topic is a distribution (which sums to 1) over the words in the corpus vocabulary. Additionally, LDA produces an $n \times K$ document-topic matrix θ, where the entry θ_{ik} is the probability that topic z_k appears in document d_i.

Since in Step B we apply LDA to both code and mail documents at the same time, we introduce the following notation, which allows an easier description of our analysis.

– Document d_i contains a timestamp $\tau(d_i)$, a type $t(d_i) \in \{c, m\}$ (code or mail), and a number of words $|d_i|$.

– For code documents ($t(d_i) = c$), there are v timestamps (versions) of the corpus, $V = \{V_1, ..., V_v\}$.

– For mail documents ($t(d_i) = m$), there are p periods in the corpus, $P = \{P_1, ..., P_p\}$.

– To make our equations general, we define the notion of a time unit T, which can be either a timestamp or a period. Time unit T contains $|T|$ documents.

– $Docs(T, x) = \{d_i \mid \tau(d_i) \in T,\ t(d_i) = x\}$ denotes all documents of type x in time unit T.

With this notation, we can isolate documents at specific times or periods and of specific types (mail or code), making analysis straightforward. We calculate several topic metrics of interest, which can be organized into two groups: those measured on the source code documents and those measured on the mail documents.

Source Code Metrics

The ***code support*** of a topic measures the number of source code documents that contain this topic with membership greater than the threshold δ:

$$\mathit{Support}_c(z_k, T) = \sum_{d_i \in Docs(T,c)} I(\theta_{ik} \geq \delta),$$

where I is a function which returns 1 if its argument is true and 0 otherwise. The ***code weight*** of a topic measures the total presence of a topic in the source code, which is the same as the total number of words across all source code documents that come from this topic:

$$\mathit{Weight}_c(z_k, T) = \sum_{d_i \in Docs(T,c)} \theta_{ik} \cdot |d_i|. \tag{4.1}$$

The ***code churn*** of a topic represents the number of source code documents matching this topic that have changed in a given time period:

$$\mathit{Churn}(z_k, T) = \sum_{d_i \in Docs(T,c)} I(\theta_{ik} \geq \delta) \cdot \mathit{Churn}(d_i, T), \tag{4.2}$$

where $\mathit{Churn}(d_i, T)$ is the churn of document d_i, that is, a binary indicator of whether the document has changed since its previous version.

Mail Metrics

The ***mail support*** and ***mail weight*** of a topic, $\mathit{Support}_m$ and Weight_m, are defined in a similar way as their code counterparts, except on mail documents. Mail support measures the number of emails that contain this topic in time unit T, while mail weight measures the total number of words in the mailing list that come from the given topic.

Source Code and Mail Metrics

Finally, we define the ***code weight ratio*** of a topic,

$$\mathit{WeightRatio}_c(z_k) = \frac{\mathit{Weight}_c(z_k)/n_c}{\mathit{Weight}_c(z_k)/n_c + \mathit{Weight}_m(z_k)/n_m}, \tag{4.3}$$

where

$$\mathit{Weight}_c(z_k) = \max_{V_j} \mathit{Weight}_c(z_k, V_j) \text{ and}$$
$$\mathit{Weight}_m(z_k) = \operatorname*{sum}_{P_j} \mathit{Weight}_m(z_k, P_j),$$

to be the ratio of a topic that is found in the source code documents (versus the mail documents). The mail weight ratio is the inverse of the code weight ratio (i.e., $\mathit{WeightRatio}_m(z_k) = 1 - \mathit{WeightRatio}_c(z_k)$). We use the $\max$ operator on code documents because each version of the code is mostly similar to the previous version, and hence summation would artificially inflate the weight ratio.

Table 4.1: Systems under study.

	Apache	PostgreSQL
		Mailing List
Dates	Jan. 2006–Apr. 2010	Jan. 2000–Dec. 2008
Number of messages	16,130	140,192
Number of threads	4,006	20,442
Number of total words[1]	912,467	5,768,355
		Source Code
Dates	Nov. 2005–Aug. 2010	May 2000–Nov. 2008
Releases	2.2.0–2.3.8	7.0.0–8.3.5
Number of releases	16	46
Number of files[2]	258	844
Number of SLOC[2]	149,564	501,202
Number of total words[1]	6,702,992	40,491,956

[1]After preprocessing. [2]In latest release.

Although many additional metrics can be measured (Thomas et al., 2010b), the metrics presented above will allow us to perform the analysis needed to answer our research questions.

4.3 Case Studies

4.3.1 Systems Under Study

We study the data from two open-source systems: PostgreSQL and Apache HTTP Server (Table 4.1). PostgreSQL is a large open source database system that has been actively developed for over 15 years (PostgreSQL, 2012). PostgreSQL is a natural choice for our study due to its thorough documentation, rich source code and mailing list repositories, and large practical impact over the years. Apache HTTP Server (hereafter Apache) is an open-source web server developed by the Apache Software Foundation (Apache Foundation, 2012a). Apache is a popular web server and undergoes rigorous software maintenance practices.

4.3.2 Study Design

We now describe the choices and parameters in our study. Unfortunately, there are no perfect methods for determining optimal values for each parameter described below, but in order to increase the repeatability of our case studies, we note each choice explicitly and present our motivation behind it.

Message Granularity. Threads are collections of logically related email messages, for example, an initial question and all the subsequent replies. Our data includes a thread identifier for each individual message in the mailing list, allowing us to aggregate the messages of each thread into a single document. Aggregating to this coarser level of granularity has the benefits of larger document size (a benefit for LDA) and more context for messages. Many individual messages only contain implicit references to previous messages in the thread, such as "Sure, I'll fix that". Taken on an individual basis, such messages lack any context and thus will not be useful for our purposes. We thus aggregate messages into threads for our study.

LDA Implementation. We use MALLET version 2.0.6 (McCallum, 2012). In addition to other useful features, MALLET has the ability to automatically generate a two-word label for each topic, based on commonly occurring bi-grams in the documents containing the particular topic. The generated label helps human readers better understand the nature of the topic.

LDA Parameters. The LDA model requires four input parameters: the number of topics, K; the number of sampling iterations; and α and β, smoothing parameters. We run 10,000 sampling iterations, after which the model has converged (Griffiths and Steyvers, 2004). MALLET automatically optimizes for α and β using state-of-the-art techniques (McCallum, 2012).

To determine the number of topics, K, we use a standard technique (Griffiths and Steyvers, 2004): we vary K and evaluate the fit of the model to our data by computing the log likelihood. We consider values of 25, 50, 100, 200, 300, 400, 500, 600, 700, 800, 900, and 1,000. For PostgreSQL data, we found that the log likelihood is monotonically increasing in our range of K values, but experiences diminishing returns as K increases beyond 200. As we are aiming for a balance between a good model fit (high likelihood) and low model complexity (low K), we set K to 200. Analogously, we set K to 100 for the Apache data.

Binary membership threshold (δ). By design, LDA yields a membership value for each topic in each document, ranging from 0.0 (document has no membership in this topic) to 1.0 (document has full membership in this topic). However, in some situations we need to determine, in a binary way, whether a document contains a topic. We have observed in practice that many documents have a small but nonzero membership for topics that the document is obviously not associated with, a result of the probabilistic nature of the LDA model. To account for this, we introduce a parameter, δ, that is used as a cut-off point for membership values to determine whether a topic is in a document. Ideally, δ would cause noisy memberships to be discarded while preserving true memberships. After experimentation, we found that the results of our analysis are relatively insensitive to the exact value and feel that $\delta =$0.10 represents a reasonable choice.

Shared topic threshold (ψ). To determine if a topic is shared or not (Research Question 1), we investigate the code weight ratio metric, i.e., how much of a topic that is found in the code versus in the mail. A code weight ratio of 0.0 means that the topic is never in the code, and a code weight ratio of 1.0 means that a topic is never in the mail, both indicating the topic is not shared between the two repositories. Due to noise in the LDA probabilistic process, ratios of, for example, 0.01 may occur for topics that are not really in the code. To account for this, we say that a topic is not shared if it has a ratio below $\psi =$ 0.05 or above 1$-\psi =$ 0.95.

Churned and discussed thresholds (γ_c and γ_m). To determine, in a binary way, if a topic has churned or not churned in a certain release, we introduce a threshold $\gamma_c =$ 1.0. We categorize a topic z_k as *churned* in time unit T if *Churn*$(z_k, T) \geq \gamma_c$ (i.e., at least one file as changed), and as *not churned* otherwise. Similarly, we define a *discussed* threshold $\gamma_m =$

1.0, meaning a topic is *discussed* in period P if at least γ_m message(s) occurred in P, and *not discussed* otherwise.

4.4 Results and Discussion

4.4.1 Topic similarity between the mailing list and source code (RQ1)

Approach

As explained in Section 4.2, we use our technique to discover topics from the source code and mailing list. We measure the code weight ratio (Equation 4.3) of the discovered topics to determine how much of a topic is found in the code versus how much of the topic is found in the mail. An extreme code weight ratio (i.e., 0.0 or 1.0) for a topic indicates that the topic is *not* shared between the source code and mailing list, while moderate values indicate that the topic is shared by both.

Findings

We find that the mailing list and source code share many topics. Tables 4.2 and 4.3 present selected topics from each studied system. We show a selection of topics specific to the code (high code weight ratio, low mail weight ratio), specific to the mail (low code weight ratio, high mail weight ratio), and shared between code and mail (similar ratios). The code-specific topics mostly deal with low-level programming concepts, such as errors, parameter validation, threads, and bytes, as well as data-structure/implementation concepts, such as queues and parsing. The mail-specific topics deal with meta-system concepts, such as version control, patches, revisions, work schedules, and compilation (e.g., `make`). The balanced topics contain concepts such as text search, user authentication, sessions, and bucket brigades (data containers in Apache).

Figure 4.4 presents the code weight ratio of each topic. The figure illustrates that there is a range of percentages: some topics are found only in the code (top right), most topics are found to varying degrees in both the code and mail (middle), and some topics are found only in the mail (bottom left). Using cut-off values of .05 and .95 (Section 4.3.2), we see that 58% (Apache) and 64% (PostgreSQL) of the topics are shared—they are found in both the mailing list and source code. This provides evidence for our hypothesis that the mailing list and source code discuss similar topics. For the remainder of this chapter, we remove topics that are not shared and focus our analysis on those topics that are. There are now 58 and 129 topics in Apache and PostgreSQL, respectively.

We find that between 58% and 64% of the topics are shared between the mailing list and source code.

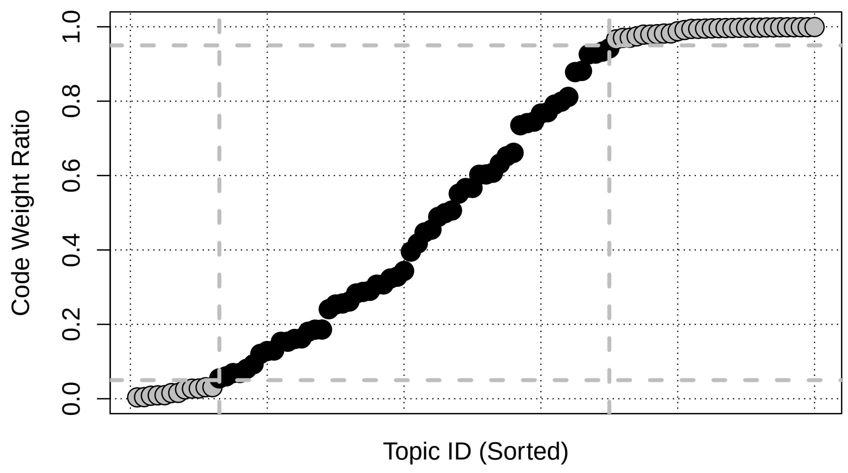
4.4.2 The temporal relationship between topic activity in the mailing list and source code (RQ2)

Approach

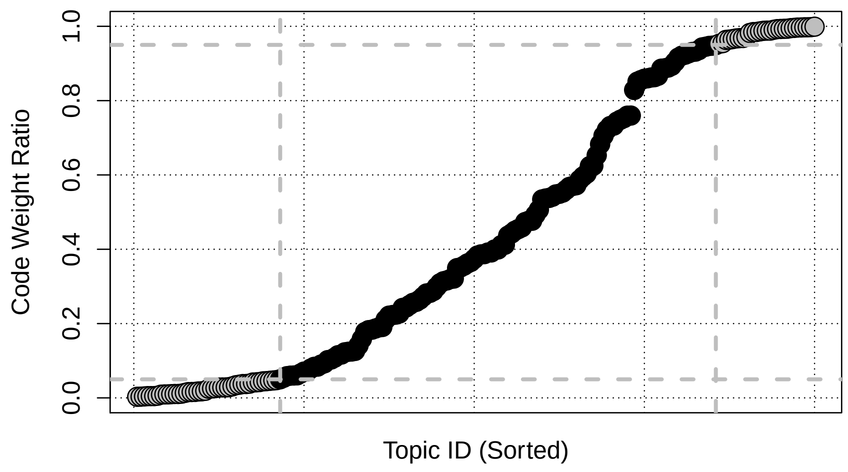
In Section 4.2.3, we defined the mail support and code churn metrics for a topic. As these two metrics are orthogonal, we define four possible ***topic life cycle states***, shown in Figure 4.5: Inactive (topic has low code churn and low mail support), Discussion (low code churn and high mail support), Implementation (high code churn and low mail support), and Active (high code churn and high mail support). We categorize each topic at each version into one of these four states by comparing the topic's mail support and code churn metrics to thresholds that divide the plot region into the four states. We use mail support and churn thresholds of 1.0 and 1.0, respectively (Section 4.3.2). We now investigate more closely the characteristics and patterns of topic churn and mail support, as defined by the four topic life cycle states.

Findings

We find that we can characterize the high-level information flow between the mailing list and source code. Figure 4.6 shows a heatmap of topic life cycle states over time. The heatmap shows that many topics enter all four states at least once in their lifetime (26% in Apache, 54% in PostgreSQL), while even more topics enter three or more states (72% in



(a) Apache: 58% of topics are shared.



(b) PostgreSQL: 64% of topics are shared.

Figure 4.4: Ratio of a topic found in the code (code weight ratio). Each dot is a topic, and the topics are sorted from lowest ratio to highest.

		Mail Support	
		Low	High
Code Churn	High	*Implementation*	*Active*
	Low	*Inactive*	*Discussion*

Figure 4.5: The four topic life cycle states, based on the code churn and mail support metrics.

Table 4.2: Selected topics from Apache. Only Balanced Topics are further considered for RQ2 and RQ3.

k	Label	Top Words	$WeightRatio_c$	$Support_c$	$Weight_c$	$Support_m$	$Weight_m$
Example Code-specific Topics							
91	cmd parm	*cmd arg parm err config filenam server*	0.999	106	22669	21	50
63	pfd desc	*pollset ring desc pollfd pfd poll set num*	0.999	23	4910	6	11
93	error aplog	*thread child pid server process worker mpm listen*	0.999	34	20141	15	45
79	buf buf	*buf len size byte nbyte iov str base*	0.998	109	11077	17	34
77	mime magic	*req variant type rec mime content charset accept*	0.998	24	10855	9	49
78	fprintf stderr	*fprintf printf stderr argv exit argc err option*	0.998	82	8316	18	37
Example Mail-specific Topics							
10	apr apr	*apr util pool success revision copi index work*	0.004	1	59	2766	28512
80	httpd httpd	*httpd trunk url modul modifi author log date*	0.008	4	133	2686	29287
21	open sourc	*list apach httpd project develop user asf peopl*	0.009	5	205	4074	40235
39	attach patch	*patch trunk backport commit work propos fix review*	0.010	3	193	4849	36464
68	releas releas	*releas vote httpd test tag tarbal alpha start*	0.015	2	140	2289	16439
100	make sens	*make work mod thing don time code good*	0.028	29	1412	9944	90986
Example Balanced Topics							
13	uri uri	*uri url method path request redirect locat intern*	0.506	36	4535	997	8070
53	tabl header	*header tabl pool xff hdr kei content err*	0.551	52	4889	601	7253
85	hook hook	*hook handler run modul function config pre data*	0.567	47	4242	777	5922
24	statu statu	*statu error code success set err return intern*	0.567	96	5471	1031	7626
82	session session	*session set add form save kei request base*	0.602	11	3303	325	3978
5	bucket brigad	*bucket brigad alloc data creat read insert ctx*	0.603	56	10382	734	12463

Table 4.3: Selected topics from PostgreSQL. Only Balanced Topics are further considered for RQ2 and RQ3.

k	Label	Top Words	$WeightRatio_c$	$Support_c$	$Weight_c$	$Support_m$	$Weight_m$
Example Code-specific Topics							
8	errcod errcod	*errcod ereport errmsg error invalid paramet syntax undefin*	0.999	195	23515	67	147
144	conn conn	*conn msg statu error messag sock els buffer*	0.998	54	18426	65	247
132	line gram	*str line make gram cat ival type opt*	0.998	38	29363	82	465
27	fprintf stderr	*stderr printf fprintf buf char argv exit endif*	0.997	157	22309	94	362
157	ptr ptr	*ptr cur state len word els entri val*	0.997	92	15655	83	298
10	dst dst	*free ctx uint dst kei len init buf*	0.997	85	14559	44	307
Example Mail-specific Topics							
198	candl pha	*hard backup drive life avenu hill pennsylvania drexel*	0.003	0	34	5790	64964
195	perl python	*perl python tcl plperl languag modul spi core*	0.004	0	14	3149	22823
142	postgr sql	*php postgr sql apach linux build admin middlewar*	0.005	0	16	3143	20649
42	candl pha	*backup drive hard life pha candl road squar*	0.005	0	62	7528	80068
186	web site	*email yahoo site onlin postgresql icq yscrappi url*	0.005	0	24	4434	31158
84	red hat	*rpm instal postgresql build packag red hat tar*	0.010	0	85	6017	55900
Example Balanced Topics							
87	outer join	*join queri outer claus select planner optim plan*	0.351	30	4441	5090	52081
5	enabl disabl	*option enabl disabl flag set configur mode turn*	0.362	25	2877	5817	32213
41	prepar statem	*statem cursor prepar execut fetch queri declar paramet*	0.366	26	3730	4245	41031
188	hba conf	*password user auth usernam authent shadow connect encrypt*	0.374	29	2959	3044	31459
105	start postmast	*postmast backend process pid start server exit startup*	0.383	36	7000	7200	71494
137	text search	*word search text tsearch configur index english function*	0.386	15	2323	2721	23425

Apache, 79% in PostgreSQL). The most common state in both studied systems is the Active state (55% in Apache, 51% in PostgreSQL), and the least common is the Inactive state for Apache (9%) and the Implementation state for PostgreSQL (6%).

Another class of observations we can make from Figure 4.6 is the characteristics of releases (columns):

- In release 6 (Apache), most topics are in the Active state (78% of all topics).
- In release 11 (Apache), most topics are in the Discussion state (46%).
- In release 14 (Apache), most topics are in the Implementation state (47%).
- In release 22 (PostgreSQL), most topics are in the Discussion state (90%).

Figure 4.7 shows state transition diagrams for Apache and PostgreSQL. Each node in the diagram is one of the states of a topic; the edge weights between the nodes are the median probability of a topic transitions from one state to another. We make the following observations.

- When a topic is being discussed, it is most likely to become active in the next state.
- When a topic is active, it is most likely to stay active in the next state.
- When a topic is inactive, it is equally likely to enter any of the other three states.
- When a topic is being implemented, it is most likely to stay implemented or become active (Apache) or to become inactive (PostgreSQL).

Note that a lack of an edge indicates that the median probability across all topics was 0 for that transition, i.e., at least half of the topics never made that transition. This is different than saying a topic *never* made that transition. Indeed, all transitions were made at least once, but some transitions were less frequent than others.

The state transition diagrams for Apache and PostgreSQL differ in a few ways.

- Topics in Apache are much more likely to be in the Implementation state (21% vs. 6%), and once there, much more likely to stay there (67% vs. 12%).

– Topics in Apache spend less time in the Discussion state than in PostgreSQL (16% vs. 35% in PostgreSQL), and once there, less likely to stay (10% vs. 38%).

– Topics in Apache rarely enter the Inactive phase compared to PostgreSQL.

The topic life cycle heatmap, as well as state transition diagrams, provide initial insight into how topics are being discussed and implemented in the systems under study, and provide a method to easily describe the current state of a topic at any given time. For example, if a developer is interested in finding discussions about a certain topic, she can use the heatmap as a guide. If the topic is Inactive for several releases in a row, she need not look at the source code or emails from the time period, and instead focus on the documents from another period.

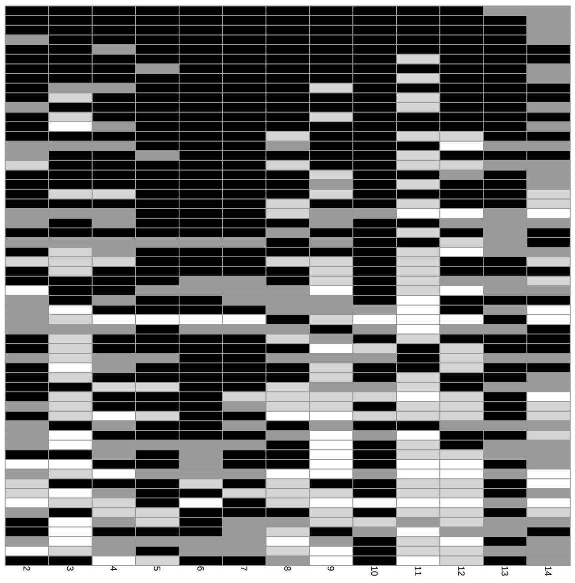
Topic life cycle states can measure the information flow between the mailing list and source code of a system.

4.4.3 Discussion

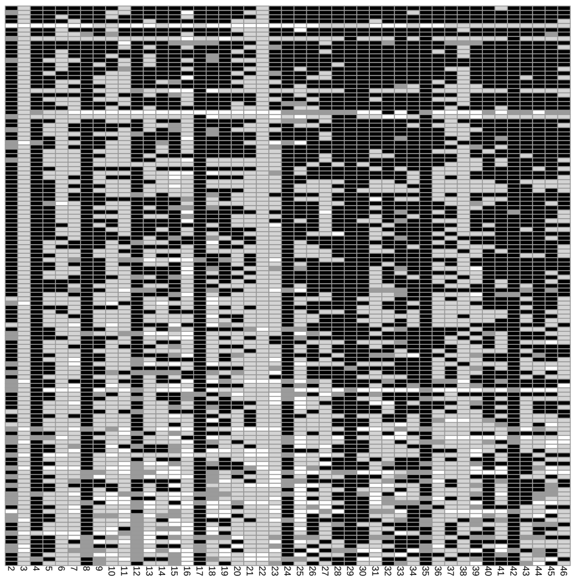
In this chapter, we have empirically demonstrated a sharing of topics across both the source code itself and developer emails. In addition, our empirical study has shown that topics are not static artifacts, but evolve over time, entering a variety of life cycle states as development moves on. In the following, we highlight the broader implications of our findings, for both practitioners and researchers alike.

Until now, only anecdotal evidence has existed about the information overlap in mailing lists and source code. Our results provide empirical evidence that suggest that methods to find traceability links between mailing lists and source code have a realistic chance of succeeding. Through our case studies, we have found that between 58%–64% of the topics in the mailing list can be also found in the source code, suggesting not only that traceability methods might succeed, but also that topic models may be a good tool to help discover the links.

In addition, designers and developers of IDEs could use these results to improve the

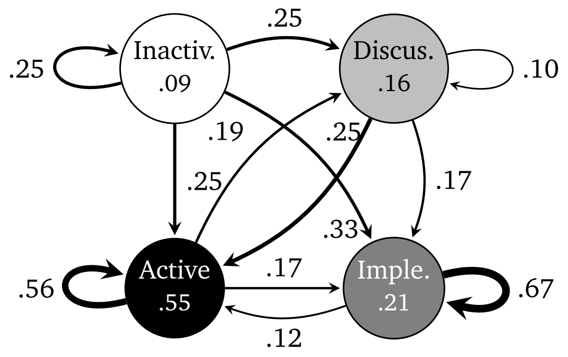


(a) Apache.

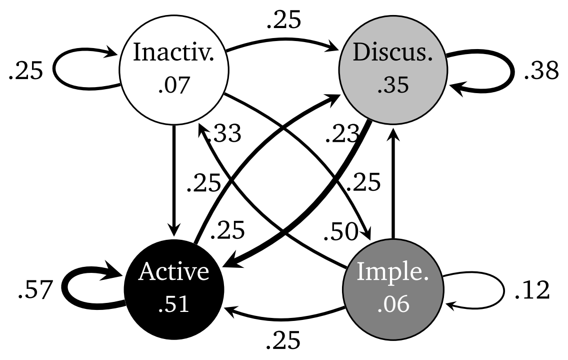


(b) PostgreSQL.

Figure 4.6: Topic life cycle states over time. Rows represent topics, while columns are versions of the code. The color of each cell indicates the state of development that the topic is currently in: white is Inactive, light gray is Discussion, dark gray is Implementation, and black is Active. Rows are sorted by number of state changes.



(a) Apache.



(b) PostgreSQL.

Figure 4.7: State transition diagrams for topic life cycles. Edge weights show the probability of transitioning from one state to another, and are averaged (median) across all topics. Weights inside the nodes indicate how often topics were in that state. Colors are the same as in Figure 4.6.

functionality of their tools, taking a step towards *software explanation*. By linking the design decisions and discussions about a source code unit to the implementation of that unit, practitioners can go back in time and understand the rationale behind changes. Imagine right-clicking a piece of code in an IDE and selecting “Show me why this piece of code came to be written the way it is now” and the IDE displays the discussions related to the topic that the piece of code is related to. We argue that providing access to such discussions would greatly enhance developers’ abilities to fully understand the code. This is somewhat different than current techniques (for example, Bacchelli et al. (2011)), which only create traceability links between a class and a discussion if the discussion explicitly mentions the class name. Using topics as the method to link code with discussions is sure to find additional relevant discussions which do not explicitly mention the class name; this is indeed the purpose of topic models (Blei and Lafferty, 2009). However, there may be a price to pay in terms of the precision of the links; more work is needed to quantify this trade-off.

Additionally, our technique can be used to determine which topics require the most discussion and which require the most implementation. This information can be used to guide training decisions or documentation efforts.

Finally, project stakeholders can use our technique to monitor the progress of a topic’s development. Consider the following example. A team decides to develop a new printing feature in their system. The team spends time discussing and deliberating over the best implementation strategy: should they use Java’s own Printing API, the GDSPrinting API, or develop their own from scratch to better suite their needs? How should the user specify printing options—under the Settings tab or in a new dialog box? After these types of decisions have been made, the team begins to make code changes to the required classes. These actions can all be observed at fine-grain level, for example by listing all email discussions and source code changes containing the “printing” topic. They can also be viewed at a higher level, as the life cycle state of the “printing” topic goes from the initial Inactive state, then to the Discussion state, followed by the Implementation and Active states, and finally back to the Inactive state. Stakeholders can thus use shared topics as a means to monitoring the status of the system, in addition to the methods they already use, such as number of bugs fixed and number of tests passed.

4.4.4 Threats to Validity and Future Work

Data preprocessing. Preprocessing the mailing list and source code so that information retrieval methods can be applied is a challenging task (Bettenburg et al., 2009). Although we were able to remove some noise from our datasets, our techniques are not perfect and affect the quality of our results. More work is needed to find better preprocessing techniques.

Parameter choices. As stated previous in Section 4.3.2, our study involved the use of many parameters that control, for example, the number of topics and message granularity, as well as thresholds to polarize values. Although we experimented with each parameter to determine reasonable values, we cannot be sure we have chosen optimally.

Additional case studies. We have performed detailed case studies on two world-class software systems whose source code and mailing lists are rich with information. Still, additional case studies are needed to determine how generalizable our results are.

Finer-grained analysis. We have analyzed the release versions of the source code, giving us a clear snapshot of the code over time. However, we may be missing important details that can only be found by analyzing each revision of the source code repository.

4.5 Conclusion

Our overarching research goal is to make intelligent use of the mailing list to support source code maintenance. As source code design decisions often take place in the mailing list, and not in the source code itself, the mailing list has great potential to help maintainers understand the insight behind the current state of the source code.

However, the mailing list is often large and difficult to digest. Finding appropriate emails for a given concept is not an easy task, as Boolean searches are often not sufficient. We therefore propose the use of IR models to find a higher-level conceptual link between the mailing list and source code.

In this chapter, we demonstrated that LDA was able to discover topics that were shared between the mailing list and source code, a trait not immediately obvious given the different nature of the two repositories. Case studies on PostgreSQL and Apache revealed that LDA was indeed able to discover a large portion of shared topics, providing evidence for the conceptual links we seek.

In addition to finding conceptual links, we also used topics to study the relationship (i.e., topic life cycle states) between the mailing list and source code. We hypothesized (and found) patterns in how topics transition between different states: sometimes topics are discussed for a period of time on the mailing list, followed by a period of implementation. Other times, topics are inactive altogether. But most of the time, topics are discussed and implemented at the same time. We quantified these states and described how topics evolve through them by creating a state transition diagram for each of our case studies.

This chapter represents a first step towards our research goal: we can find shared topics, and we can use these topics to study the relationship between the mailing list and source code. Implications of our findings include empirical evidence that traceability links can be established between mailing lists and source code using topic models, that IDE developers should consider adding support for mailing lists directly into the IDE, and that systems can be monitored using topic life cycle state changes.

Part III

Advanced IR Techniques

We found in Section 2.4 that most research to date uses basic IR techniques, such as VSM or LSI. Further, most research only uses at most one IR model. In this part of the thesis, we present two advanced IR techniques that both enhance the state-of-the-art in mining unstructured software repositories. In the first, we present a framework for combining multiple IR models. In the second, we use an advanced IR model, namely a topic evolution model, to analyze source code evolution over time. Our results indicate that researchers and practitioners should consider advanced IR techniques, in addition to the traditional basic IR techniques, to increase the effectiveness of their techniques.

– **Chapter 5: A framework to combine disparate IR models**. Most research to date builds only a single IR model, with a single set of parameter values for the model. However, in many situations, multiple IR models each have different strengths that can be harnessed. We present a framework to combine the results of any number of disparate IR models, which allows the strengths of individual models to be harnessed. We evaluate our framework in the context of bug localization, where the goal is to identify which source code entities need to be changed to fix a given bug report. We conduct two experiments on three real-world software systems: Eclipse, Mozilla, and IBM Jazz. We find that model combination is powerful: our framework achieves improved performance over the best individual IR models, even when the constituent models do not perform well individually.

– **Chapter 6: Using topic evolution models to analyze source code evolution**. Currently, researchers and practitioners usually monitor the evolution of source code by plotting numeric metrics—such as lines of code or number of bugs closed—over time. However, this monitoring technique does not provide insight into which features or implementation topics are being added or modified. In this chapter, we use an advanced IR technique, namely a topic evolution model, applied to multiple versions of source code, to uncover the evolution of source code topics over time. In two detailed case studies, we manually analyze the topics and their evolutions to determine whether they correspond with actual changes made by developers. Through case studies on two open source systems, JHotDraw and jEdit, we find that the evolutions accurately describe developer changes

87–89% of the time. Using this advanced IR technique brings new capabilities and insights to both developers and managers.

CHAPTER 5

A Framework to Combine Disparate IR Models

Most research to date uses a single IR model to mine unstructured software repositories. However, different IR models excel in different areas, and we postulate that combining the results of disparate IR models will likely achieve better results than any individual model. In this chapter, we propose the use of an advanced IR technique, that of IR model combination. We evaluate the advanced technique in the context of bug localization and find that combination almost always achieves better performance compared with the best individual model, often with significant performance increases.

Publications based on this chapter: Thomas et al. (2012d)

5.1 Motivation

BUG LOCALIZATION is defined as a classification task: given n source code entities and a (unstructured, natural language text) bug report, classify the bug report as belonging to one of the n entities (Lukins et al., 2010). The classifier returns a ranked list of possibly-relevant entities, along with a relevancy score for each entity in the list. An entity is considered *relevant* if it indeed needs to be modified to resolve the bug report, and *irrelevant* otherwise.

Current bug localization research uses IR classifiers to locate source code entities that are textually similar to bug reports, based on the unstructured text fields in the bug report. However, current research only uses the results of a single IR model, even though various

IR classifier configurations each have their own strengths and weaknesses, and each may be more appropriate in different situations. This leads to the idea that: rather than trying to determine a single classifier that performs best in every situation, we can use an ensemble (i.e., combination) of classifiers to robustly localize bugs in many situations. We further postulate that this technique would easily extend to any kind of bug localization classifier: we can combine IR-based classifiers with dynamic analysis classifiers, defect prediction classifiers, or any classifier that somehow solves the bug localization task, partially or wholly. As long as the classifiers are uncorrelated in their wrong answers (i.e., the classifiers make different mistakes from each other), then combining them will likely improve performance (Ho et al., 1994). Given the nature of the bug localization task, which has many possible wrong answers for a given bug report (i.e., all the source code entities that are unrelated to the bug report), combining models is likely to perform well (Misirli et al., 2011).

5.2 Proposal

We propose a framework for combining multiple classifiers that can together achieve better bug localization results than any single classifier. The main intuition behind classifier combination is that when a particular source code entity is returned high in the list for many classifiers, then we can guess with high confidence that the entity is relevant for the bug report. There is a rich literature in the pattern recognition community for combining classifiers, also known as classifier ensembles, voting experts, or hybrid methods (Ho et al., 1994; Kittler et al., 1998; Misirli et al., 2011). No matter the name used, the fundamental idea is the same: individual classifiers often excel in different cases and make different mistakes.

Specifically, our proposed framework (illustrated in Figure 5.1) for combining multiple bug localization classifiers has two main constituents. First, any number of classifiers are created, based on the available input data and the given bug report. Second, the classifiers are combined using any of several combination techniques, such as Borda count (Van Erp and Schomaker, 2000), score addition, or Reciprocal Rank Fusion (Cormack et al., 2009).

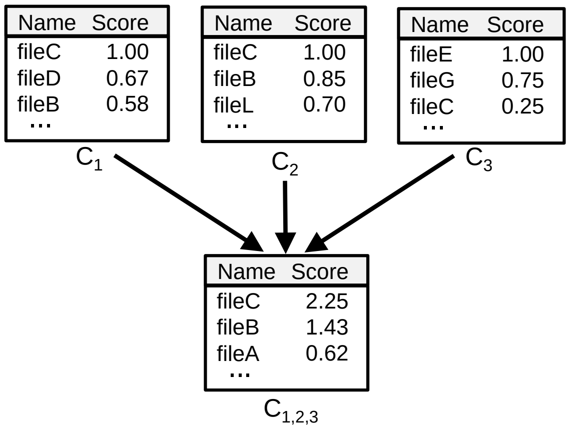


Figure 5.1: An illustration of our classifier combination framework. Here, three classifiers are created, based on the available input data and the given bug report. Then, the classifiers are combined, using score addition, to produce a single ranked list of source code entities. In this example, fileC bubbles up to the top of the combined list since it is high on each of the three classifiers' lists.

We now describe each constituent in more detail.

5.2.1 Classifiers

As previously mentioned, classifiers can come in many forms. IR-based classifiers attempt to find textual similarities between the given bug report and the available source code entities. Entity metric-based (EM-based) use entity metrics (Zimmermann et al., 2007), such as lines of code, to classify which source code entities likely to have the largest number of bugs, independent of the given bug report. In fact, we consider any variant of any technique that returns a ranked list of source code entities as a classifier. Formally, we define the *result set* of a classifier C_i, which operates on a given bug report q_j, as

$$C_i(q_j) = \Big\{ \Big(r(d_k), s(d_k) \Big) \;\; \forall d_k \in D \Big\}$$

where $r(d_k)$ is the rank of entity d_k, $s(d_k)$ is the score of entity d_k, according to C_i, and D is the set of all entities. The result set of a classifier consists of a rank and a (relevancy) score for every source code entity in the system. Note that scores need not be unique; in fact, many entities may be assigned a score of 0. In this case, they all share a rank of $M+1$, where M is the number of entities that received a non-zero score.

5.2.2 Combination Techniques

Given a set of $|C|$ classifiers, we may combine them in any of several ways. A simple rank-only combination is the Borda Count method (Van Erp and Schomaker, 2000), which was originally devised for political election systems. For each source code entity d_k, the Borda Count method assigns points based on the rank of d_k in each classifier's result set. For example, if classifier C_i assigned d_k a rank of 1, then the Borda Count for d_k would be M-1, where M is the number of entities that received a non-zero score in C_i. The entity as rank 2 would receive a score of M-2, and so on. The Borda count scores for all C classifiers are tallied for each entity, and the entity with the highest total Borda Count score is ranked first, and so on. Formally, the score for an entity d_k is defined as

$$\text{Borda}(d_k) = \sum_{C_i \in C} M_i - r(d_k \mid C_i), \tag{5.1}$$

where M_i is the number of entities that received a non-zero score in C_i, and $r(d_k \mid C_i)$ is the rank of entity d_k in C_i. The entities are then ranked according to their total Borda score.

Instead of using the ranks, the scores of each classifier can also be used. For example, the score of each entity d_k and each classifier C_i is summed to produce a total score for each entity:

$$\text{ScoreAddition}(d_k) = \sum_{C_i \in C} s(d_k \mid C_i). \tag{5.2}$$

Usually, the scores of each classifier are scaled to be in the same range (e.g., [0,1]) before combination to avoid unintentionally weighting the importance of certain classifiers. However, equations 5.1 and 5.2 can be modified to explicitly weight certain classifiers differently from others.

We note that the result of combining $|C|$ classifiers defines a new classifier. This classifier can itself be combined with other classifiers. In this way, a hierarchy of classifiers can be constructed. However, in this chapter, we only consider one level of combination.

5.3 Case Studies

This case study investigates the performance improvements that can be achieved by combining classifiers, using the framework introduced in Section 5.2. In this section, we describe two experiments: one based on a small number of manually-created classifier sets, and another based on a larger number of randomly-created classifier sets.

Classifiers Under Test

We consider two families of classifiers: IR-based classifiers and entity metrics-based classifiers.

IR-based Classifiers IR-based classifiers are based on IR models. We build classifiers based on three popular IR models, defined in Section 2.2: VSM, LSI, and LDA. Based on all the possible combinations of parameter values for these models (shown in Tables 5.1 and 5.2), we build a total of 3,168 IR-based classifiers. We describe the motivation behind the chosen parameter values in Chapter 8 of this thesis; in this chapter, we simply use this large set of classifiers as a means to evaluate classifier combination.

Entity Metric-based Classifiers The past decade has been very active for research in the area of bug prediction (D'Ambros et al., 2012). Briefly, this research aims at measuring features of the source code, such as lines of code (LOC), past bugginess, change proneness, and logical coupling between classes, to predict which source code entities contain bugs.

Entity Metric-based (EM-based) classifiers use the insight from the bug prediction literature that many entity metrics are highly correlated with an entity's bug count. To this end, EM-based classifiers first calculate one or more metrics on the source code entities. Then, the classifiers rank the source code entities based on the metrics. For example, a higher

LOC metric indicates more bugs, so one EM-based classifier would sort the entities by their LOC metric.

Observe that, unlike IR-based classifiers, the rankings of EM-based classifers are not based on the given bug report, so the same ranked list will be created for every bug report. Still, we note (and the bug prediction literature confirms) that since bugs are highly concentrated in a small number of source code entities, this list is likely to be fairly accurate for any given bug report.

The EM-based classifiers have only a single parameter: which entity metric is used to determine the bugginess of an entity. We consider four metrics, shown in Table 5.3: the lines of code (LOC) of the entity; the churn (i.e., number LOC that were added, deleted, or changed since the previous version) of an entity; the cumulative bug count (i.e., the number of bugs that have been associated with this entity in the past) of the entity; the new bug count (i.e., the number of bugs only since the previous version) of an entity. Previous research has shown that these metrics are good predictors of entity bugginess, so we expect the metrics to have reasonable performance for bug localization.

Naming Classifiers A classifier in either family is defined by a *configuration*—a set of parameter values that specify the behavior of the classifier. We use the parameter values to succinctly describe a particular configuration. For example, VSM.A1.B3.C0.D1.E2 defines a VSM-based classifier which uses the bug report title for queries (A1), trains the VSM model on identifiers and comments from the source code entities (B3), performs no preprocessing (C0), uses tf-idf term weighting (D1), and uses the overlap similarity measure (E2). (We omit a parameter if it only has a single value, such as the LDA similarity metric.) Similarity, EM.M1 defines an EM-based classifier that uses the LOC metric M1.

Studied Systems

We study three software systems: Eclipse JDT, Mozilla mailnews, and Jazz (Table 5.4). Eclipse is a large, popular integrated development environment (IDE) written in Java (Eclipse

Table 5.1: The IR configuration parameters and their values that are common to all of of the classifiers we study.

Parameter	Value
Bug report represention	A1 (Title only) A2 (Description only) A3 (Title+description)
Entity represention	B1 (Identifiers only) B2 (Comments only) B3 (Idents+comments) B4 (PBR-All) B5 (PBR-10 only) B6 (Idents+comments+PBR-All)
Preprocessing steps	C0 (None) C1 (Split only) C2 (Stop only) C3 (Stem only) C4 (Split+stop) C5 (Split+stem) C6 (Stop+stem) C7 (Spit+stop+stem)

Foundation, 2011). Eclipse JDT is the subset of Eclipse that implements Java development tools. Mozilla is an application suite concerned mostly with web browsing and email clients (Mozilla Foundation, 2012b). Written mostly in C++, Mozilla is one of the largest and most active open-source systems to date; due to its size and the requirements of our case study, we only consider the largest module of Mozilla, mailnews. Jazz is a propriety IDE developed by IBM (IBM, 2012).

We choose these systems mainly for two reasons. First, these systems are large, active, real-world systems, which allow us to perform a realistic evaluation of the classifiers under test. Second, each system carefully maintains bug tracking databases and source code version control repositories, which will allow us to build our ground-truth datasets to evaluate the classifiers. Table 5.5 gives example bug reports.

Data Collection

We begin by obtaining the raw bug data from the bug tracking database and the source code from the version control system (VCS) for each studied system.

Table 5.2: The IR family of classifiers we study. We show the configuration parameters and the values that we consider for each of the three underlying IR models: VSM, LSI, and LDA.

	Parameter	Value
Parameters for VSM only		
	Term weight	D1 (tf-idf) D2 (Sublinear tf-idf) D3 (Boolean)
	Similarity metric	E1 (Cosine) E2 (Overlap)
Parameters for LSI only		
	Term weight	F1 (tf-idf) F2 (Sublinear tf-idf) F3 (Boolean)
	Number of topics	G32 (32 topics) G64 (64 topics) G128(128 topics) G256 (256 topics)
	Similarity metric	H1 (Cosine)
Parameters for LDA only		
	Number of topics	J32 (32 topics) J64 (64 topics) J128 (128 topics) J256 (256 topics)
	α	L1 (Optimized based on K)
	β	M1 (Optimized based on K)
	Number of iterations	N1 (Until model convergence)
	Similarity metric	K1 (Conditional probability)

Table 5.3: The EM family of classifiers we study. We show the configuration parameters and the values we consider.

Parameter	Value
Metric	M1 (Lines of code) M2 (Churn) M3 (New bug count) M4 (Cumulative bug count)

Table 5.4: Studied systems.

	Eclipse (JDT)	Jazz (All)	Mozilla (mailnews)
Domain	IDE	IDE	Web
Language	Java	Java	C/C++/Java
License	Open	Closed	Open
Years considered	2002–2009	2007–2008	2004–2006
Snapshots	16	8	10
Bugs (preprocessed)	3,898	2,818	1,368
Source code files	1,882–2,559	756–887	319–332
KSLOC	232–506	133–168	173–193

Table 5.5: Example bug reports in the three studied systems, and their relevant source code entities.

System	Bug ID	Title	Relevant entity(s)
Eclipse	102645	“[open type] Open Type history shows stale visibility info in type history”	`util.TypeInfo.java`
	106638	“[misc] Support BiDi chs in logical expr in java editor”	`JavaSourceViewer.java`
	100062	“[formatting] Code formatter is broken on test case from bug 99999”	`CodeFormatterVisitor.java`
	100302	“StackOverflowError during completion”	`CompletionParser.java`
Jazz	28284	“Create button enabled for process iterations while editor is loading”	`processpart.java`
	18317	“Move team area dialog should not allow to show archived areas.”	`teamareamovewizard.java`
	21247	“Offer to switch to edit mode when I start typing in the over page”	`wikiformpage2.java`
	30963	“Error messages in bluesdev server log for I20070912-2000”	`auditableserver.java`
Mozilla	105964	“Drop 20 bytes off each imgRequest object”	`imgRequest.cpp`
	24668	“[DOGFOOD]Crash when clicking on Finish on Account Setup”	`xpcwrappedjsclass.cpp`
	11001	“[4.xP] Table spacing borders incorrect at http://www.choochem.com”	`nsElementTable.cpp`
	222023	“regression: pref parser should accept single-quote delimited strings”	`prefread.cpp`

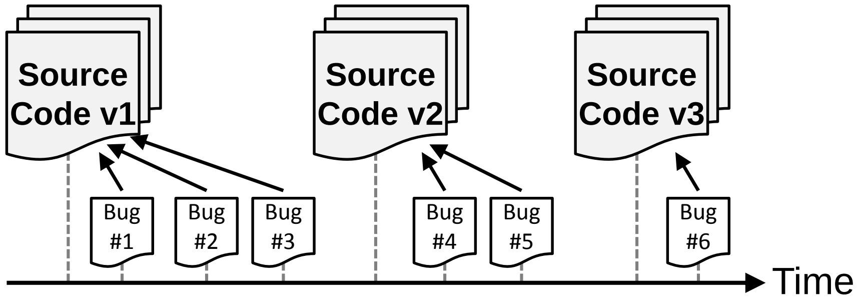


Figure 5.2: Our evaluation procedure. First, we take snapshots of the studied systems' source code at 6 month intervals to build the classifiers. For each bug report under test, we use the classifiers built on the most recent (relative to the bug report) snapshot.

Creating the Ground Truth To create the ground truth data that we use to evaluate relevancy, we use the syntactic analysis portion of the popular SZZ linking algorithm to link resolved bug reports to the version control system that were changed to resolve the bug (Sliwerski et al., 2005). The algorithm parses the commit log messages from the source code repository (e.g., CVS or SVN), looking for messages such as "Fixed Bug #45433" or similar variations. If found, the algorithm establishes a link between all the source code entities in the commit transaction with the identified bug ID. The algorithm uses several heuristics that in concert find fairly accurate links (Sliwerski et al., 2005). The result is a reliable set of links between bug reports and source code entities, which we use to evaluate the classifiers under test.

Source Code Preprocessing We work at the file level of granularity. We preprocess the source code entities at each snapshot according to the specified classifier configuration.

Bug Report Preprocessing After collecting bug reports from the bug repositories, we preprocess them in the following manner. First, we remove bug reports that meet one or more of the following conditions.

– Bug report not marked as "FIXED"
– Bug report does not result in at least one entity change. Some bugs, such as bug #6994 in Eclipse, deal with meta-source code issues, such as configuring an IDE correctly to build

the system. Fixing these bugs results in no actual entity changes, and thus no links are created.

– Bug report has empty title field after pre-processing

– Bug report links to build or configuration files (we are only interested in source code entities)

We preprocess the remaining bug reports according to the specified classifier configuration. For a given classifier, the same preprocessing steps are applied to the source code and bug reports.

Evaluation Procedure

Ideally, for each bug report in our testing set, we would take a snapshot of the source code at the exact time the bug report was filed, build all classifiers using data from that snapshot, and perform the classification. This process would provide the most accurate and realistic evaluation scenario, because it uses the source code that the classifiers would have available in a real-world situation. However, doing so would be too computationally expensive for our evaluation, as we are evaluating thousands of bugs and thousands of classifiers, each taking upwards of several minutes to build using our unoptimized research prototypes. As a compromise, we first take snapshots of each system's source code from the studied systems' VCS at six month intervals over the duration of the system and precompute the classifiers at each snapshot (Figure 5.2). Given a bug report during evaluation, we determine the most recent snapshot of the source code and use the corresponding precomputed classifiers. This process allows us to consider temporally appropriate source code for each bug report without requiring substantial computation.

For each classifier under test, we perform the following procedure. For every bug report in each studied system, we determine the nearest snapshot, and use the corresponding classifier to obtain the ranked list of source code entities. Using the ground truth, we determine the rank of the first relevant entity on the list.

Performance Metrics

To measure the performance of a individual classifier, we use the top-k accuracy metric. The ***top-k accuracy*** metric measures the percentage of bug reports in which at least one relevant source code entity was returned in the top k results. Formally,

$$\text{top-}k(C_j) = \frac{1}{|Q|}\sum_{i=1}^{|Q|} I\big(\exists d \in D \mid \text{rel}(d, q_i) \wedge r(d \mid C_j, q_i) \leq k\big),$$

where $|Q|$ is the number of queries (i.e., bug reports), q_i is an individual query, rel(d, q_i) returns whether entity d is relevant (using truth information) to query q_i, $r(d|C_j, q_i)$ is the rank of d given by C_j in relation to q_i, and I is the indicator function, which returns 1 if its argument is true and 0 otherwise. For example, a top-20 accuracy value of .25 indicates that for 25% of the bug reports, at least one relevant source code entity was returned in the top 20 results. Previous work has set k to 20, with the idea that 20 is a reasonable number of entities for a developer to search through before growing impatient and resorting to other means of bug localization (Nguyen et al., 2011).

Other metrics which are commonly used for evaluating IR models, such as precision, recall, and mean average precision (MAP), are inappropriate for our purposes. In bug localization, the task is to locate the *first* relevant entity for a given bug report, and therefore there is at most one correct answer in the list of returned entities; precision, recall, and MAP metrics are appropriate only when there are many possible correct answers.

Experiment 1: Manually-created Classifier Sets

In this experiment, we investigate whether combining the best-performing classifier configurations (as identified in Section 8) can improve performance.

As noted in Section 5.2, classifier combination works best when the individual classifiers err in different ways. Thus, choosing classifiers that are likely to result in the most uncorrelated mistakes—such as those based on different data sets or those based on different underlying IR models—is likely to have the best result. Accordingly, we consider classifier

Table 5.6: Classifier sets under consideration.

	$\|C\|$	C = Classifiers
CS1	5	VSM.A1.B3.C7.D1.E1, VSM.A2.B3.C7.D1.E1, VSM.A1.B5.C7.D1.E1, VSM.A2.B5.C7.D1.E1, EM.M3
CS2	5	LDA.A1.B3.C7.J256, LDA.A2.B3.C7.J256, LDA.A1.B5.C7.J256, LDA.A2.B5.C7.J256, EM.M3
CS3	5	LSI.A1.B3.C7.F2.G128, LSI.A2.B3.C7.F2.G128, LSI.A1.B5.C7.F2.G128, LSI.A2.B5.C7.F2.G128, EM.M3
CS4	13	CS1 ∪ CS2 ∪ CS3

sets that are based on different input data representations.

Recall that for IR-based classifiers, two data sources must be represented: that of the source code entity, and that of the bug report. We define a set of four VSM-based classifiers, shown in Table 5.6 as CS1, as follows. The first classifier in the set uses the entities' textual content (B3: identifiers and comments) and the bug reports' titles (A1), and optimal settings for the other three parameters (preprocessing, term weighting, and similarity measure). The second classifier uses the the entities' textual content (B3) and the bug reports' descriptions (A2), again with optimal settings for the other three parameters. The third classifier uses the entities' past bug reports (B5) and the bug reports' titles (A1), and the fourth classifier uses the entities' past bug reports (B5) and the bug reports' descriptions (A2). Finally, the set also includes the best-performing EM classifier, EM.M3.

We define similar sets based on LDA (CS2 in Table 5.6) and LSI (CS3). Finally, we combine CS1–CS3 into a new set, CS4.

These classifier sets each contain five classifiers that (a) operate on independent data representations (e.g., the identifiers and comments will be very different from the past bug reports) and (b) have optimal values for the other parameters. Thus, we expect that combining these classifiers will increase overall performance.

For each of the classifier sets defined above, we consider two combination techniques: the Borda count method (BRD) and the score addition method (ADD), each described previously in Section 5.2.2. We run each classifier set on all 8,084 bugs reports in the three studied systems, and calculate the top-20 performance of each classifier set.

Experiment 2: Randomly-generated Classifier Sets

We conduct a second experiment to investigate whether classifier combination helps in situations where the best configuration of each classifier is not known in advance.

Similar to Experiment 1, we define classifier sets with five classifiers: four based on different IR data sources, and one based on entity metrics. However, in this experiment, we do not use the optimal values for the other parameters. Instead, we let them vary randomly.

Specifically, we randomly build a classifier set with the following classifiers (using regular expression notation): VSM.A1.B3.*.*.*, VSM.A2.B3.*.*.*, VSM.A1.B5.*.*.*, VSM.A1.B5.*.*.*, and EM.*. We build 50 such sets, each time randomly choosing values for the varying parameters (i.e., "*"). We do the same for LDA and LSI, yielding a total of 150 randomly-generated classifier sets. (We provide a description of the resultant sets online (Thomas, 2012).) Doing so allows us to examine the effects of many situations: combining good classifiers with bad classifiers; bad with bad; and so on. It also allows us to determine whether combination only helps in the case of optimally-performing classifiers, or whether it helps in the general case.

For each classifier set, we again consider two combination techniques: the Borda count method (BRD) and the score addition method (ADD). We run each classifier set on all 8,084 bugs reports in the three studied systems, and calculate the top-20 performance metric.

5.4 Results and Discussion

Experiment 1: Manually-created Classifier Sets

Table 5.7 shows the top-20 performance of the four classifier sets, as well as their relative performance improvements over the best individual classifier in the sets, for both the Borda count and score addition methods. The relative improvement of classifier set CS is calculated as

$$\text{RI}(CS) = \frac{\text{top-20}(CS) - \max_{C_i \in CS}\left(\text{top-20}(C_i)\right)}{\max_{C_i \in CS}\left(\text{top-20}(C_i)\right)}.$$

In all three studied systems, classifier combination improves performance for all classifier sets, often significantly. In Jazz, for example, classifier set CS2, which combines four LDA configurations and EM.M3, results in a 95% relative improvement over the best individual classifier, going from a top-20 performance of 33% to a top-20 performance of 64%. We find similar results for each other studied systems, classifier sets, and combination techniques, indicating that combining the best-performing classifiers can improve bug localization performance.

Combining IR classifiers results in a 10.3%–95% improvement in top-20 performance. Both the Borda count and score addition combination techniques always lead to improved performance in our experiments.

Experiment 2: Randomly-generated Classifier Sets

Table 5.8 shows the percentage of the 150 classifier sets that were improved by combination, in terms of top-20 performance. Overall, classifier combination helps in the large majority of sets. In Eclipse, 93% or 84% of the classifier sets had a better performance after combination, depending on whether the Borda count or score addition methods were used, respectively. In Mozilla, the results were even better: 88% or 96% of the classifier sets improved after combination. The best results came in Jazz, where 97% and 100% of classifiers sets were improved.

Table 5.8 also quantifies the amount of improvement the classifier sets experienced after combination. The mean relative improvements for Eclipse were 37% and 35% for the Borda count and score addition methods, respectively. This means that for a random combination of classifiers, one can expect performance to improve by at least 35%. The same is true for the other two studied systems: in Jazz, the mean relative improvements were 56% and 54%, and in Mozilla they were 14% and 19%.

Even with a random set of classifiers, classifier combination improves the top-20 performance in a large majority (84%–100%) of cases, and by a significant amount (+14%–+56%).

Table 5.7: Top-20 performance of the four manually-created classifier sets, CS1–CS4 (see Table 5.6), and their relative improvements over the best individual classifier in the sets.

	Performance of best individual classifier	Performance of combined classifiers			
		Borda count	Relative improvement (%)	Score addition	Relative improvement (%)
Eclipse					
CS1: 4 VSM + Best EM	0.467	0.706	+51.2	0.666	+42.7
CS2: 4 LDA + Best EM	0.405	0.522	+28.9	0.464	+14.8
CS3: 4 LSI + Best EM	0.405	0.708	+75.1	0.661	+63.3
CS4: 4 VSM + 4 LDA + 4 LSI + Best EM	0.467	0.699	+49.5	0.640	+37.0
Jazz					
CS1: 4 VSM + Best EM	0.576	0.739	+28.4	0.737	+28.1
CS2: 4 LDA + Best EM	0.330	0.643	+95.0	0.568	+72.2
CS3: 4 LSI + Best EM	0.438	0.732	+67.3	0.721	+64.7
CS4: 4 VSM + 4 LDA + 4 LSI + Best EM	0.576	0.697	+21.1	0.694	+20.7
Mozilla					
CS1: 4 VSM + Best EM	0.739	0.844	+14.2	0.837	+13.3
CS2: 4 LDA + Best EM	0.667	0.787	+18.0	0.717	+7.6
CS3: 4 LSI + Best EM	0.703	0.836	+18.9	0.820	+16.6
CS4: 4 VSM + 4 LDA + 4 LSI + Best EM	0.739	0.815	+10.3	0.818	+10.7

Table 5.8: Improvement of classifier combination in 150 randomly-generated classifier sets. We show the percentage of classifier sets in which the performance of the combination was better than the performance of the best individual classifier in the set. We also report summary statics of the relative improvement that combination provides.

		% of sets improved	Relative improvements provided by combination					
		by combination	Min	1st Qu.	Med.	Mean	3rd Qu.	Max.
Eclipse	Borda count	92.7	-20.0	17.7	39.4	37.2	55.1	121.0
	Score addition	84.0	-33.9	14.7	36.9	35.1	55.9	95.0
Jazz	Borda count	97.3	-16.3	37.6	56.8	56.1	72.8	142.2
	Score addition	100.0	0.3	43.0	54.4	54.3	65.7	114.3
Mozilla	Borda count	88.0	-37.7	7.3	14.4	13.9	22.2	53.7
	Score addition	96.0	-6.1	12.5	17.8	19.2	24.3	56.5

Discussion

The goal of this case study was to determine whether classifier combination improves bug localization performance. In Experiment 1, which combined the best-performing individual classifiers, we found that classifier combination always improves performance. In Experiment 2, which combined random sets of classifiers, we found that classifier combination usually (i.e., at least 84% of the time, and up to 100% of the time) improves performance. These results provide strong evidence that classifier combination is a valuable method for improving bug localization performance.

We saw the smallest relative improvements in Mozilla, and the largest in Eclipse. We note that individual classifiers in Mozilla already have high performance (i.e., top-20 values above .80), leaving little room for improvement for combination. Individual classifiers in Eclipse, on the other hand, have relatively worse performance (a maximum top-20 value of 0.54).

In general, the Borda count combination method worked better than the score addition method. In all four manually-created classifier sets, and all studied systems (Table 5.7), the Borda count offered a greater improvement than score addition (with one exception: the Borda count method in CS4 in Mozilla had a relative improvement of 10.3%, compared to score addition's 10.7%). In addition, when considering the 150 randomly-created classifier sets, the Borda count method offered better mean and median relative improvements over score addition, for the Eclipse and Jazz systems (Table 5.8). In Mozilla, the mean and median relative improvements of Borda and score addition were comparable.

In these experiments, we combined sets of five component classifiers, based on the logic that classifiers using different data sources as input will result in uncorrelated errors. We also investigated combining all 3,172 component classifiers of Case Study 1. Surprisingly, we found that the top-20 performance of their combination to be comparable to the top-20 performance of the best individual classifier. (Specifically, the relative improvements ranged from -2% to +12%, depending on the studied system and combination method used.) Given that the set of 3,172 contains many classifiers with very low performance, it is encouraging that their combination still achieves such high performance.

5.4.1 Threats to Validity

This section discusses potential threats to the validity of our case studies.

Internal Validity

One potential threat to the internal validity of our case studies is our truth data collection technique, which was based on part of the SZZ algorithm (Sliwerski et al., 2005). Even though the SZZ algorithm is the state-of-the-art algorithm for linking bug reports to source code entities, given a bug repository and a history of source code changes, recent research has discussed the algorithm's linking biases (Bachmann et al., 2010; Bird et al., 2009a; Nguyen et al., 2010).

External Validity

Even though we performed three extensive case studies on large, active, real-world systems, our results still must be considered in context. In particular, our datasets represent only a fraction of all real-world systems, domains, programming languages, and development paradigms, so we cannot definitively say that our results will hold for all possible systems.

5.5 Conclusion

In this chapter, we cast the bug localization task as one of classification, and introduced an advanced IR framework to combine the results of disparate classifiers—such as those based on IR models, entity metrics, dynamic analysis, or any algorithm whatsoever—using well-known combination techniques.

Combining various classifiers greatly improved bug localization performance. In all three studied systems, we found that combining classifiers resulted in better performance than any individual classifier, both when the individual classifiers were optimal and when the individual classifiers were suboptimal. This is true no matter the underlying classifiers used, or the specific combination technique used.

By identifying how to combine individual classifiers in the most effective way, our results have substantially improved the state-of-the-art in bug localization. We conclude that even in the face of unstructured text in bug reports and unstructured linguistic data in source code, IR models can effectively localize bugs and reduce maintenance efforts and costs for developers.

Since bug localization has only recently gained the attention of researchers, there are many exciting avenues to explore in future work. The most obvious avenue is the addition of classifier families to the combination framework presented in Section 5.2. Existing classifier families include those based on PageRank (Page et al., 1999; Revelle et al., 2010), formal concept analysis (Poshyvanyk and Marcus, 2007), dynamic analysis (Poshyvanyk et al., 2007), static analysis (McMillan et al., 2011), and BugScout, a variant of LDA (Nguyen et al., 2011). Additional IR models can be considered, such as BM25F (Robertson et al., 2004), and other variants of LDA, such as the Relational Topic Model (Chang and Blei, 2009). Recently, researchers have proposed query expansion techniques (Carpineto and Romano, 2012), which may be a useful preprocessing step to any IR-based classifiers. Finally, we have yet to fully investigate many possible combination techniques, such as variants of the Borda count and Reciprocal Rank Fusion (Cormack et al., 2009).

CHAPTER 6

Using Topic Evolution Models to Analyze Source Code Evolution

Studying concept evolution in source code can help developers and managers monitor the changes to source code at a conceptual level, as opposed to a file or method level. In this chapter, we propose a technique based on an advanced IR model, named the Hall topic evolution model, that uses the history of the source code to extract the evolution of concepts over time. We perform case studies on JHotDraw and jEdit to determine whether the the inferred conceptual evolutions are valid and useful for developers. We find that 87–89% of the inferred evolutions correspond well with actual code change activities by developers, meaning that software development teams can use these evolutions as a means to automatically monitor and analyze their source code evolution.

Publications based on this chapter: Thomas et al. (2010b, 2012a)

6.1 Motivation

CONTEMPORARY SOFTWARE DEVELOPMENT development consists of multiple developers working together. The developers often make changes to the source code in parallel and in rapid succession. Developers may work on disparate aspects of the source code at the same time, and the end result is that the software is evolving faster than any single person can easily comprehend, or even monitor.

Traditional software evolution monitoring techniques involve counting the number of lines of code in the source code over time, or counting the number of changes to a source code module, or counting the number of bugs fixed to date. While useful to some extent, these techniques do not answer the questions that developers often ask (Bradley and Murphy, 2011; de Alwis and Murphy, 2008; Fritz and Murphy, 2010): *What is being changed in the source code? What are the important concepts in the code, and which concepts are being actively developed right now? Who is responsible for which concepts?* As source code files are often not tagged into conceptual units, it is difficult to automatically monitor the evolution of software at this level.

Understanding how the use of a concept evolves in a software repository over time, however, could provide many benefits for project stakeholders (Linstead et al., 2008b). For example, stakeholders could monitor the *drift* of a concept, i.e., when the implementation of a concept in the source code gradually diverges from the original design (similar to architectural drift (Perry and Wolf, 1992)). Because of refactoring, re-engineering, maintenance and other development tasks, a concept that was once focused and modularized may become more scattered across the system over time, getting out-of-sync with the mental model that designers and architects have about the system. Automatically discovering and monitoring these concept drifts would be a useful technique for developers and project managers wishing to keep their system in good health. Concept evolutions could also be used by newcomer developers wishing to quickly understand the history of certain aspects of the system, such as when specific features were added or removed from the system. Additionally, topic evolutions could be used to monitor the day-to-day development tasks, answering questions such as "Who is working on what concepts?" and "What concepts changed since last week?"

Recently, researchers have applied topic evolutions models, such as LDA and the Hall model (Section 2.2), to the version history of a source code repository to uncover the evolutions of topics (Linstead et al., 2008b). These applications of topic evolution models all make one key assumption: that topic evolution models, when applied to source code histories, describe the changes made to the source code in way that is useful to a project stakeholder. But this assumption is not trivially validated: topic models were built for natural language corpora, which source code clearly is not. Further, changes to source code are often frequent and random, violating assumptions by many topic evolution models. Even if topic evolution models can be successfully applied to source code changes, it is not clear whether these discovered evolutions will represent the high-level concepts in which practitioners are interested. Our goal in this chapter, then, is to determine if topic evolution models provide an accurate account of the changes to source code.

6.2 Proposal

We propose to use the Hall topic evolution model (Section 2.2) to abstract the source code into high-level topics, which will act as representations of the conceptual units embedded in the source code. Further, we will use the evolutions of the topics to monitor how the source code concepts evolve over time. We hypothesize that the Hall topic evolution models can be applied to source code to uncover a set of underlying concepts, providing an approximate and inexpensive model of the evolution of concepts in a software system, which can help answer the questions that developers have. A preliminary study found evidence that supported this proposition (Linstead et al., 2008b), but more work is needed to fully understand and quantify how well topic evolution models can represent the actual source code evolution.

We will evaluate the effectiveness of the Hall model by performing manual analyses on the discovered topics on real-world software systems. Given a discovered topic and its evolution (i.e., whether the popularity of that topic is increasing or decreasing at any given time), We can manually examine the source code changes, commit log messages, and change notes of the system to determine if the discovered evolution aligns with reality. Doing this for all topic evolutions will give me a good sense to how well the topic models are able to represent source code evolution.

6.3 Case Studies

We conduct a detailed case study of two real world system, focusing on the following research questions.

RQ1 *How well do the discovered topic evolutions correspond to actual change activities in the*

	JHotDraw	jEdit
Purpose	Drawing framework	Text editor
Implementation language	Java	Java
License	Open source	Open source
Time period considered	Feb. 2001–Aug. 2010	Dec. 2000–Dec. 2004
Number of releases	13	12
Lines of code (thousands)	9.4–84	57–157
Number of source code documents	160–613	248–427
Number of committers	11	120

Table 6.1: Characteristics of our two systems under study, JHotDraw and jEdit.

source code?

We wish to determine the accuracy of topic evolution models. Given a set of change events in a discovered topic evolution, as well as a set of change activities to the source code, what is the correspondence?

RQ2 *What is the relationship between code change categories and topic evolution?*

For those evolutions that are meaningful, we wish to perform a descriptive study to determine the common relationships between evolutions and *code change categories* (i.e., bug fixes, feature additions, and refactorings).

RQ3 *What are the patterns of topic evolution?*

We wish to quantify and study how topics evolve, to gain insight into both the abstract notion of topic evolution as well as into the development processes of the studied systems.

Answering RQ1 allows us to evaluate the use of topic models for studying the evolution of source code, bringing us one step closer towards a robust software monitoring technique built upon topic evolution models (Section 6.4.2). Answering RQ2 and RQ3 allows us to better understand *why* and *how* topics evolve in source code, strengthening our understanding of the results of topic models and software evolution in general (Sections 6.4.3 and 6.4.4).

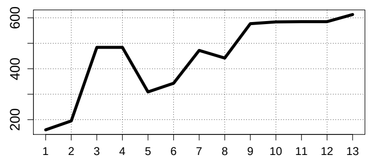
6.3.1 Systems Under Study

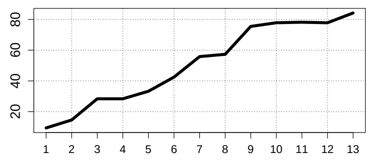
We address our research questions by performing in-depth case studies on the source code histories of two well-known software systems, JHotDraw and jEdit. JHotDraw is a medium-sized, open source, 2-D drawing framework developed in the Java programming language (Gamma, 2012). It was originally developed as an exercise of good program design and has become the de facto standard system for experiments and analysis in topic and concern mining (for example, by Robillard and Murphy (2007) and Binkley et al. (2006)). JHotDraw is a good choice for our purposes due to its good design practices and manageable size for manual analysis. We consider 13 release versions of JHotDraw (5.2.0–7.5.1). These versions were released over a nine year period and saw a growth of over 600% in the number of lines of code, several complete restructurings, and the addition of several new features (see Table 6.1 and Figures 6.1a–6.1c). During this time, 11 individual developers committed changes to the code base.

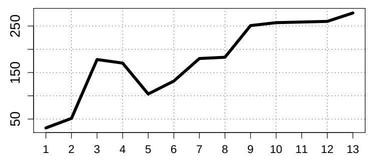
jEdit is a medium-sized, open source text editor written in the Java programming language (Pestov, 2012). jEdit focuses on providing rich features for developers, including syntax highlighting, macro scripting, and a comprehensive plug-in environment. jEdit is a good choice for our study because it is well organized, has extensive documentation, and has a manageable size for manual analysis. We consider 12 release versions of jEdit (3.0.0–4.2.0). The versions span a four year period where the code base grew by almost 200% (see Table 6.1 and Figures 6.1d–6.1f). During this time, 120 individual developers committed changes to the code base.

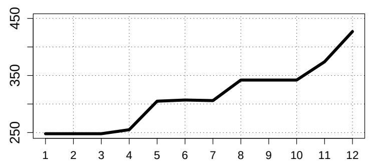
6.3.2 Data Preprocessing

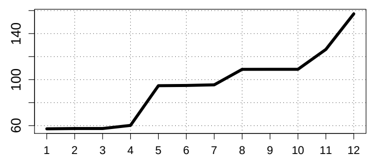
We preprocess the source code histories of each system by applying the typical source code preprocessing steps required by any information retrieval technique (Section 2.2). We split identifier names, remove stop words, stem, and prune the vocabulary (remove those that occur in more than 80% or less than 2% of the documents). For JHotDraw, the preprocessing resulted in a total of 2.3M words (comprised of 964 unique words) in 5,833 documents,


(a) Number of documents in JHotDraw.


(b) KSLOC in JHotDraw.


(c) Number of words (×1,000) in JHotDraw.


(d) Number of documents in jEdit.


(e) KSLOC in jEdit.

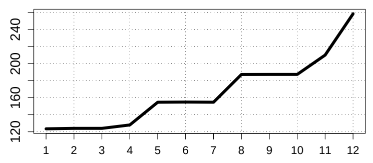

(f) Number of words (×1,000) in jEdit.

Figure 6.1: Data characteristics over time (version number) for JHotDraw and jEdit. In (c) and (f), we show the number of words remaining after the preprocessing steps have been performed.

an average of 394 words per document. For jEdit, the preprocessing resulted in a total of 1.9M words (816 unique) in 3,744 documents, an average of 507 words per document.

6.3.3 Topic Modeling Technique

For the LDA computation, we used MALLET version 2.0.6 (McCallum, 2012). MALLET is a highly scalable Java implementation of the Gibbs sampling algorithm. We ran for 10,000 sampling iterations, the first 1,000 of which were used for parameter optimization (Griffiths and Steyvers, 2004). We allowed MALLET to use hyper-optimization for the α and β input parameters, which are smoothing parameters for the model. Figure 6.2 provides a brief replication guide for our study.

In addition to discovering topics, MALLET also automatically discovers a two- or three-word label for each topic that helps describe the topic in a compact way (e.g., “mouse click”, “file format”). The label is based on commonly-occurring n-grams (i.e., co-located words) in the documents containing the topic. Although not all of the discovered labels are ideal (i.e., what a human would create) and sometimes produce double words (e.g., “elem elem”), we

1. Collect source code data. This can be performed by either checking out copies from the system's software configuration manager (e.g., SVN, CVS, Git) or by downloading source code snapshots from the system's webpage. Either way, the end result should be a collection of source code snapshots corresponding to the versions of interest (in our case, major releases).
2. Preprocess the data. Isolate source code identifiers and comments. Split the words based on common naming schemes. Convert all letters to lower case. Remove stop words. Stem each word. Remove overly common words (those that appear in more than 80% of the documents) and rare words (less than 2%).
3. Transform the data into MALLET format. If *input-dir* is the name of the top-level directory containing the preprocessed source code documents, and `${MALLET-BIN}` is the path to the MALLET executable, then the command

   ```
   ${MALLET-BIN} import-dir --input input-dir --output data.mallet --keep-sequence
   ```

   will create the output file *data.mallet*.
4. Discover the topics. Run the command

   ```
   ${MALLET-BIN} train-topics \
   --input data.mallet \
   --num-topics 45 \
   --num-iterations 10000 \
   --optimize-burn-in 1000 \
   --optimize-interval 100 \
   --output-doc-topics allfiles.txt \
   --output-topic-keys topics.dat \
   --topic-word-weights-file wordweights.dat \
   --word-topic-counts-file topiccounts.dat \
   --xml-topic-phrase-report topic-phrases.xml
   ```

   substituting the number of topics, iterations, etc. as desired.
5. Analyze the data. The output file *wordweights.dat* contains the unnormalized, unsorted word weights for each topic. The output file *allfiles.txt* contains the resulting topic memberships for each input document. The output file *topic-phrases.xml* contains the topic labels for each topics. Using these documents, compute topic metrics according to Equations 6.1, 6.2, and 6.3 on each slice of time (i.e., all documents at each version). Perform additional analyses and visualizations as desired. In our case, we relied on the R statistical environment (Ihaka and Gentleman, 1996).

Figure 6.2: Replication guide for our study. For each system under test, perform these steps.

find the labels to be useful. For the remainder of this chapter, when we present a label for a topic, we are presenting the label automatically discovered by MALLET.

Choosing the Number of Topics (K)

For any given corpus, there is no provably optimal choice for K (Wallach et al., 2009). The choice is a trade-off between coarser topics (smaller K) and finer-grained topics (larger K). Setting K to extremely small values results in topics that contain multiple concepts (imagine only a single topic, which will contain all of the concepts in the corpus!), while setting K to extremely large values results in topics that are too fine to be meaningful and only reveal the idiosyncrasies of the data. Our goal in this study is to discover topics of medium granularity, so we seek a non-extreme value for K.

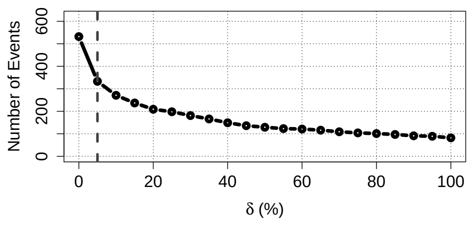
Previous work has used 45 topics for JHotDraw, arguing that 45 discovers topics of medium granularity (Baldi et al., 2008; Thomas et al., 2010b). To be comparable to these studies, we also set K to 45 for both JHotDraw and jEdit. However, as we show in Section 6.4.1, the discovered topics are stable (i.e., not particularly sensitive to the exact value of K) in at least the range of 30–60.

In the Hall model, each version of the source code will not necessarily contain all K topics. Since all of the versions are applied to the LDA model at once, and LDA is looking for a total of K topics, some versions may contain fewer than K topics. Imagine, for example, that new XML functionality is inserted into the code at version V_2, and that LDA discovers a topic representing XML concepts. Since no code dealt with XML in version V_1, this topic will not appear in V_1. We say that the topic is *born* at V_2, and that V_1 has at most K-1 topics. Similarly, a topic *dies* if all source code related to the topic is removed at some version. Thus, the number of topics K represents the total number of topics that exist across all time in the corpus, not necessarily at each point in time.

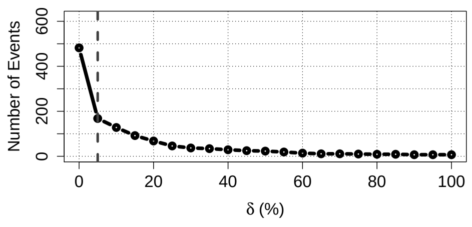
Choosing the Change Threshold (δ)

Due to the probabilistic nature of the LDA model, the same word in a given document may be assigned to different topics in different versions of the document. For example, the word "button" in a document d could be assigned to a "GUI"-related topic in version V_1, and then assigned to a "dialog box"-related topic in version V_2. Because of this small, unlikely, and statistically uninteresting change, both topics will experience a small change in their metric values (i.e., the weight metric will increase or decrease by 1).

To account for these small, probabilistic changes in topics over time, we introduce the δ threshold for determining if a metric value has changed significantly from one version to the next. This threshold, used in Equation 6.4 to classify changes as spikes, drops, or stays,



(a) JHotDraw.



(b) jEdit.

Figure 6.3: Number of detected events (spikes and drops) for various δ thresholds, according to Equation 6.4. The vertical dashed lines represents the "knee" in the curve, which we use as our cut-off value.

will help weed out uninteresting changes while preserving the interesting ones. Our goal is to set the threshold to a value that will achieve this balance.

Figure 6.3 shows the number of events (i.e., spikes and drops) as a function of δ. When δ is zero, there are many events; almost one for every topic at every version. Many of these are obviously not interesting, as the metric's value only changes by less than 0.1%. In this chapter, we are interested in studying major topic events, so we choose a δ value of 5% for both systems, as this appears to weed out many uninteresting events without weeding out too many major events (i.e., near the "knee" of the curves in Figure 6.3). We note that different thresholds may be appropriate for other systems and other tasks, depending on the desired sensitivity to change.

6.3.4 Topic Evolution Metrics

We measure how a topic changes over time by computing metrics on the topic at each point in time and comparing the values. In this section we describe three metrics that can be used to characterize topic evolutions, although many more exist.

The ***assignment*** of a topic is the sum of the topic memberships of all documents in that topic, which gives an indication of the total presence of the topic throughout the code (Baldi et al., 2008). A higher topic assignment means that a larger portion of the code is related to the topic. We define the assignment of topic z_k at version V_j as

$$\mathrm{A}(z_k, V_j) = \sum_{i=1}^{|V_j|} \theta_{d_{ij}}[k]. \tag{6.1}$$

The ***weight*** of a topic is similar to its assignment, but in addition considers the length of each document. The weight metric exactly captures how many words in the entire corpus were generated by a topic, whereas the assignment metric captures the portion of documents that were generated by a topic, and therefore can be skewed by small documents. We define the weight of topic z_k at version V_j by

$$\mathrm{W}(z_k, V_j) = \sum_{i=1}^{|V_j|} \theta_{d_{ij}}[k] * |d_{ij}|. \tag{6.2}$$

The ***scattering*** of a topic is the normalized entropy of that topic over all documents (Baldi et al., 2008). Entropy is a common metric used in information theory to determine how uncertain, or spread out, a distribution is (Shannon, 2001); we normalize it by the number of documents to account for differing numbers of documents in each version. A topic with a high entropy value will be more spread throughout the system than a topic with a low entropy value. We define the scattering of topic z_k at version V_j by

$$\mathrm{S}(z_k, V_j) = \frac{1}{|V_j|} * \left[- \sum_{i}^{|V_j|} \theta_{d_{ij}}[k] * \log(\theta_{d_{ij}}[k]) \right]. \tag{6.3}$$

Other topic metrics, which we only briefly describe, are: the ***tangle*** of a topic, which indicates how many other topics a given topic is usually co-located with in a document; the α-***support*** of a topic, which indicates the number of documents that have a membership of α or higher in the topic; the ***turnover*** of a topic, which captures the number of new documents matching the topic at a given version, compared to the previous version; and the ***similarity*** between two topics, which describes how similar the word distributions are between the two topics.

Finally, the ***evolution*** E of a metric m of a topic z_k is a time-indexed vector of metric values for that topic: $E(z_k, m) = [m(z_k, V_1), m(z_k, V_2), ..., m(z_k, V_v)]$.

We define a ***change event*** in a topic evolution as an increase (*spike*), decrease (*drop*), or no change (*stay*) in a metric value between successive versions. We classify a change event as a spike or a drop if there is at least a δ% increase or decrease in metric value compared to the previous version, and as a stay otherwise. Formally, for a metric m of topic z_k at version V_j, the change $c = (m(z_k, V_j) - m(z_k, V_{j-1}))/(m(z_k, V_{j-1}))$ is classified as

$$\text{Event}(m, z_k, V_j) = \begin{cases} \text{spike} & \text{if } c \geq \delta, \text{or if } m(z_k, V_{j-1}) = 0 \\ & \quad\quad \text{and } m(z_k, V_j) > 0; \\ \text{drop} & \text{if } c \leq -\delta; \\ \text{stay} & \text{otherwise.} \end{cases} \tag{6.4}$$

In later sections, we will use the notion of change events to characterize and validate topic evolutions.

6.4 Results and Discussion

We first examine the discovered topics in both a qualitative and quantitative fashion in order to obtain a intuition about the topics: do they make sense? What do they look like? Are they stable? We then present the results of our three research questions in turn.

6.4.1 Examining the Discovered Topics

Before we begin our analyses of the discovered topic evolutions and their ability to describe actual changes in source code, we familiarize ourselves with the nature of the topics and evolutions discovered by our technique. To do so, we examine a few selected topics, consider various visualizations of their evolutions, and examine the stability of the topics.

Label	Top words	Top 3 related docs
JHotDraw		
“bezier path”	*path bezier node index coord mask point geom*	`BezierPath.java` (0.92), `GrowStroke.java` (0.59), `BezierPointLocator.java` (0.56)
“elem elem”	*attribut elem param child full attr return type*	`IXMLElement.java` (1.00), `XMLAttribute.java` (1.00), `XMLElement.java` (1.00)
“input stream”	*stream read end encod length offset line charact*	`PIReader.java` (0.54), `ContentReader.java` (0.52), `CDATAReader.ja va` (0.50)
jEdit		
“abbrev abbrev”	*abbrev mode expand line global expans set*	`Abbrevs.java` (1.00), `AbbrevsOptionPane.java` (0.44), `AddAbbre vDialog.java` (0.29)
“gnu regexp”	*index input match token retoken rematch current*	`RE.java` (1.00), `RETokenRepeated.java` (1.00), `REMatchEnumer ation.java` (1.00)
“plugin manag”	*plugin jar instal string edit version list return*	`PluginList.java` (1.00), `JARClassLoader.java` (0.53), `PluginList Handler.java` (0.51)

Table 6.2: Example topics from JHotDraw and jEdit. The labels are automatically generated by MALLET. The top words define the topic, and we display the documents with the three highest membership values (shown in parenthesis) for each topic.

The Topics Themselves

The full listing of the topics discovered by our technique is given in Figures 6.4 and 6.5 for JHotDraw and jEdit, respectively. We find topics that span a range of concepts, including “mous motion”, “affin transform”, “zoom factor”, and “undoable edit” topics in JHotDraw and “xml pars”, “tool bar”, “bin dir”, “font”, “hyper search”, and “gnu regexp” topics in jEdit. Table 6.2 shows selected topics, their top (stemmed) words, and their top three matching documents for each system. The groupings of top words seem to make sense to a human reader, based on their semantic similarities. Anyone who is familiar with XML, for example, will agree that the words “attribute”, “element”, and “child” naturally go together in this context. Further, the top documents seem to be a natural fit with both the given topic and the other topic documents. Just as other communities have found topics to make sense and be useful for their purposes, we find source code topics to be coherent and useful.

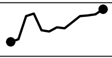
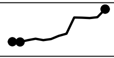
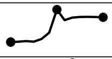
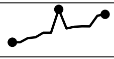
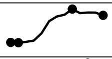
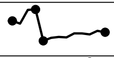
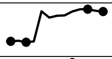
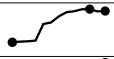
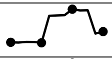
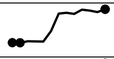
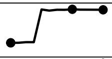
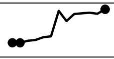
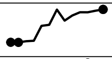
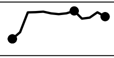
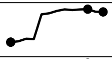
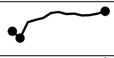
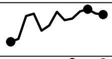
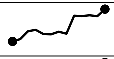
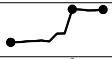
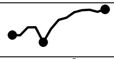
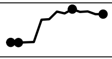
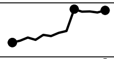
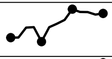
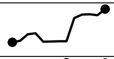
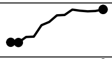
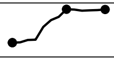
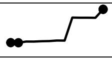
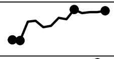
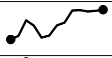
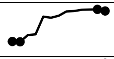
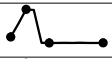
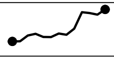
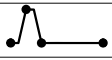
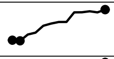
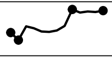
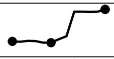
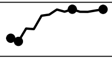
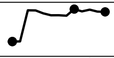
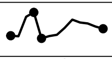
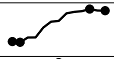
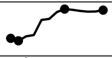
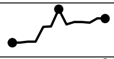
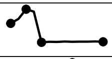
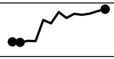
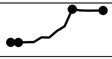
	Label	Top Words	Trend		Label	Top Words	Trend
1.	method invok	*method except object invok resourc param target obj*		24.	zoom factor	*grid bag constraint awt swing javax set add*	
2.	input stream	*stream read end encod length offset line charact*		25.	menu item	*action menu add window view item set jmenu*	
3.	buf append	*elem attribut write buf append creat figur ixmlel*		26.	creation tool	*tool figur creat mous draw creation view prototyp*	
4.	open elem	*elem attribut current add object write docum ioexcept*		27.	bezier path	*path bezier node index coord mask point geom*	
5.	stop color	*color gradient space paint focu fraction stop arrai*		28.	attribut kei object	*elem attribut read str style inherit ioexcept number*	
6.	reader reader	*reader param system elem except entiti line data*		29.	scroll pane	*set layout add pane swing editor line javax*	
7.	properti chang	*properti listen chang enabl event action handler updat*		30.	chang listen	*listen event figur chang remov draw list invalid*	
8.	attribut kei	*button editor add attribut action label tool draw*		31.	connect figur	*connect figur connector start end target point draw*	
9.	suit add	*flavor suit transfer data drag focu compon clipboard*		32.	intern frame	*border frame bar pane desktop set compon window*	
10.	attribut kei	*color stroke kei attribut handl bound fill transform*		33.	storag format	*format file draw input output filter extens storag*	
11.	undoabl edit	*action edit undo label redo util bundl chang*		34.	code code	*code point pointd param curv digit bound max*	
12.	buffer imag	*imag buffer draw render width height hint set*		35.	tree model	*font list node famili collect select tree path*	
13.	attribut kei	*attribut kei set figur object map entri draw*		36.	affin transform	*transform rectangl figur width draw stroke handl clone*	
14.	color chooser	*color compon slider index icon model space system*		37.	text holder	*text font figur holder edit area size tab*	
15.	draw view	*view draw editor select constrain figur activ set*		38.	figur figur	*figur draw child composit children add list remov*	
16.	undo activ	*public command return undo activ figur set undoabl*		39.	start time	*link code return param task target method figur*	
17.	junit doclet	*test junit method doclet begin end javadoc except*		40.	inset inset	*inset layout width height left bound size top*	
18.	mous motion	*handl select figur mous view draw event evt*		41.	field set	*field gbc set button grid label opac bag*	
19.	elem elem	*attribut elem param child full attr return type*		42.	content produc	*draw set paramet applet content url input produc*	
20.	poli line	*point line angl width rectangl pointd height corner*		43.	stroke dash	*stroke color map put fill width dash text*	
21.	locat locat	*handl locat owner bound transform pointd figur point*		44.	recent file	*view file applic app action project set chooser*	
22.	displai box	*figur public box displai draw decor return read*		45.	option pane	*code sheet pane messag listen option dialog compon*	
23.	icon icon	*icon descriptor bean properti event gen method color*					

Figure 6.4: Topic trendlines for JHotDraw. For each topic, we show the automatically-generated topic label, the top eight words in the topic, and the trend line of topic assignment evolution.

Visualizing the Evolutions

To better understand the topic evolutions, we experiment with two visualization techniques, each with its own advantages and disadvantages.

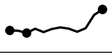
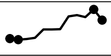
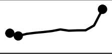
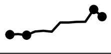
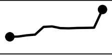
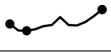
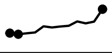
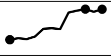
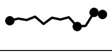
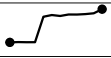
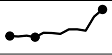
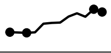
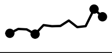
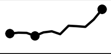
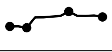
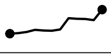
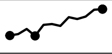
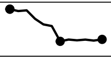
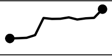
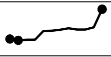
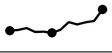
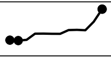
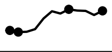
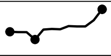
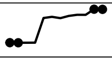
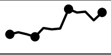
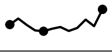
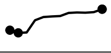
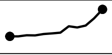
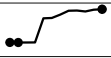
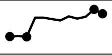
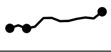
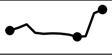
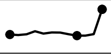
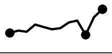
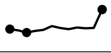
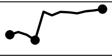
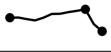
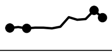
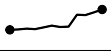
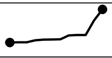
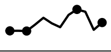
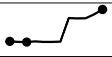
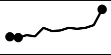
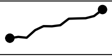
Label	Top Words	Trend	Label	Top Words	Trend
1. tree model	*tree event node path model select result mous*		24. font font	*font print color style size text width privat*	
2. out write	*log url error privat return elem file els*		25. parser rule	*rule token context line set last parser keyword*	
3. plugin manag	*plugin jar instal string edit version list return*		26. token begin token	*activ move liter state kind start token check*	
4. action listen	*action list add con button select label set*		27. text area	*text area regist edit select caret set public*	
5. hyper search	*search replac set view buffer matcher find edit*		28. flag invalid	*offset size defin flag type stack format endif*	
6. gjt jedit	*view macro action edit record code repeat buffer*		29. bin dir	*instal size dir system public progress oper directori*	
7. pars except	*token node scope except consum close pars scan*		30. path path	*file path directori browser view session entri filter*	
8. set text	*add set box border action layout panel listen*		31. gjt jedit	*public return param edit compon pre comp gjt*	
9. line line	*line select caret start end offset buffer int*		32. bug fix revision	*revision syntax set fix bit bug updat improv*	
10. buf append	*append path buf return length str compar case*		33. bsh java	*method type return bsh variabl object interpret space*	
11. xml pars	*read buffer entiti attribut except encod elem type*		34. model model	*model tabl row col entri public return size*	
12. text area	*line color text highlight area gutter set font*		35. menu item	*menu item action add histori set mous popup*	
13. compil header	*file log script launcher path return modul server*		36. dockabl window	*window dockabl position width entri contain height top*	
14. edit properti	*line marker edit return buffer properti set file*		37. gnu regexp	*index input match token retoken rematch current mymatch*	
15. option pane	*option edit properti pane set box select add*		38. instal instal	*str instal kei path data reg error log*	
16. abbrev abbrev	*abbrev mode expand line global expans set hashtabl*		39. simpl node	*node eval type interpret jjt simpl callstack child*	
17. icon icon	*icon menu properti color return edit code style*		40. kei bind	*kei bind event shortcut return evt code case*	
18. view view	*buffer view file edit log properti set return*		41. eval error	*object type error primit eval field obj interpret*	
19. tool bar	*pane edit bar tool text buffer split area*		42. color color	*color style tabl option set add border background*	
20. work thread	*thread request progress work run log pool abort*		43. buffer buffer	*buffer view edit set properti updat jedit pane*	
21. input stream	*stream read code buffer length size write input*		44. edit properti	*mode properti edit set tab line indent size*	
22. line line	*line offset fold info method buffer start count*		45. sourc file	*error eval except interpret messag file sourc target*	
23. word sep	*index word text length start charact return end*				

Figure 6.5: Topic trendlines for jEdit. For each topic, we show the automatically-generated topic label, the top eight words in the topic, and the trend line of topic assignment evolution.

Line charts. Figure 6.11 shows a traditional line chart view of the weight evolutions of four selected topics. The layout of a line chart is just as one would expect: the x-axis shows time and the y-axis shows the metric value at each point in time. Line charts are helpful for closely inspecting an individual topic evolution: to visualize the periods of spikes and drops and quickly reveal periods of activity and interest in the topic. The line chart is intuitive to understand but requires considerable space, especially when visualizing hundreds of topics.

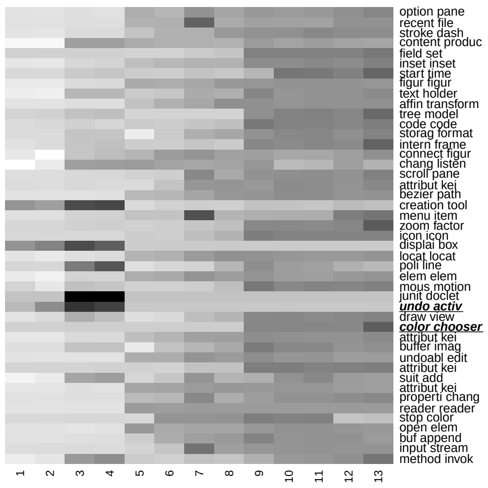
For a complete listing of the discovered topics, Figures 6.4 and 6.5 show all of the topics discovered for both systems, along with a simplified line chart to save space and give a quick impression of the general behavior of the topic evolution. The simplified line charts include up to four thick circles on the trend lines, indicating the first and last values on the line as well as the minimum and maximum values.

Heatmaps. Figure 6.6 shows a heatmap view of the assignment evolutions for both JHotDraw and jEdit. The color of each cell represents the assignment value for a topic at a particular time, where darker colors indicate higher assignment values. This visualization allows us to quickly and compactly compare and contrast the trends exhibited by the various topics. For example, Figure 6.6a indicates that some topics in JHotDraw (e.g., "undo activ", bolded in the figure) become increasingly active during the initial versions of the system, but then die at later versions. Other topics do not see any activity until later versions, such as the "color chooser" topic (bolded in the figure).

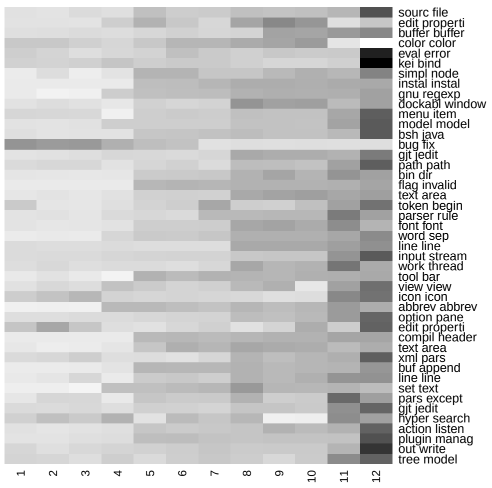
An interesting quality of the heatmap visualization is the ability to visually detect different phases of development, or releases with many changes to many topics. For example, there is a *visual wall* effect in Figure 6.6a between versions V_4 and V_5, evidenced by a large color change in several topics. This suggests that there was a large development effort that concurrently affected multiple concepts. Indeed, version V_5 was a major release that experienced a multitude of refactorings and changes to the core framework of the system (cf. large drop in Figure 6.1a).

Examining Topic Stability

The topics discovered from statistical topic models are a result of the input data, the input parameters, and the statistical sampling methods. *Topic stability* refers to how stable the



(a) JHotDraw.



(b) jEdit.

Figure 6.6: Heatmap views of the 45 topic evolutions discovered from the JHotDraw and jEdit source code. Darker cells indicate a higher assignment metric value. Rows are topics (shown with their automatically-generated topic labels) while columns are versions.

discovered topics are across these factors (Steyvers and Griffiths, 2007). For example, it would be undesirable if changing K from 45 to 46 would cause a completely new, unrelated set of topics to be discovered and all of our results to be affected. Likewise, sampling for a few more or less iterations ideally should not have a large impact on the topics. In this section, we investigate topic stability.

Topic stability is determined by measuring the Kullback-Leibler (KL) distance (Cover and Thomas, 2006) between pairs of topics in different runs (or instantiations) of the topic model (Steyvers and Griffiths, 2007). The KL distance between two topics is given by

$$\mathrm{KL}(z_1, z_2) = \frac{1}{2}\sum_{i=1}^{N} \phi_{z_1}[i] \log_2 \frac{\phi_{z_1}[i]}{\phi_{z_2}[i]} + \frac{1}{2}\sum_{i=1}^{N} \phi_{z_2}[i] \log_2 \frac{\phi_{z_2}[i]}{\phi_{z_1}[i]}, \tag{6.5}$$

taking care to align the word orders in the word vectors. By computing the KL distance matrix for pairs of topics in each run, reordering the matrix in a greedy fashion so that the most similar topics are on the diagonal, and viewing the output as a heatmap, we can get a sense for how similar the topics are in the two runs. If the topics are indeed similar, that is, each topic in run 1 has a corresponding topic in run 2, then we can be confident that the topics are stable in the two runs. For example, Figure 6.7a shows a similarity matrix of JHotDraw compared against itself, which will be the most stable case possible. A dark line down the diagonal indicates that, indeed, each topic in run 1 is very similar to a topic in run 2 (which are the same in this case). Figure 6.7b, on the other hand, shows the similarity matrix of the topics discovered from JHotDraw measured against those of jEdit, which we do not expect to be very similar. Less than 10 of the topics show high similarity (e.g., both systems have an "input stream" topic), but most topics are more dissimilar and appear in the figure as random noise.

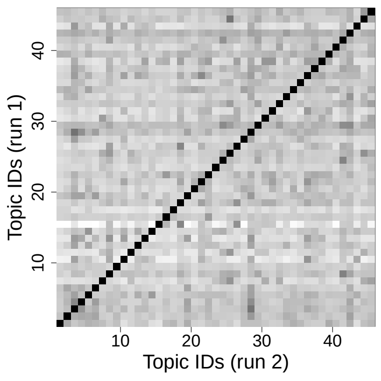
Figure 6.8 shows the results of topic stability for JHotDraw and jEdit for different numbers of topics (K = 30, 45, and 60), different numbers of sampling iterations (I=9,000 and 10,000), and different amounts of input data (all versions at once with K=45 and only the latest version with K=42). Overall, we find that the topics in both JHotDraw and jEdit are moderately to very stable, as indicated by the presence of dark diagonals that span most topics.

The discovered topics are moderately stable to increases in K or I.

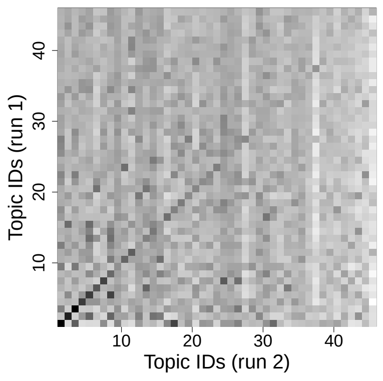
6.4.2 Manual Analysis of Validity (RQ1)

Our first research question examines the meaningfulness of topic evolution models applied to source code: do the discovered topics evolve because of actual change activity in the source code? If so, then topic evolution models may be an appropriate tool for understanding the change history of source code. However, if the discovered topic evolutions do not correspond well with the change activity in the source code, then topic evolutions are not a good choice.

We determine the meaningfulness of the discovered topic evolutions by analyzing in detail the change events of the evolutions. In particular, we perform a manual analysis of



(a) JHotDraw vs. itself.



(b) JHotDraw vs. jEdit.

Figure 6.7: Example topic similarity matrices. (a) JHotDraw is compared against itself, which yields the most stable topics possible, indicated by the black diagonal. (b) JHotDraw is compared against a completely different system, jEdit, which yields completely unstable topics, indicated by a lack of a strong diagonal.

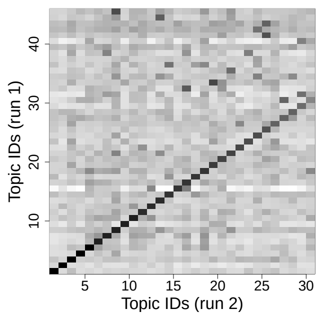
a random subset of the discovered change events of the evolutions. We selected enough random samples to yield a 90% confidence level with a margin of error of 10% (Scheaffer and McClave, 1994), Our sample size, n, for each type of event of each system is calculated as

$$n = \frac{t}{1 + \frac{t-1}{N}}$$

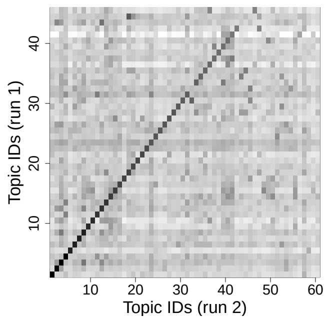
where N is the total number of events of this type, $t = (Z^2 p(1-p))/B^2$, $B = 0.10$ and $1 - \alpha = 0.90$, so $Z = z_{\alpha/2} = z_{0.05} = 1.645$ (Scheaffer and McClave, 1994). Since we have no prior knowledge on the probability p of each event, we set p to 0.5. To account for differing numbers of spikes, drops, and stays, we sample each event independently. Thus, this calculation totals 132 out of 540 events for JHotDraw and 113 out of 495 events for jEdit.

In our study, there are four possible outcomes for each event:

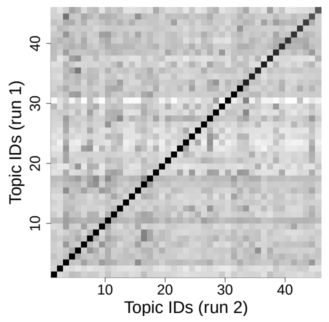
– *True positive.* A spike or drop event is detected in a topic's evolution, and the source code exhibits a change relating to that topic.



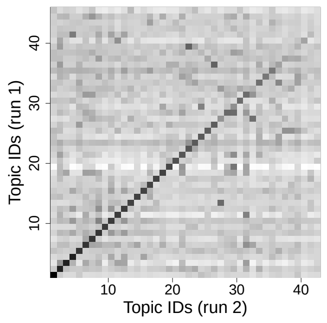
(a) JHotDraw, K=45 vs. 30.



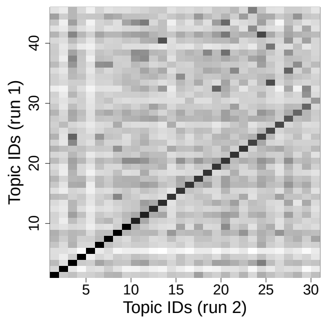
(b) JHotDraw, K=45 vs. 60.



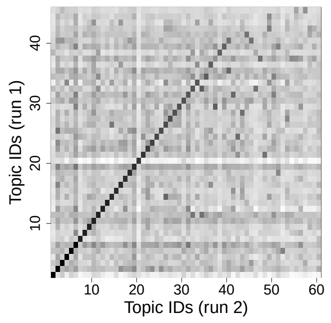
(c) JHotDraw, I=10K vs. 9K.



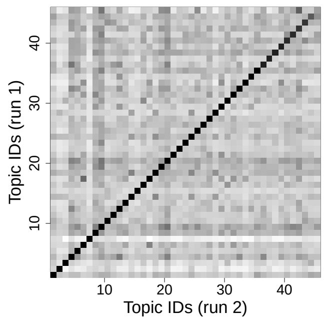
(d) JHotDraw, K=42 (latest version) vs. 45 (all versions).



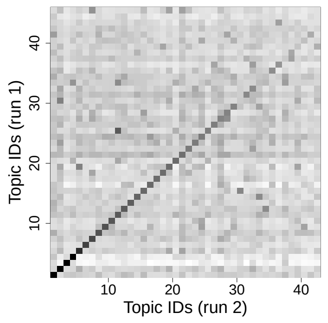
(e) jEdit, K=45 vs. 30.



(f) jEdit, K=45 vs. 60.



(g) jEdit, I=10K vs. 9K.



(h) jEdit, K=42 (latest version) vs. 45 (all versions).

Figure 6.8: Topic similarity matrices indicating the stability of the topic models. Each cell is the KL distance between pairs of topics. Darker cells mean less distance (more similarity). A column having one dark cell indicates that the topic in run 2 is very similar to exactly one topic in run 1.

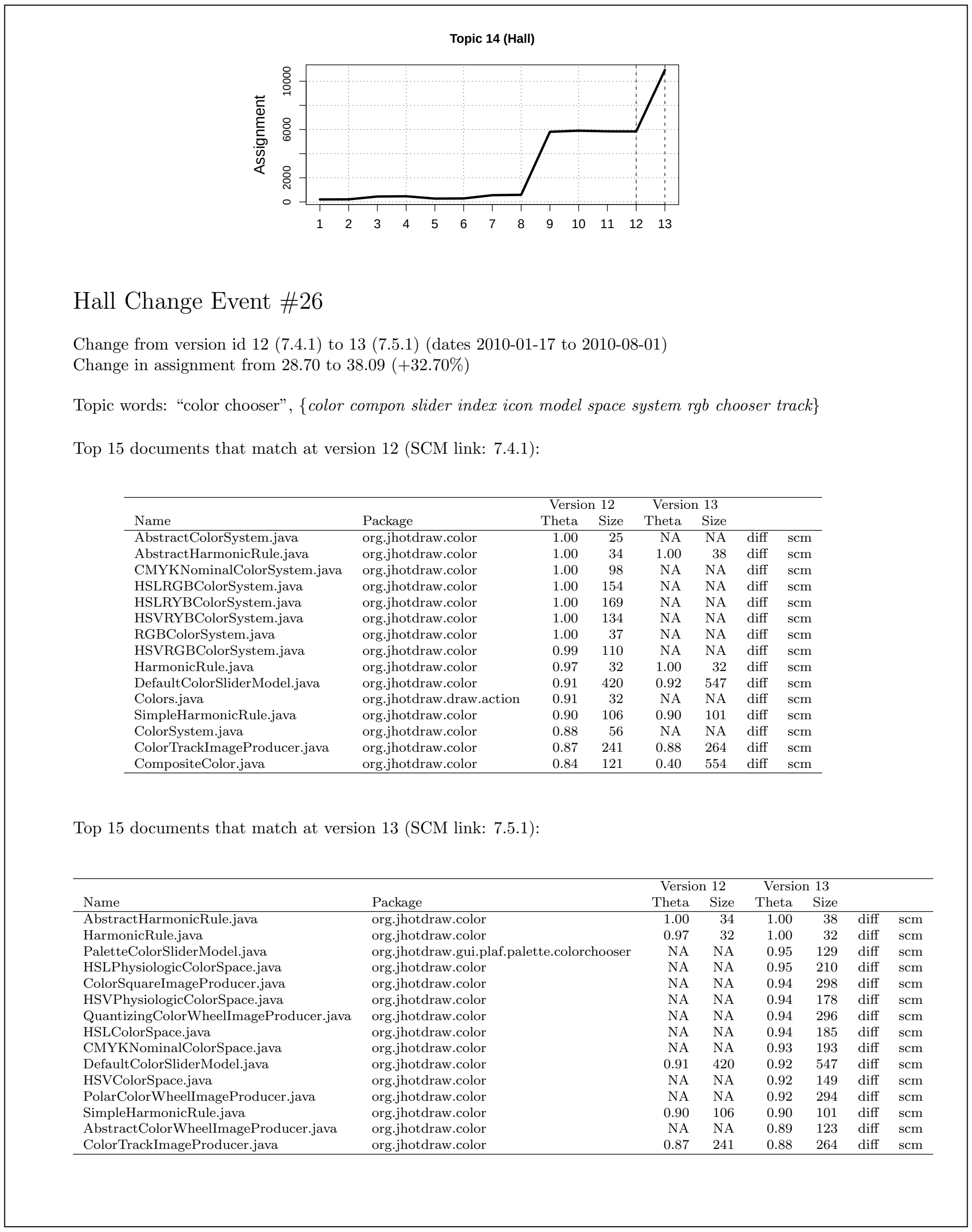
– *False positive.* A spike or drop event is detected in a topic's evolution, but the source code does not exhibit a change relating to that topic.
– *True negative.* A stay event is detected in a topic's evolution, and the source code does not exhibit a change relating to that topic.
– *False negative.* A stay event is detected in a topic's evolution, but the source code exhibits a change relating to that topic.

To aid us in our manual analysis, we developed a tool that randomly selects a given number of spike, drop, and stay events in the evolution of the weight metric. We choose the weight metric because it most accurately depicts the number of lines added and removed for a topic, making manual analysis more intuitive. For each event selected (between two versions V_j and V_{j+1}), the tool presents the following information.

– The topic label and the 10 most probable words.
– A plot showing the weight evolution, with the selected change event highlighted.
– A description of the change event (i.e., metric values before and after the event).
– A list of the top 15 documents matching the topic at version V_j. For each document, the tool lists the document's membership for this topic, the size of the document, a link to the diff between versions V_j and V_{j+1} of the document, and a link to the SCM entry for the document.
– A list of the top 15 documents matching the topic at version V_{j+1} (along with the metrics and links just mentioned).

Figure 6.9 shows an excerpt from the tool. The excerpt shows a single change event, in this case a spike between versions V_{12} and V_{13} of JHotDraw. In the tool, the "diff" and "scm" words are clickable, taking the user to a web page containing the diff report of the document between the two versions or the SCM entry for that documents, respectively. In addition, when the user clicks on the version numbers (e.g., 7.5.1), the release notes for that version are displayed.

Armed with this information, we examined the system documentation (including release notes, commit logs, and source code comments), looking for evidence that supported



Hall Change Event #26

Change from version id 12 (7.4.1) to 13 (7.5.1) (dates 2010-01-17 to 2010-08-01)
Change in assignment from 28.70 to 38.09 (+32.70%)

Topic words: “color chooser”, {*color compon slider index icon model space system rgb chooser track*}

Top 15 documents that match at version 12 (SCM link: 7.4.1):

		Version 12		Version 13			
Name	Package	Theta	Size	Theta	Size		
AbstractColorSystem.java	org.jhotdraw.color	1.00	25	NA	NA	diff	scm
AbstractHarmonicRule.java	org.jhotdraw.color	1.00	34	1.00	38	diff	scm
CMYKNominalColorSystem.java	org.jhotdraw.color	1.00	98	NA	NA	diff	scm
HSLRGBColorSystem.java	org.jhotdraw.color	1.00	154	NA	NA	diff	scm
HSLRYBColorSystem.java	org.jhotdraw.color	1.00	169	NA	NA	diff	scm
HSVRYBColorSystem.java	org.jhotdraw.color	1.00	134	NA	NA	diff	scm
RGBColorSystem.java	org.jhotdraw.color	1.00	37	NA	NA	diff	scm
HSVRGBColorSystem.java	org.jhotdraw.color	0.99	110	NA	NA	diff	scm
HarmonicRule.java	org.jhotdraw.color	0.97	32	1.00	32	diff	scm
DefaultColorSliderModel.java	org.jhotdraw.color	0.91	420	0.92	547	diff	scm
Colors.java	org.jhotdraw.draw.action	0.91	32	NA	NA	diff	scm
SimpleHarmonicRule.java	org.jhotdraw.color	0.90	106	0.90	101	diff	scm
ColorSystem.java	org.jhotdraw.color	0.88	56	NA	NA	diff	scm
ColorTrackImageProducer.java	org.jhotdraw.color	0.87	241	0.88	264	diff	scm
CompositeColor.java	org.jhotdraw.color	0.84	121	0.40	554	diff	scm

Top 15 documents that match at version 13 (SCM link: 7.5.1):

		Version 12		Version 13			
Name	Package	Theta	Size	Theta	Size		
AbstractHarmonicRule.java	org.jhotdraw.color	1.00	34	1.00	38	diff	scm
HarmonicRule.java	org.jhotdraw.color	0.97	32	1.00	32	diff	scm
PaletteColorSliderModel.java	org.jhotdraw.gui.plaf.palette.colorchooser	NA	NA	0.95	129	diff	scm
HSLPhysiologicColorSpace.java	org.jhotdraw.color	NA	NA	0.95	210	diff	scm
ColorSquareImageProducer.java	org.jhotdraw.color	NA	NA	0.94	298	diff	scm
HSVPhysiologicColorSpace.java	org.jhotdraw.color	NA	NA	0.94	178	diff	scm
QuantizingColorWheelImageProducer.java	org.jhotdraw.color	NA	NA	0.94	296	diff	scm
HSLColorSpace.java	org.jhotdraw.color	NA	NA	0.94	185	diff	scm
CMYKNominalColorSpace.java	org.jhotdraw.color	NA	NA	0.93	193	diff	scm
DefaultColorSliderModel.java	org.jhotdraw.color	0.91	420	0.92	547	diff	scm
HSVColorSpace.java	org.jhotdraw.color	NA	NA	0.92	149	diff	scm
PolarColorWheelImageProducer.java	org.jhotdraw.color	NA	NA	0.92	294	diff	scm
SimpleHarmonicRule.java	org.jhotdraw.color	0.90	106	0.90	101	diff	scm
AbstractColorWheelImageProducer.java	org.jhotdraw.color	NA	NA	0.89	123	diff	scm
ColorTrackImageProducer.java	org.jhotdraw.color	0.87	241	0.88	264	diff	scm

Figure 6.9: An excerpt from our manual analysis tool.

	Spikes		Drops		Stays		Total	
	Sample size	Valid (%)	Sample size	Valid (%)	Sample size	Valid (%)	Sample size	Valid (%)
JHotDraw	53/251	87±10	26/42	96±10	53/247	89±10	132/540	89±10
jEdit	47/150	83±10	10/11	80±10	56/334	91±10	113/495	87±10

Table 6.3: Results of the manual analysis (RQ1) for JHotDraw and jEdit. Valid spikes and drops correspond to true positives, whereas valid stays correspond to true negatives.

each change event. If the change event clearly corresponded to an actual change activity in the source code, then we classified the event as valid. Examples of clear correspondence for spikes and drops include: a direct mention of the topic in the release notes; multiple new documents being added to or deleted from the system that match the topic; and multiple matching documents being heavily modified. For stay events, we looked for a lack of change, including the topic not being mentioned in the release notes, related documents remaining unchanged, and no new related documents being added or deleted. In any case, if we found no such correspondence, then we classified the event as invalid.

Results

Table 6.3 shows our results. After performing the manual analysis of the selected change events, we found that almost all (89% and 87% for JHotDraw and jEdit, respectively) of the randomly selected events correspond to actual change activities in the source code (i.e., true positives and true negatives), backed by at least one source of system documentation. In these cases, there was a clear correspondence between the topic changes and the changes in the source code.

We found that the change events that were not backed by documentation and did not correspond to change activity in source code (i.e., false positives) were largely caused by one of the following issues.

Noisy membership changes. As LDA is a probabilistic model, it is possible that a given document will be assigned slightly different topic memberships on different executions of the model. In the Hall model, two successive versions of a document, although exactly the same in textual content, might be assigned slightly different topic memberships. Further, if this happens for several documents that all match a given topic, then the topic's metrics will change, even though the source code never did (see Figure 6.10).

Confounded topics. Ideally, LDA will discover topics that perfectly represent single real-world concepts. However, due to noise in the data, suboptimal parameter choices, and the sampling techniques of the inference algorithm, LDA sometimes discovers topics in which two or more logical concepts coexist—the topic is confounded with multiple logical concepts. Furthermore, these concepts do not necessary have an equal representation in the topic. In these cases, the evolutions for a topic can appear spurious and incorrect, due to a lighter concept receiving a legitimate spike or drop, while the larger concept should not experience any, or vice versa. An example of this is the "view" topic in jEdit. This topic contains the words "view", "buffer", and "edit", clearly representing the concept of different editor views of a file (i.e., buffer). This topic also contains the words "properti" and "set", representing the concept of managing properties (e.g., global properties or plugin properties). This single topic thus represents two concepts, a result of these concepts frequently being co-located in the source code. As a result, at versions when the "property" concept is changed, the "view" topic will exhibit a spike or drop, even though no code related to the "editor views" concept was actually changed.

We found that the majority (89–91%) of stay events were valid (true negatives). Many topics indeed exhibited periods of inactivity: none of their related documents were changed, no new documents were added that matched the topic, and there was no mention of the topic in the system documentation. However, in a few cases, large refactorings or document name changes occurred that the topic evolutions were unable to detect (false negatives). In these cases, even though documents were moved from one directory to another and methods were moved from one document to another, the topic weight metric remained the same or very similar. As a result, the topic evolution exhibited a stay and the project stakeholders would not be informed about these changes. We note that in some refactorings, enough

Version V_9	
	z_{18}
`EditServer.java`	0.69
`Install.java`	0.43
`LatestVersionPlugin.java`	0.55
...	
A(z_{18}, V_9) =	4.22

Version V_{10}	
	z_{18}
`EditServer.java`	0.60
`Install.java`	0.30
`LatestVersionPlugin.java`	0.29
...	
A(z_{18}, V_{10}) =	3.22

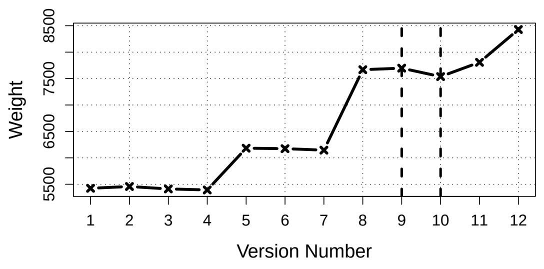
Figure 6.10: An example from jEdit of an assignment change caused by noise in the LDA model. The topic memberships for topic 18 ("view") are shown for three documents for versions V_9 and V_{10}. In this example, none of the text in the three documents changed between the two versions, but due to the probabilistic processes of LDA, some of their topic memberships did change slightly. As a result, the assignment metric for this topic changes between the two versions.

changes were made to produce a spike or drop in the topic, which *were* detected by our technique.

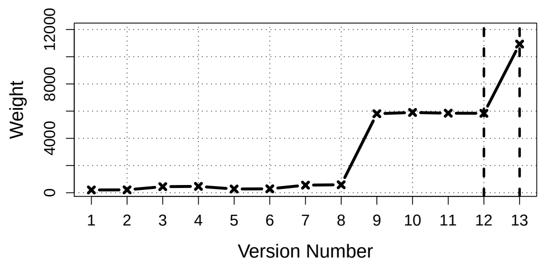
Examples

Figure 6.11a shows a change event for the "view" topic in jEdit that we classified as invalid. In this example, the weight metric decreased by 23% between versions V_9 and V_{10}, which is indeed a drop according to our definition in Equation 6.4. However, after manual analysis, it became clear that no actual changes occurred to any of the source code documents that matched this topic. The change in metric value was caused by noise in the LDA model: four of the seven documents that matched this topic received a lower topic membership in V_{10} than they did in V_9, even though no changes occurred to any of the documents during this period.

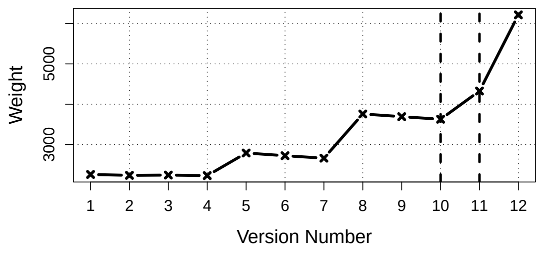
On the other hand, Figure 6.11b shows a change event from JHotDraw that we classified as valid. The evolution for the "color chooser" topic experiences a large spike (87%) in the weight metric between versions V_{12} and V_{13}. The release notes for version V_{13} include the description: "*An UI delegate for JColorChooser has been added to the "Palette" look and feel*". Additionally, new source code documents like `DefaultColorSliderModel.java` and `HSLColorSpace.java` were added into the `org.jhotdraw.color` package, both matching the "color chooser" topic. Thus, there is a clear correspondence between the spike event in



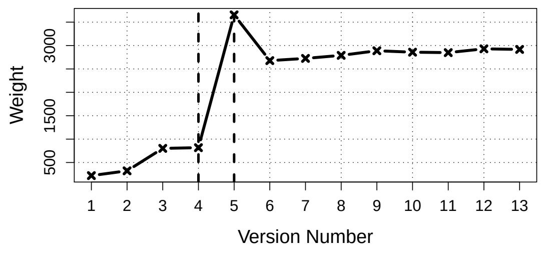
(a) An invalid drop in jEdit's "view" topic.



(b) A valid spike in JHotDraw's "color chooser" topic.



(c) A spike in jEdit's "macro action" topic, which was caused by the addition of new functionality (AppleScript).



(d) A spike in JHotDraw's "elem elem" (XML) topic, which was caused by the addition of a new library (NanoXML).

Figure 6.11: Example topic evolutions from JHotDraw and jEdit. The selected change event is highlighted by the vertical dashed lines.

the topic evolution and the actual change activity in the source code.

In both of our case studies, we found that the majority of topics evolve due to actual change activities in the source code, with only a small minority of change events caused by noise or confounded topics in the probabilistic LDA model. We conclude that topic evolution models are appropriate and useful for describing source code evolution.

6.4.3 Investigation of Code Change Categories (RQ2)

In this section, we investigate the relationship between topic evolutions and source code change categories. Longo et al. (2008) recently proposed three categories of software evolution interventions (i.e., reasons for software evolution):

C1. Corrective evolution (i.e., bug fixes)

C2. Refactoring (i.e., code improvement and adaptation)

- C2.1. Adoption of coding conventions and style
- C2.2. Adoption of new framework or libraries
- C2.3. Improvement of the internal structure of the code

C3. New functionalities and features

We refer to each item above as a *change category*. These change categories are not mutually exclusive—it is possible, for example, for the adoption of new libraries to also result in new features, or for bug fixes to result in improvement of the internal structure of the code.

We adopt this taxonomy as a high-level model for why software changes. We revisit our two systems under study, JHotDraw and jEdit, and manually analyze the valid change events from the randomly-selected subset of change events. (We use the same subset as we did in RQ1.) We do not consider stay events because by definition stay events involve no code changes. We again use our analysis tool, now with the goal of determining to which change category (or change categories) each change event belongs. We investigate system documentation and source code changes to determine whether each spike or drop is related to a bug fix, refactoring, or new functionalities or features.

Results

Table 6.4 shows the results of manually classifying each randomly-selected change event as one of the five possible change activities. We make a few observations below.

– JHotDraw and jEdit do not exhibit the same behaviors. For example, JHotDraw development is less concerned with bug fixes (8% compared to 64%) while being more concerned with internal improvement (79% compared to 28%). This makes sense, because the development history of JHotDraw was focused on the internal design of the system with emphasis placed on increasing the use of design patterns and standard libraries.

– New coding conventions (C2.1) do not occur often for either system (4% and 6%).

– Drops are rarely caused by the addition of new functionalities or features. This makes intuitive sense, because adding features involves adding new code, which always causes spikes in the weight metric for one or more topics.

	JHotDraw			jEdit		
	All	Spikes	Drops	All	Spikes	Drops
C1. Corrective evolution (i.e., bug fixes)	8%	11%	4%	64%	67%	50%
C2. Refactoring						
C2.1. New coding conventions	4%	2%	8%	6%	5%	13%
C2.2. New frameworks, libraries, etc.	10%	4%	0%	2%	3%	0%
C2.3. Internal improvement	79%	70%	96%	28%	26%	38%
C3. New functionalities and features	19%	46%	8%	49%	59%	0%

Table 6.4: Relationship between code changes and topic evolutions in JHotDraw and jEdit. The columns may add up to more than 100% because the categories are not mutually exclusive.

– Drops in JHotDraw almost always indicate one of the refactoring change activities. Drops in jEdit, on the other hand, could indicate any change activity.

Examples

Figure 6.11c shows an example spike at version V_{11} in the evolution of the “macro action” topic in jEdit. In our manual analysis, we classified this spike as being related to the addition of new functionality (C3). This topic deals primarily with the macro scripting language feature in jEdit. The release notes for version V_{11} include the line: *“You can* [now] *run AppleScripts (compiled, uncompiled and standalone)”*, referring to the new capability of running scripts written in the AppleScript language. This capability was realized by adding new documents at version V_{11}, for example the `AppleScriptHandler.java` which has a membership of 0.88 in the “macro action” topic.

Figure 6.11d shows an example spike at version V_5 in the evolution of the “elem elem” (XML) topic in JHotDraw. In our manual analysis, we classified this spike as being related to the addition of a new library. The release notes for version V_5 include the line: *”Added a tweaked version of NanoXML back into the framework”*. The NanoXML library consists of 24 documents, many of which match the “elem elem” topic with a membership of 0.50 or higher. Thus, this spike was related to the addition of a new library.

We have found that topics evolve due to a variety of underlying change activities, including bug fixes, refactoring efforts, and the addition of new functionalities. For this reason, we conclude that topic evolutions are a convenient and meaningful analysis tool for understanding source code evolution.

6.4.4 Exploring Evolution Patterns (RQ3)

We now examine topic evolution patterns. Our goal is to quantify common and uncommon behavior and explore the tendencies of topic evolutions.

Results

Table 6.5 summarizes the types and amounts of change events in the discovered topic evolutions for the two systems, for both the assignment and weight metrics. In both systems, there are more spikes than drops in the weight metric, consistent with the overall code growth over the studied periods. On average, a topic's weight evolution in JHotDraw has 4 spikes, 1 drop, and 7 stays; compare this to jEdit's relatively inactive 1 spike, 0 drops, and 10 stays. When a spike does occur, it does so by a median of 670 (JHotDraw) and 537 (jEdit) non-unique words (i.e., the median spike contains an addition of 670 and 537 non-unique words into the source code), but much smaller and larger spikes do occur. When a drop occurs, it is marked by the removal of a median of 838 (JHotDraw) and 365 (jEdit) non-unique words. In total, the weight evolutions in JHotDraw exhibited 251 spikes, 42 drops, and 247 stays, while jEdit exhibited 150 spikes, 11 drops and 334 stays.

We identify the following patterns of the evolution of the weight metric and provide relevant examples.

Overall Growth. Ninety-six percent (JHotDraw and jEdit) of topics have a higher weight in the last version of the source code than they do in the first, indicating total overall growth over the lifetime of the source code. This can also be observed from Figure 6.6, in which most topics appear darker towards the end of their lifetime. For example, in JHotDraw, the "input stream" topic (topic 2) has a weight of 306.0 in version V_1 and a weight of 4,920.8

	JHotDraw				jEdit			
	Min.	Median	Mean	Max.	Min	Median	Mean	Max.
Number of assignment spikes	0.0	2.0	1.9	4.0	0.0	0.0	0.5	3.0
Number of assignment drops	0.0	0.0	0.4	2.0	0.0	0.0	0.0	1.0
Number of assignment stays	7.0	10.0	9.7	12.0	8.0	11.0	10.5	11.0
Assignment change (spikes)	0.1	2.7	5.8	147.9	1.0	2.0	2.8	13.7
Assignment change (drops)	-147.4	-3.2	-11.9	-1.0	-2.4	-2.0	-1.9	-1.2
Number of weight spikes	1.0	4.0	3.9	6.0	0.0	1.0	1.4	4.0
Number of weight drops	0.0	1.0	0.8	4.0	0.0	0.0	0.1	2.0
Number of weight stays	4.0	7.0	7.4	11.0	7.0	10.0	9.5	11.0
Weight change (spikes)	39.0	670.0	1589.0	72230.0	54.9	537.5	903.3	7090.0
Weight change (drops)	-68210.0	-838.1	-3749.0	-68.9	-614.2	-365.6	-352.4	-99.1

Table 6.5: Characteristics of the discovered evolutions for JHotDraw and jEdit, for the assignment and weight metrics.

in version V_{13}, capturing the increasing capabilities of JHotDraw for reading various input file formats. In jEdit, the “hyper search” topic (topic 5) increased its weight from 2,623.0 in version V_1 to 4,453.3 in version V_{12}, as the searching functionality in jEdit was greatly enhanced over time.

Similarly, only 4% (JHotDraw and jEdit) of topics have a lower weight in the last version than they do in the first version. For example, in JHotDraw, the “displai box” topic (topic 22) has a large weight in the first version of the source code, but almost no weight in the last version. This indicates the removal of the “display box” topic. (In this case, the removal occurred at version V_5, when a new `jhotdraw.draw` package was added to refactor older classes such as `RectangleFigure.java`, which contained the “displai box” topic.) No topics in either JHotDraw or jEdit had the same weight in the last version as they did in the first version.

Major Events. Ninety-six percent (JHotDraw) and 16% (jEdit) of topics have at least one change event that changed the weight metric value by more than 75% of the previous version, indicating a major change of some kind. (To calculate this measure, we only included changes that involved an actual metric change of 50.0, to avoid scenarios where small changes to small values result in large percentage changes, for example when a topic weight changes from 2.0 to 4.0, i.e., a 100% increase that is obviously not important.) As JHotDraw was completely refactored and rewritten several times, it makes sense that most topics will exhibit major events. For example, the weight of the "color chooser" topic (topic 14) changed from 5,838.4 in version V_{12} to 10,930.3 in version V_{13}, a percentage difference of +87%. As previously noted in Section 6.4.2, the color chooser module of JHotDraw was heavily modified in version V_{13}, changing its GUI look and feel and adding several new classes.

Births and Deaths. Seven percent of topics in JHotDraw die at some point (i.e., have a positive weight value in some version V_j followed by a zero value in the remaining versions $V_{j+1}, ..., V_n$), indicating the removal of features and deletion of code. For example, in JHotDraw, the "undo activ" topic (topic 16) is removed at version V_5. (The corresponding release notes include the entry "Undo/Redo is now implemented based on the Swing undo package", indicating that JHotDraw no longer implements its own undo/redo functionality, and instead uses an off-the-shelf package in Swing.)

In jEdit, on the other hand, no topics die. This could mean that either jEdit is more stable and all features are useful, or that when features are removed, the obsolete code is not deleted.

Likewise, 45% (JHotDraw) and 7% (jEdit) first appear (i.e., are born) at some version V_j, $j > 1$. This indicates that JHotDraw has experienced a significant change between the first and last versions, since almost half of the topics were not present in the first version. For example, in JHotDraw, the "junit doclet" topic (topic 17) is first introduced in version V_2, which corresponds to when JUnit testing was first added to the system. jEdit is relatively more stable, but still exhibits some completely new topics. For example, the "instal" topic (topic 38) first appears in version V_5. Three new packages were added in this version, all related to an enhanced installation procedure for jEdit: `windows.jeditshell.jedinstl`, `windows.jeditshell.jeditinit`, and `windows.jeditshell.jeditlauncher`.

Constant Topics. Whereas none of the topics in JHotDraw are constant (i.e., experience no spikes or drops), 22% of the topics in jEdit are. While these topics do exhibit some change over time, the change is always less than the δ threshold. This finding reveals that no feature or concept is safe from change in JHotDraw, likely due to the constant-improvement mentality of the system. On the other hand, features and concepts in jEdit follow more the "if it is not broken, do not fix it" paradigm. For example, the "xml pars" topic (topic 11), which captures jEdit's XML parsing feature, exhibits hardly any changes, and when it does, the changes are trivial.

Changes per Version. On average, 60% (JHotDraw) and 7% (jEdit) of topics change in any given version. This implies that each release in JHotDraw brings changes to many topics concurrently, possibly indicating a hidden coupling of the code or perhaps the aggressive nature of the developers. On the other hand, fewer topics are changed at each release in jEdit, indicating either a more modular code design or a more focused development cycle.

Unstable Topics. We call a topic *unstable* if it exhibits both more spikes and drops than stays—the topic metric rapidly spikes and drops in succession. Such a behavior could indicate a poorly designed topic that undergoes constant refactorings. We observe that only 5% (JHotDraw) and 0% (jEdit) of topics are unstable. An example unstable topic in JHotDraw is the "content produc" topic (topic 42), which represents the abstract concept of producing content in the included sample applications. Since the sample applications change significantly from version to version in order to highlight the changes made to the JHotDraw framework, the topic experiences unstable behavior.

Spike-only Topics. Forty-three percent (JHotDraw) and 74% (jEdit) of topics exhibit spikes without also exhibiting drops. No topics (JHotDraw and jEdit) exhibit drops without spikes. This is due to the tendency of source code of the studied systems to only grow over time (Figures 6.1b and 6.1e) as features and functionalities are added.

An example in JHotDraw is the "attribut kei" topic (topic 8), which is related to GUI buttons (which change internal attributes based on attribute keys). As JHotDraw grows over time and more GUI functionality is added, this topic experiences many spikes. In jEdit, the topic "out write" (topic 2), which captures the logging functionality of jEdit, is added to more and more of jEdit's classes over time, and rarely removed from a class.

Overall, the topic evolutions in JHotDraw are very active, as evidenced by their overall growth (96%), the number of major events (95%), few constant topics (0%), and many changed topics per version (60%). This is consistent with JHot-Draw's active development cycle and multiple redesign efforts. Comparatively, the topic evolutions in jEdit are more calm: many are constant (22%), and few are changed per version (7%). This is consistent with jEdit's more mature and stable system status. Topics in both systems tend to grow, not shrink, as does the size of the source code. Finally, topics can be born and can die—not all topics exist in every version of the source code, representing major new or deleted features and concepts in the source code.

6.4.5 Potential Threats to Validity

Internal Validity. Our work involves choosing several parameters for LDA computation, perhaps most importantly the number of topics, K. Also required for LDA is the number of sampling iterations, as well as prior distributions for topic and document smoothing parameters, α and β. There is currently no theoretically guaranteed method for choosing optimal values for these parameters, even though the resulting topics are obviously affected by this choice. To counteract this limitation, we use the same value for K as previous work (Baldi et al., 2008; Thomas et al., 2010b) and let MALLET automatically choose optimal values for α and β (McCallum, 2012).

Additionally, our choice of δ (metric change threshold) was based on finding a knee in a curve of change events. We used this data-driven technique in an attempt to choose a value that eliminated most noisy changes and preserved most actual changes. Still, our results are affected by this subjective choice to some degree.

We performed several preprocessing steps on the source code documents, such as splitting, stemming, stopping, and pruning. Although most research to date also performs some combination of these steps before applying information retrieval models to source code (see Section 2.4), there is currently no guidance or consensus on which steps are actually necessary or beneficial for topic evolution models. (We note that in Chapter 8, we found that all preprocessing steps add value.)

We performed a detailed manual analysis of the discovered topic evolutions. However, as we were familiar with the details of the topic modeling techniques, it is possible that we were unknowingly biased (overly harsh or overly generous) during evaluation. Our work could be enhanced by also having outsiders to our study manually analyze the discovered topic evolutions. Further, as only the first author performed the manual analysis, our work can be enhanced by having additional people perform the analysis and comparing results.

External Validity. We have focused our in-depth analysis efforts on JHotDraw and jEdit, due to their robust designs, extensive documentation, and manageable sizes. However, our results may be dependent on these qualities and thus generalize poorly to systems with worse designs. Furthermore, as both JHotDraw and jEdit are medium-sized open-source systems, we cannot yet be sure if our results generalize well to small or large sized open-source systems, or to any closed-source systems. We also cannot generalize our results with any confidence to systems from different communities, such as databases or web browsers. Additional case studies are needed to investigate these alternatives.

6.5 Conclusion

In this chapter, we performed a detailed investigation of the usefulness of an advanced IR model, the Hall topic evolution model, for analyzing software evolution. We applied the Hall topic evolution modeling technique to the history of the unstructured linguistic data of the source code a software system and computed metrics on the discovered topics. We found that most of these metric changes correspond well with actual software change activities (87–89%), such as corrective evolution, internal improvements, and the addition of new features. The change events that we found to be inaccurate or invalid were mostly caused by noise in the probabilistic LDA model, although they did not occur often. Our case studies on JHotDraw and jEdit support the notion that topic models are useful tools for mining the unstructured source code data of a software system in order to uncover its conceptual evolution.

There are many ways to extend this research. First, improved topic evolution models can be used. We have found that some of the incorrectly discovered change events (i.e., spikes, drops, and stays) in our technique are due to noise in the Hall model. Since LDA is a probabilistic process based on sampling techniques, some randomness is inevitable when LDA assigns topics to documents. This randomness is responsible for some false-alarm change events. We believe that performing one or more smoothing operations on the discovered evolutions may mitigate this issue, further improving the usefulness of our technique. In addition, some of the incorrectly discovered change events are due to confounded topics, i.e., those topics that try to represent more than one real-world concept. In future work we wish to explore the possibility of automatically detecting and removing (or splitting) these topics to further increase the accuracy of our technique.

Recently, a new topic evolution model, called the Diff model (Thomas et al., 2011), has been developed specifically for source code histories. The Diff model is an extension to the Hall model that applies LDA only to the changes of a document between successive versions, as opposed to the full document of each version. The idea is that since source code changes relatively infrequently, a given document is likely to be mostly similar to its previous version. This similarity can bias the topics that LDA discovers by skewing the word co-occurrence frequencies. The Diff model has been shown to provide modest to significant improvement over the Hall model, in terms of topic distinctness, evolution accuracy, and evolution sensitivity (Thomas et al., 2011).

A second avenue for future work is the improvement of the way in which we detect change events. In this chapter, we used a simple technique to detect change events, namely Equation 6.4, which is based on the δ threshold. In future work, we would like to explore more advanced detection techniques, such as trend detection or time-series analysis. More advanced techniques could help deal with noise in the LDA model and help detect slowly-spiking or slowly-dropping topics.

Finally, we could consider additional metrics. In this thesis, we focused primarily on the assignment and weight metrics, which give different views on the overall presence of a topic in the source code. These metrics allowed us to perform our initial studies on the correspondence between changes in topic metrics and change activity in source code. However, additional metrics exist for topics, as outlined in Section 6.3.4. We hypothesize that some of these metrics (or a combination of metrics) can be useful for automatically determining the type of change activity to the source code. For example, a refactoring or restructuring of the source code will likely result in a decreased scatter metric, while the weight metric will mostly stay unchanged or increase.

Part IV

Understanding IR Model Assumptions and Parameters

In Section 2.4, we found that researchers tend to use IR models as black boxes, without fully understanding their assumptions and parameter sensitivities. In this part, we take steps towards opening the black box to gain a better understanding of how IR models should be applied to unstructured software repositories.

– **Chapter 7: Addressing data duplication with the Diff model**. The state-of-the-art topic evolution model that has been used in previous software engineering research contains the implicit assumption that each document will occur exactly once over time. However, researchers either were not aware of this assumption, or have chosen to ignore it, because source code repositories violate this assumption: most source code documents do not change from version to version, resulting in many duplicates over time. In this chapter, we find that this data duplication negatively affects the results of topic evolution models. We propose the *Diff* model that alleviates this problem by removing all such duplication before applying the topic evolution model. Through case studies on two open source systems, JHotDraw and PostgreSQL, we compare the Diff model against the state-of-the-art topic evolution model, one which software engineering researchers have used in the past, and find our proposed Diff model produces more distinct topics and more accurate topic evolutions.

– **Chapter 8: Understanding the effects of data preprocessing and IR model parameters**. Many researchers do not consider the effects of data preprocessing (e.g., splitting identifiers, word stemming, removing stop words) or IR model parameters (e.g., term weighting, similarity measures) on model performance. As a result, research is inconsistent in the choices of preprocessing steps and parameter values, leaving future researchers and practitioners in a state of flux. Worse, their particular choices are far from ideal, resulting in decreased model performance. In this chapter, we embark in a large empirical study in the context of bug localization to determine whether these design decisions matter, and if so, which particular choices are best. Through case studies on three real-world systems, Eclipse, Mozilla, and IBM Jazz, we find definitively that the preprocessing steps and IR model parameters are significantly important to bug localization results, as the performances between difference choices can vary by an order of magnitude. practices for making these design decisions.

CHAPTER 7

Addressing Data Duplication with the Diff Model

Studying the evolution of topics in a software system is an emerging technique to automatically shed light on how the system is changing over time: which topics are becoming more actively developed, which ones are dying down, or which topics are lately more error-prone and hence require more testing. Existing techniques for modeling the evolution of topics in software systems violate the implicit assumption of the underlying IR models that documents are unique across time. To address this issue, we propose the Diff *model, which applies a topic model only to the changes of the documents in each version instead of to the whole document at each version. A comparative study with a state-of-the-art topic evolution model shows that the Diff model can detect more distinct topics as well as more sensitive and accurate topic evolutions, which are both useful for analyzing source code histories.*

Publications based on this chapter: Thomas et al. (2011)

7.1 Motivation

RECENT RESEARCH HAS found that topic evolution models are a valid and useful way to automatically describe and monitor the changes to source code concepts (Thomas et al., 2010b). Indeed, case studies have provided evidence in this direction.

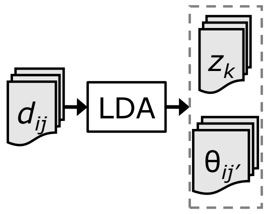
However, we maintain that topic evolution models (Section 2.2) were designed for corpora in which the documents are unique across time. This is true, for example, for conference proceedings: it is not the case that an article one year is only slightly updated and republished the next year. Instead, each article (i.e., the specific combination of words within an article) is unique across time. However, source code repositories are typically updated incrementally, with each version making only small changes to the previous. Therefore, we would expect to see significant overlap in the data (i.e., word co-occurrences) between versions, especially when the systems have a long version history.

Consider, for example, two open source systems, 13 versions of JHotDraw (Gamma, 2012) and 46 versions PostgreSQL (PostgreSQL, 2012) (Table 7.1). We make two key observations.

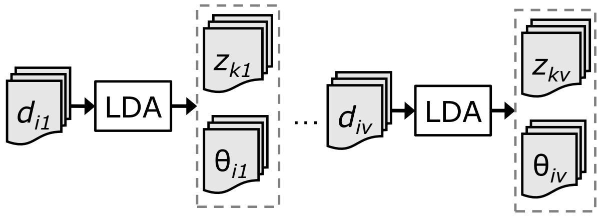
1. Most files are not altered between versions. On average, between 16% (PostgreSQL) and 36% (JHotDraw) of the source code files experienced some change between versions, measured by the number of files that had any change activity (i.e., lines added, removed, or modified). In other words, on average at least 64% (JHotDraw) and up to 84% (PostgreSQL) of the source code files are exact duplicates from release to release. These unaltered documents will obviously have the same word co-occurrences as their previous versions, since no changes were made.

2. Most changes are very small. For the average file that experienced a change between versions, only 0.1% (both PostgreSQL and JHotDraw) of its words actually changed, measured as the number of changed words over the number of total words in the file. Almost all of a file's content remains unaltered, and hence the word co-occurrences will largely be the same.

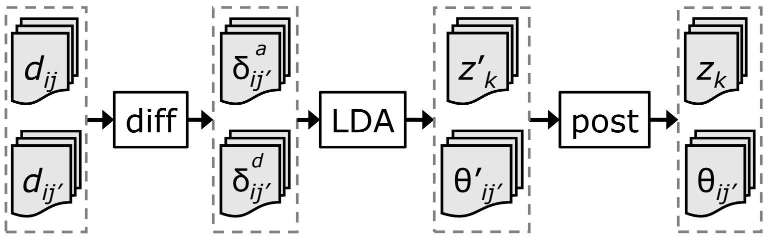
We hypothesize that the above observations affect the results obtained by the Hall model. Recall that the Hall model applies LDA to all versions of all files. Since we know that most files are not changed at all between versions, and even the files that are changed are not changed by much, we can conclude that the word co-occurrences that LDA operates on will be skewed in the direction of the duplicated files.



(a) The Hall model applies LDA to all versions at the same time.



(b) The Link model applies LDA to each version separately, then links the topics across versions.



(c) The Diff model adds a diff step to isolate the changes between versions of each document and a post-processing step to reconstruct the membership vectors. Here, $d_{ij'}$, is the next version of d_{ij}.

Figure 7.1: A graphical depiction of the three topic evolution models.

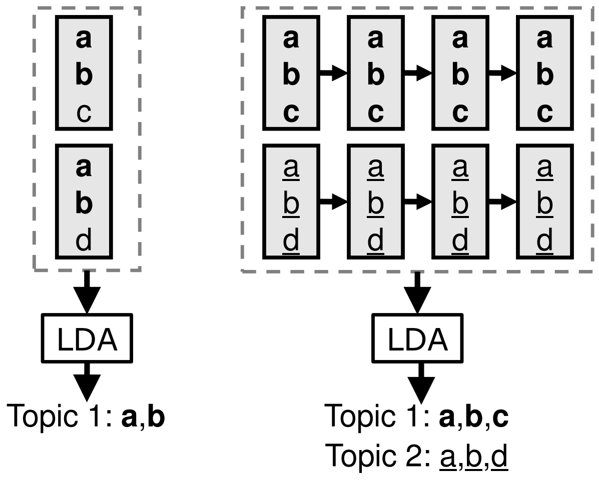


Figure 7.2: On the left, a repository without duplication (2 unique files, 3 words each) yields a preferable topic. On the right, a repository with duplication (2 files, 4 unchanged versions each) yields confounded topics (should only be a single topic $\{a, b\}$).

		Min.	Med.	Mean	Max.
JHotDraw	% files changed per version	<0.1	33.8	35.9	89.5
	% words changed per changed file	<0.1	0.1	0.2	35.2
PostgreSQL	% files changed per version	<0.1	6.3	15.9	73.8
	% words changed per changed file	<0.1	0.1	0.2	17.7

Table 7.1: Change characteristics of JHotDraw and PostgreSQL.

7.2 Proposal

In order to address the data duplication effect found in source code histories, we propose a simple but effective topic evolution model, the ***Diff*** model. The key idea is that the Diff model prepends a *diff step* to the Hall model to isolate the changes between successive versions of each document. This diff step effectively removes all duplication and leaves only the changed portion of the document, hence ridding the corpus of the duplication effect. The Diff model can be thought of as an extension to the Hall model, since the models are largely the same. However, as we will show in this chapter, the diff step is critical when applying topic evolution models to software repositories with duplication.

7.2.1 Diff Model Description

Figure 7.1c depicts the Diff model process. For each source code document d_{ij} in the system, we first compute the edits between successive versions V_j and $V_{j'}$ ($j' = j + 1$) using the GNU `diff` file comparison tool (GNU, 2010). `diff` classifies each edit as an *add* or *delete*, depending on whether the edit resulted in more or fewer lines of code, respectively. If an existing line is modified, it is considered by `diff` to be deleted and then added again. For each version of each document, we create two *delta documents*, $\delta^a_{ij'}$ and $\delta^d_{ij'}$. We place all the added lines between d_{ij} and $d_{ij'}$ into $\delta^a_{ij'}$ and all the deleted lines into $\delta^d_{ij'}$. We use the notation $|\delta^a_{ij'}|$ to represent the number of words in $\delta^a_{ij'}$.

We must handle two special cases: 1) When we encounter a new document d_i at version V_j (either because $j = 1$ or d_i is a new document), we classify the entire document as added words, and thus we add the entire document to the delta document δ_{ij}^{a}; and 2) when a document d_i is removed at version V_j, we classify the entire document as deleted words, and thus add the entire document to the delta document δ_{ij}^{d}.

Next, we apply LDA to the entire set of delta documents, resulting in a set of extracted topics and membership values for each delta document.

Finally, we post-process the output of LDA to compute the membership values of the original documents at each point in time. The corresponding $\theta_{d_{ij}}$ vector of a document d_i at version V_j is defined recursively as

$$\theta_{d_{ij}} = \frac{\theta_{d_{i\hat{j}}} \left|d_{i\hat{j}}\right| + \theta_{\delta_{ij}^{a}} \left|\delta_{ij}^{a}\right|}{\left|d_{i\hat{j}}\right| + \left|\delta_{ij}^{a}\right|} - \varphi\left(\frac{\theta_{d_{i\hat{j}}} \left|d_{i\hat{j}}\right| - \theta_{\delta_{ij}^{d}} \left|\delta_{ij}^{d}\right|}{\left|d_{i\hat{j}}\right| - \left|\delta_{ij}^{d}\right|}\right) \tag{7.1}$$

where $\varphi()$ is the normalizing function and $\hat{j} = j - 1$ is the index of the previous version of document d_i. $\varphi()$ accounts for the scenario when more words matching a given topic were subtracted than were in the previous version of the document for that topic (e.g., subtracting 20 words about topic A when the previous version only had 10 words about topic A), which is unlikely but possible because LDA is a probabilistic process.

The intuition behind Equation 7.1 is that for each version of a document, we adjust the θ vector by adding the lines of the δ^{a} document and subtracting the lines of the δ^{d} document, thus arriving at a representative state of the document at each version. This cumulative definition is necessary since we only model the *changes* at each version, but we want to know the θ vector for the entire document at each version.

7.2.2 On The Origin of the Diff Model

The Diff model was originally motivated by our own experiences and struggles when applying the Hall model to source code histories. In a previous study, we applied the Hall model to the source code history of JHotDraw to study its evolutions of topics (Thomas et al., 2010b). At the time, the Hall model was the known standard for such a task, and we found the results to be acceptable.

In a subsequent study, we applied the Hall model to a repository with significantly more versions than JHotDraw, and hence, more duplication. Understanding the topic evolutions in this case proved to be difficult, and we felt there was something wrong with the topics and evolutions. After considerable investigation, we hypothesized the cause of the problem to be the duplication effect, and the Diff model was created to address this issue.

We note that the Diff model can be viewed as an extension to the Hall model, for two primary reasons. First, after the diff step, the Hall and Diff models perform equivalent actions on the data. Second, when applied to a traditional corpus with no duplication (i.e., a history of conference proceedings), the diff step will have no effect—it will essentially be a `noop`. Thus, as we will show, the Diff model improves the results of Hall on repositories that have duplication; on all other repositories, the two models are equivalent.

7.3 Case Studies

We now perform an empirical evaluation of the Diff and Hall models. As was described in Section 7.1, we hypothesize that when the Hall model is applied to a repository with duplication, it will generate imperfect topics that confound multiple concepts. On the other hand, the Diff model will create more distinct topics that stand on their own and thus allow the documents to be described more naturally. In fact, producing distinct topics is known to be a desirable property for topic models (Blei et al., 2010).

In addition, because the topics discovered by the Diff model are more distinct and better describe the documents, we hypothesize that the resulting topic evolutions will more accurately describe the changes to the repository.

We now formulate the research hypotheses that we focus on in our case study.

Hypothesis 1. The removal of data duplication will result in more *distinct* topics.

Hypothesis 2. More distinct topics will allow the discovery of more *sensitive* evolutions.

Hypothesis 3. More distinct topics will allow the discovery of more *accurate* evolutions.

We test our first two hypotheses by conducting an experiment on real-world open source systems (Section 7.3.1), and we build a *simulated* system, whose properties are well-known, to test our third hypothesis (Section 7.3.2).

7.3.1 Experiment 1: Evaluation on Real-World Systems

Our goal in this section is to determine whether there is a difference between the Hall and Diff models in the distinctness of the discovered topics and the sensitivity of the discovered topic evolutions.

Studied Systems

We applied both models to the source code histories of two open source systems: JHotDraw (Gamma, 2012) and PostgreSQL (PostgreSQL, 2012). JHotDraw is a medium-sized drawing framework implemented in Java and has been the subject of many previous studies. PostgreSQL is a large database management system and is chosen due to its extensive documentation.

Our JHotDraw dataset consists of 13 releases (versions 5.2.0–7.5.1). The latest release contains 613 files totaling 84K source lines of code (SLOC). Our PostgreSQL dataset is comprised of 46 release versions (versions 7.0.0–8.3.5). The latest release contains 844 files totaling 501K SLOC.

Study Setup

We preprocessed the source code of each system using the steps described in Section 2.2. Namely, we isolated identifier names and comments from the source code, removed stopwords, split identifiers, performed word stemming, and pruned the vocabulary so that overly common (>80%) or overly rare (<2%) words are removed. For JHotDraw, the preprocessing resulted in a total of 2.3M words (964 of which are unique) in 5,833 documents. For PostgreSQL, the preprocessing resulted in 40M words (2,867 of which are unique) in 29,559 documents.

For the LDA computation, we used MALLET version 2.0.6 (McCallum, 2012). We ran for 10,000 sampling iterations, the first 1,000 of which were used to optimize the α and β parameters (Griffiths and Steyvers, 2004). We modeled JHotDraw with $K = 45$ topics and PostgreSQL with $K = 100$ topics. We chose more topics for PostgreSQL because it has a larger, more complex code base.

Evaluation Measures

A ***change event*** in a topic evolution is an increase (*spike*), decrease (*drop*), or no change in a metric value (*stay*) between successive versions. We classify a change event as a spike or drop if there is at least a 20% increase or decrease in metric value compared to the previous version, and as a stay otherwise. Formally, for a metric m of topic z_k at version V_i, the change $c = (m(z_k, V_i) - m(z_k, V_{i-1}))/m(z_k, V_{i-1})$ is classified as

$$\text{Event}(m, z_k, V_i) = \begin{cases} \text{spike} & \text{if } c \geq 0.2, \text{or if } m(z_k, V_{i-1}) = 0 \\ & \qquad \text{and } m(z_k, V_i) > 0; \\ \text{drop} & \text{if } c \leq -0.2; \\ \text{stay} & \text{otherwise.} \end{cases} \tag{7.2}$$

A distinct topic is one that stands on its own—it is not similar to any other discovered topics. We define the ***topic distinctness*** of a topic z_i as the mean KL divergence between the word membership vectors of z_i and z_j, $\forall j \neq i$:

$$\text{TD}(\phi_{z_i}) = \frac{1}{K-1} \sum_{j=1, j \neq i}^{K} \text{KL}(\phi_{z_i}, \phi_{z_j}). \tag{7.3}$$

A higher TD measure indicates that a topic is more distinct. We use the TD measure to test Hypothesis 1.

We define the ***evolution sensitivity*** of an evolution $E(z_i)$ as the mean number of detected spikes per version of the system:

$$\text{ES}(E(z_i)) = \frac{|\{\text{Detected spikes and drops in } E(z_k)\}|}{(v-1)}. \tag{7.4}$$

If a detected evolution has more spikes and drops, then we say it is more sensitive than an evolution with fewer spikes and drops. We use the evolution sensitivity measure to test Hypothesis 2. To ensure that the detected spikes are not false positives, we manually investigate topic evolutions in a controlled environment in Section 7.3.2.

7.3.2 Experiment 2: Evaluation on Simulated Data

We now test Hypothesis 3 by quantifying the *accuracy* of each model when applied to source code histories—whether each model is able to detect source code changes correctly and completely. Hypothesis 2 concluded that the Diff model created more sensitive topic evolutions, but it might be the case that the model is overly sensitive, discovering false-positive change events. To investigate this possibility, we must assess the accuracy of the discovered change events.

Since there is no public dataset for evaluating the accuracy of topic evolution models, we perform a controlled experiment on a manually-created *simulated* software system. We have created two simple scenarios with a representative variety of source code changes so that we could exactly determine whether the evolutions extracted by the models were accurate— that is, whether the change events detected by each model correspond to the actual changes that we introduced in the source code (precision) and whether the detected evolutions contained all the changes that we introduced in the source code (recall).

Data Generation

We built the simulated software system by starting with the `backend.access` (version 8.2.1) subsystem of PostgreSQL. The `backend.access` subsystem contains 58 source code files and 8 subdirectories, and is responsible for implementing functionalities such as hash tables, transactions, and NBTrees. We chose this subsystem due to its medium size and clear functionality definitions.

We created two simulated scenarios as follows. First, we made ten duplicates of all 58 source code files in the `backend.access` to create ten (unchanged) versions of the package.

We called these versions (V_1–V_{10}) the baseline scenario.

The first scenario modifies the baseline scenario by introducing three documents from the unrelated `timezone` subsystem at version V_5, then removing all three documents at version V_6. Thus, there are 58 files in versions V_1–V_4, 61 files in version V_5, and again 58 files in versions V_6–V_{10}. This scenario simulates two typical actions: the introduction of new functionality and the removal of existing functionality.

The second scenario starts with the documents of the first scenario and makes the following two additions: 1) Eight documents from the unrelated `ecpg` subsystem were inserted in versions V_9 and V_{10}. The first half of each document was inserted in version V_9, while the second half of each document was inserted at version V_{10}. 2) Five documents from the unrelated `backend.regex` subsystem were inserted in version V_1, they remained (unchanged) in versions V_2 and V_3, and were removed at version V_4.

Study Setup

We preprocessed the source code of each system under study using the steps described in Section 2.2 in the same way as we did for the real-world systems (Section 7.3.1). For Scenario 1, the preprocessing resulted in a total of 1.2M words (3,629 of which are unique) in 583 documents. For Scenario 2, the preprocessing resulted in a total of 1.2M words (3,666 of which are unique) in 614 documents.

For the actual LDA computation, we used the same setup as in Section 7.3.1, with the exception that we modeled each simulated scenario using $K = 20$ topics.

Evaluation Measures

To quantify the accuracy of each model, we calculate precision and recall. The ***precision*** of a model describes how many of the discovered change events were correct. We compute the precision of an evolution $E(z_k)$ as

$$P(E(z_k)) = \frac{|\{\text{Correct events in } E(z_k)\}|}{|\{\text{All discovered events in } E(z_k)\}|}. \tag{7.5}$$

		Hall	Diff	p-value
JHotDraw	Mean topic distinctness	3.72	4.43	<0.001
	Spikes per version	9.58	15.17	<0.001
PostgreSQL	Mean topic distinctness	2.56	3.64	<0.001
	Spikes per version	2.58	4.49	<0.001

Table 7.2: Results of our case studies on JHotDraw and PostgreSQL.

Diff topic 63:	*acl role privileg mode oid grant owner roleid*
Diff topic 40:	<u>stmt</u> <u>creat</u> <u>comment</u> <u>defel</u> <u>command</u> <u>defnam</u>
Hall topic 68:	*oid* <u>stmt</u> tupl <u>comment</u> *owner* rel <u>creat</u> *acl list*

Table 7.3: The "ACL" and "commands" topics from Diff, and the "ACL-commands" topic from Hall. The words in Diff topic 63 are *emphasized* and the words in Diff topic 40 are <u>underlined</u> in the Hall topic 68.

We are able to determine which discovered change events are correct since we have manually created the changes in the simulated system. For example, we expect to see a spike at version V_5 in the evolution relating to the `timezone` subpackage, since we first added the `timezone` documents at V_5.

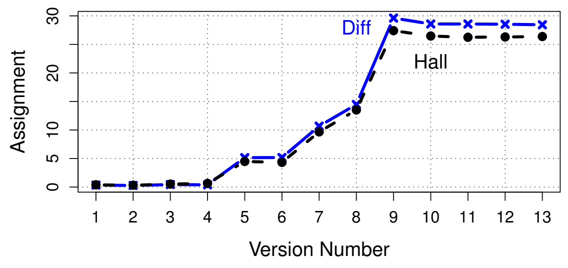
The ***recall*** of a model describes how many of the truth events were discovered by the model. We compute the recall of an evolution $E(z_k)$ as

$$R(E(z_k)) = \frac{|\{\text{Correct events in } E(z_k)\}|}{|\{\text{Truth events}\}|}. \tag{7.6}$$

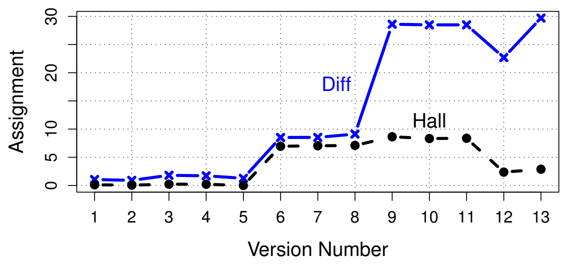
We are able to determine which truth events exist because we manually created the truth events in the source code.

7.4 Results and Discussion

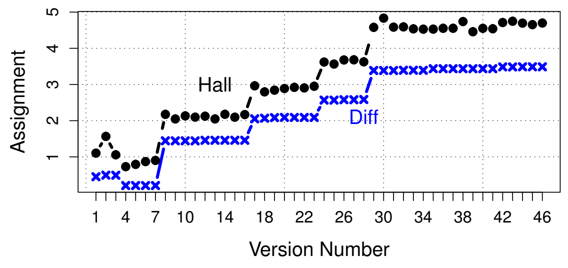
We now present the results of our two experiments.



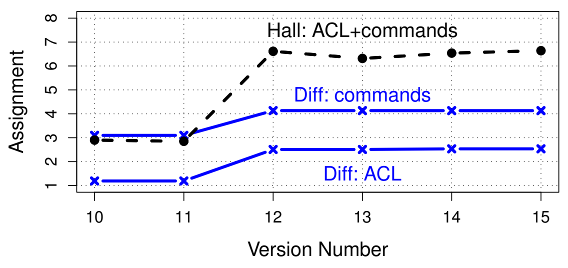
(a) Similar evolutions: the "menu icon" topic.



(b) Dissimilar evolutions: the "color gradient" topic.



(c) Similar evolutions: the "multi xact" topic.



(d) Evolutions for the "ACL" and "commands" topics.

Figure 7.3: Sample topic evolutions from the case studies on JHotDraw ((a), (b)) and PostgreSQL ((c), (d)). In all plots, the dashed black line (with circles as points) shows the evolution discovered by the Hall model while the solid blue line (with crosses as points) shows the evolution discovered by the Diff model.

7.4.1 Experiment 1: Real-World Systems

To illustrate the kind of evolutions that are discovered by the two models, Figures 7.3a–7.3c show three discovered evolutions. Figures 7.3a and 7.3c show examples of evolutions that were similar between the two models for JHotDraw and PostgreSQL, respectively. In these cases, the topics themselves contained similar words and thus their evolutions followed similar paths. Figure 7.3b shows example evolutions from JHotDraw that were different between the two models, despite the topics being similar.

After computing our evaluation measures on the resulting topics and evolutions, we make the following two observations.

Observation 1. The Diff model produces more distinct topics than the Hall model, supporting Hypothesis 1.

Table 7.2 shows the topic distinction measures for the two models for both JHotDraw and PostgreSQL (Equation 7.3). In both systems, the Diff model produces significantly more distinct topics, supporting our first research hypothesis.

To illustrate this, consider the following example from the case study on PostgreSQL. The `lockcmds.c` document is responsible for taking a lock command from the user, checking the access control list (ACL) to see if the user has permissions, and performing a lock on a table if the user does have permissions. Thus, the document contains the concepts of "commands" and "access control lists". In the Diff model, this document is matched to two topics: topic 63 (with membership .25), which describes access control lists, and topic 40 (with membership .58), which describes commands. Under the Hall model, `lockcmds.c` is assigned to only one topic, topic 68 (with membership 0.93), which confounds the locking and commands concepts into a single topic. Table 7.3 shows theses topics and highlights the similarity between Hall's topic 68 and the two Diff topics, 63 and 40. The Hall topic 68 is less distinct, with a topic distinctness of 2.49, compared to the relatively distinct Diff topic distinctnesses of 3.46 (topic 40) and 3.49 (topic 63).

Observation 2. The Diff model is more sensitive to detecting changes in the data than the

Hall model, supporting Hypothesis 2.

Table 7.2 shows the results of evolution sensitivity for the two models for both JHotDraw and PostgreSQL, measured as the mean number of detected spikes per version (Equation 7.4). In both systems, the Diff model detects significantly more spikes than the Hall model. This finding supports our second hypothesis and is in part a result of Observation 1: since the topics in the Diff model are more distinct, new documents will be matched to more topics, and thus there will be more spikes in the evolution. On the other hand, the Hall model finds less distinct topics (i.e., each topic has multiple concepts confounded), and new documents will tend to be matched to fewer topics, resulting in fewer spikes in the evolution.

To illustrate this difference, consider again the `lockcmds.c` file from PostgreSQL, which was first added to the system at version V_{12} along with a group of similar files. Figure 7.3d shows the evolutions produced by the Diff and Hall models that match the `lockcmds.c` file. Under the Diff model, two topics received spikes between version V_{11} and V_{12}, which were the topics for ACLs and commands. Under the Hall model, however, only a single topic received a spike, which was the confounded ACL and commands topic.

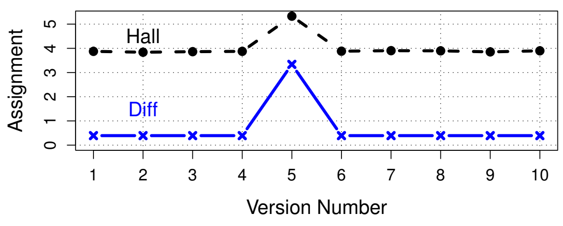
The Diff model produces more distinct topics and more sensitive topic evolutions.

7.4.2 Experiment 2: Simulated Data

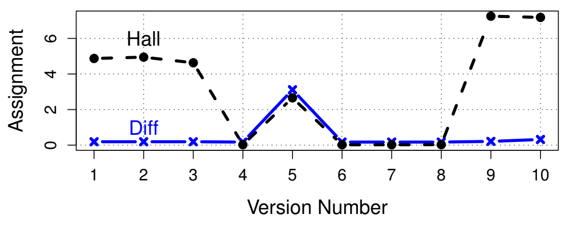
Scenario 1 (Single Change): This scenario contains 10 unchanging versions of 58 documents, with the exception that three documents from the unrelated `timezone` subsystem were inserted at version V_5, and removed at version V_6.

Expectations: We expect to see a spike in a `timezone` related topic at version 5, a drop at version 6, and no change in assignment to any other topic at any other time.

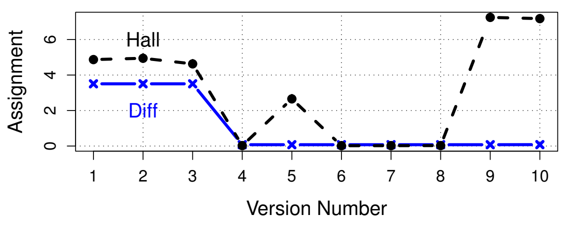
Results: Figure 7.4a shows the discovered topic evolutions for the `timezone` related topics. (Figure 7.5 shows the actual topics.) In this scenario, the Diff model creates a topic just for these `timezone` documents—the evolution has a value of 0 at all versions except at version



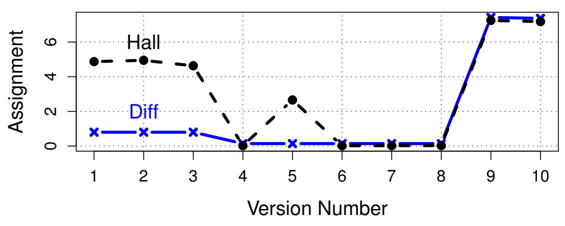
(a) The `timezone` topic evolutions in Scenario 1.



(b) The `timezone` topic evolutions in Scenario 2.



(c) The `regex` topic evolutions in Scenario 2.



(d) The `ecpg` topic evolutions in Scenario 2.

Figure 7.4: Sample topic evolutions for the simulated scenarios. In all plots, the dashed black line (with circles as points) shows the evolution discovered by the Hall model, while the solid blue line (with crosses as points) shows the evolution discovered by the Diff model.

V_5, where it spikes to an assignment value of around 3.0, then drops again at version V_6. Indeed, all three of the `timezone` documents have high memberships in this topic, and low memberships in all other topics. Likewise, no documents besides the three `timezone` documents have a non-zero membership in this topic.

The Hall model, on the other hand, does not create a topic solely for the `timezone` documents. Instead, the Hall model assigns the three `timezone` documents to an existing topic that already had a non-zero assignment value from other, unrelated documents. This topic spikes and drops at versions V_5 and V_6, respectively. In this case, we say that the discovered evolution is incorrect, because the change events are discovered in a topic that is not related to the timezone documents.

Table 7.4 lists the precision and recall measures for Scenario 1 across all topics. In this scenario, there is only one truth spike and one truth drop. Since the Hall model found two incorrect events and no correct events, it achieves a precision and recall measure of 0.0. The Diff model, on the other hand, correctly discovered the truth spike and drop without

		Hall model		Diff model	
		Precision	Recall	Precision	Recall
Scenario 1 (Single Change)	Spikes	0.0	0.0	1.0	1.0
	Drops	0.0	0.0	1.0	1.0
Scenario 2 (Multiple Changes)	Spikes	0.60	1.0	0.75	1.0
	Drops	0.66	1.0	0.66	1.0

Table 7.4: Precision and recall results of the experiments on the simulated data.

spurious events, achieving precision and recall values of 1.0. In all cases, the Diff model achieves higher recall and precision measures, and Hypothesis 3 is supported.

Scenario 2 (Multiple Changes): This scenario is the same as Scenario 2 except that eight documents from the `ecpg` subsystem are added at version V_9 and five documents from the `backend.regex` subsystem are present (and unchanging) in versions V_1–V_3.

Expectations: We expect to see a spike for the `timezone` related topic at version V_5, followed by a drop at version V_6. We also expect to see a spike for the `ecpg` related topic at version V_9 and a drop in the `backend.regex` related topic at version V_4.

Results: Figures 7.4b–7.4d show the discovered topic evolutions for the three subpackages involved in the scenario. (Figure 7.5 shows the actual topics.) The Hall evolutions in the three figures are actually showing the same topic, because only a single topic was created to house the `timezone`, `ecpg`, and `backend.regex` related documents. While the three figures show that the expected events were indeed discovered by the Hall method (i.e., a drop at version V_4, a spike and drop at version V_5 and V_6, and a spike at version V_9), the topic itself confounds three separate concepts and hence is not easy to interpret (further supporting Hypothesis 1).

The Diff model, on the other hand, captured all three subpackages in their own distinct topics, each having the expected change events.

Table 7.4 shows the precision and recall results. In this case, since the Hall model created a new topic for the newly introduced documents, and captured all of the expected change events correctly, we say that all of the evolutions in Figures 7.4b–7.4d are correct. However, other topic evolutions discovered spurious change events, causing the precision score to be less than 1. The Diff model also discovered a few spurious change events (including the drop shown at version V_4 in Figure 7.4d), and also receives a precision score less than 1. Even still, Hypothesis 3 is supported because the Diff model outperforms the Hall model.

The Diff model produces more accurate topic evolutions than the Hall model.

7.4.3 Further Evaluation of Accuracy

Our case studies on real-world systems have confirmed our first two hypotheses: the Diff model results in more distinct, understandable topics, and it is more sensitive for detecting the changes in a source code repository. Furthermore, our study on a simple, manually-created simulated system provides support for our third hypothesis—that the Diff model produces evolutions that are more accurate.

These initial results, based on mathematical metrics and simulated data, are encouraging. We plan to further test our hypotheses by conducting a user study to determine the usefulness of the models for practitioners. We have already performed an initial user study to this effect, where the first author blindly rated events (spikes and drops) in the topic evolutions of JHotDraw and PostgreSQL against system documentation to determine whether the events were justified (Thomas et al., 2010a). The results of this initial study were positive, with precision scores of 92% for the Diff model, compared to 69% for the Hall model. We are currently preparing a larger scale user study to confirm these findings.

7.4.4 Threats to Validity

Parameter Choices. During our studies, we had to choose values for several model parameters, including the number of topics (K); the delta threshold for classifying an event as a spike or drop; the number of sampling iterations to run in LDA; and the pruning parameters

Scenario 1: Diff: Topic 7 (Label: seq search)	
Words	*time year case continu const offset type rule hentri*
Docs	`scheck.c`, `strftime.c`, `localtime.c`
Scenario 1: Hall: Topic 17 (Label: normal transact)	
Words	*transact clog xid accum entri statu subtran commit datum*
Docs	`reloptions.c`, `transam.c`, `ginbulk.c`
Scenario 2: Diff: Topic 2 (Label: offset posix section)	
Words	*time case year continu type tmp offset const result*
Docs	`scheck.c`, `strftime.c`, `localtime.c`
Scenario 2: Hall: Topic 9 (Label: ecpglog ecpgget)	
Words	*var case chr state struct break reg end assert*
Docs	`scheck.c`, `strftime.c`, `localtime.c`
Scenario 2: Diff: Topic 5 (Label: case sql)	
Words	*case sqlca ecpg break connect ecpgt var lineno pval*
Docs	`error.c`, `data.c`, `descriptor.c`
Scenario 2: Diff: Topic 14 (Label: scan consist fine)	
Words	*chr state reg assert end var struct begin subr*
Docs	`regexec.c`, `regcomp.c`, `rege_dfa.c`
JHotDraw: Diff: Topic 6 (Label: icon icon)	
Words	*icon descriptor bean properti event gen method color set*
Docs	`JAttributeSliderBeanInfo`, `ODGPropertiesPanelBeanInfo`, `AbstractToolBarBeanInfo`
JHotDraw: Diff: Topic 12 (Label: color color)	
Words	*color gradient space index compon rgb model system min*
Docs	`ColorSystem`, `HSLRGBColorSystem`, `HSLRYBColorSystem`
JHotDraw: Hall: Topic 23 (Label: icon icon)	
Words	*icon descriptor bean properti event gen method color set*
Docs	`JAttributeTextArea`, `JDisclosureToolBar`, `JLifeFormattedTexArea`
JHotDraw: Hall: Topic 5 (Label: stop color)	
Words	*color gradient space paint focu fraction stop arrai linear*
Docs	`LinearGradientPaintContext`, `GradientPaintContext`, `MultipleGradientPaint`
JHotDraw: Hall: Topic 14 (Label: color chooser)	
Words	*color compon slider index icon model space system rgb*
Docs	`AbstractHarmonicRule`, `HSLRGBColorSystem`, `HSLRYBColorSystem`
PostgreSQL: Diff: Topic 37 (Label: multi xact)	
Words	*page multi xact clog share lock offset ctl xid*
Docs	`slru.c`, `clog.c`, `varsup.c`
PostgreSQL: Diff: Topic 63 (Label: acl acl)	
Words	*acl role privileg priv mode oid user grant aclcheck*
Docs	`acl.c`, `aclchk.c`, `superuser.c`
PostgreSQL: Hall: Topic 5 (Label: arg lappend)	
Words	*constraint stmt column list tabl attr index pstate type*
Docs	`analyze.c`, `tablecmds.c`, `tupdesc.c`
PostgreSQL: Hall: Topic 68 (Label: gettext noop)	
Words	*guc gettext pgc val config conf variabl noop sourc*
Docs	`guc.c`, `variable.c`, `guc-file.c`

Figure 7.5: Selected topics. The figure contains the topic label, top words, and top matching documents for selected topics from our case study.

for preprocessing the source code histories. Our choices are somewhat subjective, as there is no standard way to determine optimal values. However, we used the same parameter values in both the Hall and Diff models, providing an opportunity for an equal comparison.

Generality of Results. Although we studied two real-world systems from different domains, of different sizes, and implemented in different programming languages, we cannot necessarily generalize our results to all other systems. First, the systems we studied were both open source, and therefore we cannot generalize our results to proprietary systems developed in the industry. Second, both studied systems follow established variable naming schemes and often use descriptive comments where possible, which permit meaningful topics to be discovered. In systems that do not follow standard naming schemes or have consistent commenting practices, topic models might not be effective tools.

7.5 Conclusion

Topic evolution models have the potential to help software developers understand the histories and trends of their software repositories in a new and deep way. However, traditional topic evolution models were designed for versioned corpora whose documents are never duplicated, such as conference proceedings and newspaper articles. We have found that source code histories deviate from this norm: in some cases, 99% of the source code files in a new version are unmodified copies from the previous version. This *duplication effect* alters the results of topic models.

These characteristics of software repositories motivated us to propose a new topic evolution model, which is a simple but powerful extension to the Hall model. Our model operates only on the changes between versions of the repository, effectively eliminating the duplication effect. We evaluated the Diff model through case studies on two real-world software systems, JHotDraw and PostgreSQL, as well as a simulated software system that is well-understood. We found that the Diff model produces significantly more distinct topics, which are preferable to the confounded topics found by the Hall model. We also found that the more distinct topics allowed more sensitive and accurate evolutions to be discovered, which better model the changes in a source code history.

These encouraging results motivate us to consider additional case studies (simulated and real) to confirm the results seen here, as well as evaluate the Diff model against the Link model.

CHAPTER 8

Understanding the Effects of Data Preprocessing and IR Model Parameters

Most research to date does not consider the effects of data preprocessing and IR model parameters when mining unstructured software repositories. In this chapter, we perform a large-scale empirical study that explicitly and carefully considers such effects. Using bug localization as our context, we determine whether and how data preprocessing and IR model parameters affect the results of IR models. Through case studies on Eclipse, IBM Jazz, and Mozilla, we find that the data preprocessing and parameters of a IR model have a significant effect on bug localization performance, and therefore practitioners and researchers must consider them carefully. In particular, we find that the VSM model, compared to LSI and LDA, achieves the best top-20 performance; using both the bug report's title and description is best; using the source code's identifiers, comments, and past bug reports is best; and all three preprocessing steps (stopping, stemming, and splitting) each improve performance. Finally, tf-idf weighting and cosine similarity are best for VSM.

Publications based on this chapter: Thomas et al. (2012d)

8.1 Motivation

WE RETURN TO the task of bug localization, defined in Section 5. In bug localization, a classifier identifies a list of possibly-relevant source code entities, given a bug report. A developer uses the list make the necessary modifications to the source code.

Current bug localization research uses IR classifiers to locate source code entities that are textually similar to bug reports, based on the unstructured text fields in the bug report. However, current results are ambiguous and contradictory: some claim that VSM provides the best performance (Rao and Kak, 2011), while others claim that LDA is best (Lukins et al., 2010), while still others claim that a new IR model is needed (Nguyen et al., 2011). These mixed results are due to the use of different datasets, different performance metrics, and different classifier configurations (see Table 8.1). (A *classifier configuration* is a specification of all the parameters that define the behavior of the classifier, such as the the way in which the source code is preprocessed, how terms are weighted, and the similarity metric between bug reports and source code entities.) As a result, researchers and practitioners have conflicting advice on how to best use IR models to localize bugs.

8.2 Proposal

In this chapter, we aim to resolve the ambiguity in current research results by performing a large-scale empirical study to compare thousands of IR classifier configurations on a large quantity of bug reports. By using the same datasets and performance metrics, we can perform an apples-to-apples comparison of the various configurations, identifying (a) how big of an impact configuration has on performance and (b) which particular configurations are best.

In the next section, we propose a framework for defining and analyzing large sets of IR classifier configurations. We note that this framework can also be used in a similar manner to study the effects of classifier configurations on other software engineering tasks, such as traceability linking, test case prioritization, and defect prediction.

8.3 Case Study

The goal of this case study is to evaluate the space of classifier configurations: which data representations, preprocessing steps, and other IR model parameter values result in the best bug localization performance?

Table 8.1: Summary of the classifier configurations used by existing bug localization work, and their performance results. Question marks indicate that the parameters were not specified.

	Classifier configuration				Studied systems	Performance metric	Results*
	Entity rep.	Bug rep.	Preprocess	IR model			
Lukins et al. (2010)	Idents + comments	Title + descr.	Stem	LDA (K=100, α=50/K, β=0.01, *iters*=?, *Sim*=CP)	Mozilla	Mean rank	5.8
					Rhino	Mean rank	1–1,062
					Eclipse	Mean rank	492–11,234
Nguyen et al. (2011)	Idents + comments	Title + descr.	Split+stem	LDA (K= 300, α=0.01, β=0.01, *iters*=?, *Sim*=cosine)	Jazz	Top-20	0.39
					Eclipse	Top-20	0.33
					AspectJ	Top-20	0.28
					ArgoUML	Top-20	0.34
				BugScout (K= 300, α=0.01, β=0.01, *iters*=?, *Sim*=cosine)	Jazz	Top-20	0.48
					Eclipse	Top-20	0.39
					AspectJ	Top-20	0.51
					ArgoUML	Top-20	0.45
Rao and Kak (2011)	Idents	??	Split	VSM (*TW*=tf-idf, *Sim*=cosine)	AspectJ	MAP	0.0796
				LSI (*TW*=tf-idf, K=500, *Sim*=cosine)	AspectJ	MAP	0.0650
				LDA (K=150, α=0.33, β=0.01, *iters*=?, *Sim*=KL)	AspectJ	MAP	0.0125

* We present results to the best of our ability. Some papers only provide results in graph form, forcing us to estimate the exact value.

In this section we outline the design of our case study: which classifiers we test and our evaluation procedure. We use the same datasets, systems under test, and performance metrics as the case study in Chapter 5.

Classifiers Under Test

We build 1,368 IR-based classifiers based on the three popular IR models described in Section 2.2: VSM, LSI, and LDA. For each IR model, we must decide which bug report representation to use for the query, which source code entity representation we use to train the IR model, how we preprocess the bug report and source code representation, and the remaining parameter values for the particular IR model. Tables 5.1 and 5.2 show all the parameters and their possible values that we consider, which we now describe in more detail.

For source code entity represention, we consider six values. The first three are based on the text of the source code entity itself: the identifier names (i.e., variable and method names) only; comments only; and both identifiers and comments. As proposed by Nguyen et al. (2011), we also consider the past bug reports (PBR) related to a source code entity. To do so, we represent the source code entity as a collection of the text of all of its PBRs. The idea is that a new bug report might be more textually similar to a past bug report, as opposed to the identifier names or comments, of an entity, giving the IR model a better chance for success. We consider two values: using all the PBRs of a entity; and using just the 10 most recent PBRs of an entity. Finally, we consider all possible data for a entity: its identifier, comments, and all PBRs.

For bug report (i.e., query) represention, we consider three values: the title of the bug report only; the description of the bug report only; and both the title and description of the bug report.

We consider three common preprocessing steps (Section 2.2): splitting compound words based on common names schemes; removing stop words; and stemming. Since the application of each preprocessing step is binary (i.e., is performed or is not performed), and all three preprocessing steps can be applied independently, we test a total of 8 possible preprocessing techniques.

The VSM model has two parameters: term weighting and similarity score. For term weighting, we consider the tf-idf and sublinear tf-idf weighting schemes (Manning et al., 2008), as well as the more basic Boolean weighting scheme (Manning et al., 2008). For similarity score, we consider both the cosine and overlap similarity scores (Manning et al., 2008).

The LSI model has three parameters: term weighting, similarity score, and number of topics. We consider the same three term weighting schemes as we do for the VSM model. We hold the similarity score constant at cosine, since research has shown this to be the best similarity score for LSI (Manning et al., 2008). Finally, we consider four numbers of topics: 32, 64, 128, and 256.

The LDA model has five parameters: number of topics, a document-topic smoothing parameter, a topic-word smoothing parameter, number of sampling iterations, and similarity score. We consider four numbers of topics: 32, 64, 128, and 256. The LDA implementation we use, called MALLET (McCallum, 2012), automatically optimizes for the document-topic and topic-word smoothing parameters, so we do not manually set values for these parameters. We also do not manually specify the number of iterations, and instead let the model run until it has converged. Finally, we consider the conditional probability score, as it is most relevant for IR applications (Wei and Croft, 2006). Conditional probability does not require the bug reports to be included in the LDA model at run-time, which makes conditional probability much more flexible. (In practice, re-running the LDA model on the source code entities and the bug reports, for each and every new bug report, would be impractical.)

Fully Factorial Design To quantify the performance of all possible classifiers (given our considered parameters and their possible values) we use *a fully factorial* design of our case study (Kuehl, 2000). In this design, we explore every possible combination of parameter values. In our case, a fully factorial design results in a total 3,168 IR-based classifiers (VSM: 864=(3 bug data)*(6 entity data)*(8 preprocessing)*(3 term weights)*(2 similarity); LDA:

576=(3 bug data)*(6 entity data)*(8 preprocessing)*(4 no. of topics); LSI: 1728=(3 bug data)*(6 entity data)*(8 preprocessing)*(3 term weights)*(4 no. of topics)).

Analysis Method

We use two methods for analyzing the results of the classifiers. First, we sort the classifiers by their top-20 metric (Section 5.3) in descending order, which indicates the top overall performing classifiers. Second, we use Tukey's Honestly Significant Difference (HSD) test (Tukey and Braun, 1994) to compare the performance of each value of each parameter. The HSD test is a statistical test on the means of each parameter value—holding one parameter constant, and letting all other parameters vary. For a given parameter (e.g., "bug report representation"), the HSD test compares the mean of each possible value with the mean of every other possible value (e.g., "title only" vs. "description only" vs. "title+description"). Using the studentized range distribution (Tukey, 1991), the HSD test determines whether the differences between the means exceeds the expected standard error. The result of HSD is a set of statistically-equivalent groups of parameter values. If two parameter values belong to the same group, then the performances of the parameter values are not statistically different. Note that a parameter value can belong to multiple groups, and group memberships are not transitive: if parameter value A and parameter value B belong to the same group, and parameter value B and parameter value C belong to the same group, value A and value C do not necessarily belong to the same group.

8.4 Results and Discussion

From our analysis of 3,168 classifiers, each evaluated on 8,084 bug reports, we make the following conclusions.

- Configuration matters: the difference between one classifier configuration and another is often significant.

– The VSM classifier can achieve the best overall top-20 performance. LSI is second, and LDA is last.

– Using both the bug report's title and description results in the best overall performance, for all IR models.

– Using the source code entities' identifiers, comments, and past bug reports results in the best overall performance, for all IR models.

– Stopping, stemming, and splitting all improve performance, for all IR models.

Table 8.2 shows the best and worst four configurations of each of the three IR classification techniques (VSM, LSI, and LDA), for each of the three studied systems, ordered by the top-20 metric. (We provide the full set of results for all configurations and additional metrics online (Thomas, 2012).) For all three studied systems, the VSM classification technique achieves the best overall performance, consistent with previous findings (Rao and Kak, 2011). The best configuration of VSM varies slightly from system to system, and many configurations have comparable performance.

Table 8.3 shows the dispersion of the performance of the various configurations: the worst, average, and best configuration performances for each of the three classification techniques. We find that configuration matters: for all studied systems and all classification techniques, the differences between the worst configuration and best configuration is significant. The differences between the "average" configuration and the best configuration is also significant: for example, in Eclipse, using the VSM classification technique, the median top-20 performance is 21% (configuration VSM.A2.B2.C4.D1.E2), while the best performance is 55% (configuration VSM.A3.B6.C7.D1.E1). This suggests that the choice of bug report representation, entity representation, preprocessing steps, and IR model parameters has a large effect on the overall performance of a classifier, no matter the underlying classification technique.

We now present the results of Tukey's HSD statistical test on each parameter listed in Tables 5.1 and 5.2.

Table 8.2: The best four and worst four configurations, for each classifier family (VSM, LSI, LDA, and EM) and each studied system. The configurations are ordered according to their top-20 performance.

VSM			LSI			LDA		
Rank	Config.	Top-20	Rank	Config.	Top-20	Rank	Config.	Top-20
Eclipse								
1	VSM.A3.B6.C7.D1.E1	0.548	1	LSI.A3.B6.C4.F2.G256	0.462	1	LDA.A1.B4.C7.J128	0.290
2	VSM.A3.B6.C4.D1.E1	0.535	2	LSI.A3.B6.C7.F2.G256	0.444	2	LDA.A1.B4.C6.J128	0.290
3	VSM.A3.B3.C7.D1.E1	0.524	3	LSI.A3.B6.C1.F2.G256	0.438	3	LDA.A1.B5.C7.J128	0.282
4	VSM.A3.B6.C5.D1.E1	0.512	4	LSI.A3.B6.C2.F2.G256	0.434	4	LDA.A1.B5.C6.J128	0.282
861	VSM.A2.B3.C0.D3.E1	0.012	1725	LSI.A2.B6.C0.F1.G32	0.065	573	LDA.A2.B3.C3.J64	0.009
862	VSM.A2.B3.C0.D3.E2	0.012	1726	LSI.A2.B3.C3.F1.G32	0.061	574	LDA.A3.B6.C0.J64	0.009
863	VSM.A2.B3.C1.D3.E2	0.011	1727	LSI.A3.B3.C0.F1.G32	0.056	575	LDA.A3.B3.C0.J64	0.008
864	VSM.A2.B3.C1.D3.E1	0.011	1728	LSI.A2.B3.C0.F1.G32	0.044	576	LDA.A3.B6.C0.J32	0.007
Jazz								
1	VSM.A3.B6.C6.D1.E1	0.686	1	LSI.A3.B6.C2.F2.G256	0.588	1	LDA.A3.B1.C7.J256	0.336
2	VSM.A3.B6.C7.D1.E1	0.673	2	LSI.A3.B6.C6.F2.G256	0.584	2	LDA.A3.B3.C7.J256	0.331
3	VSM.A3.B6.C2.D1.E1	0.669	3	LSI.A3.B6.C6.F2.G128	0.576	3	LDA.A3.B6.C7.J256	0.330
4	VSM.A3.B6.C4.D1.E1	0.657	4	LSI.A3.B6.C2.F2.G128	0.574	4	LDA.A1.B6.C7.J256	0.318
861	VSM.A2.B6.C1.D3.E2	0.069	1725	LSI.A2.B3.C0.F2.G32	0.144	573	LDA.A2.B3.C0.J64	0.066
862	VSM.A2.B6.C1.D3.E1	0.068	1726	LSI.A2.B3.C0.F1.G64	0.139	574	LDA.A2.B3.C0.J256	0.065
863	VSM.A2.B6.C0.D3.E2	0.068	1727	LSI.A2.B3.C3.F1.G32	0.121	575	LDA.A2.B3.C3.J32	0.057
864	VSM.A2.B6.C0.D3.E1	0.068	1728	LSI.A2.B3.C0.F1.G32	0.098	576	LDA.A2.B3.C0.J32	0.051
Mozilla								
1	VSM.A3.B6.C6.D1.E1	0.802	1	LSI.A3.B3.C2.F2.G128	0.794	1	LDA.A3.B6.C7.J128	0.523
2	VSM.A3.B6.C7.D1.E1	0.796	2	LSI.A3.B3.C6.F2.G128	0.782	2	LDA.A3.B6.C7.J256	0.522
3	VSM.A3.B6.C4.D1.E1	0.794	3	LSI.A3.B3.C2.F2.G256	0.781	3	LDA.A3.B6.C4.J256	0.517
4	VSM.A3.B6.C5.D1.E1	0.788	4	LSI.A3.B6.C2.F2.G128	0.778	4	LDA.A3.B6.C7.J64	0.505
861	VSM.A2.B2.C1.D3.E2	0.067	1725	LSI.A2.B4.C1.F3.G32	0.242	573	LDA.A1.B5.C4.J256	0.058
862	VSM.A2.B2.C1.D3.E1	0.066	1726	LSI.A2.B4.C0.F3.G32	0.240	574	LDA.A1.B5.C2.J256	0.058
863	VSM.A2.B2.C0.D3.E2	0.062	1727	LSI.A2.B4.C5.F3.G32	0.228	575	LDA.A1.B5.C7.J256	0.000
864	VSM.A2.B2.C0.D3.E1	0.061	1728	LSI.A2.B4.C3.F3.G32	0.225	576	LDA.A1.B5.C6.J256	0.000

Table 8.3: Performance dispersion amongst classifier configurations, using the top-20 performance metric.

	Configs.	Min.	1st Qu.	Med.	Mean	3rd Qu.	Max.
Eclipse							
VSM	864	0.011	0.083	0.209	0.211	0.321	0.548
LSI	1728	0.044	0.175	0.228	0.234	0.287	0.462
LDA	576	0.007	0.053	0.074	0.088	0.101	0.290
Jazz							
VSM	864	0.068	0.214	0.322	0.325	0.430	0.686
LSI	1728	0.098	0.306	0.354	0.358	0.403	0.588
LDA	576	0.051	0.149	0.195	0.192	0.238	0.336
Mozilla							
VSM	864	0.061	0.271	0.441	0.433	0.604	0.802
LSI	1728	0.225	0.441	0.536	0.537	0.647	0.794
LDA	576	0.000	0.228	0.322	0.318	0.418	0.523

Table 8.4: The results of Tukey's HSD test for the bug report representation parameter, for each IR classification technique and each studied system. If two values appear in the same group, then their top-20 performances were not statistically different.

VSM			LSI			LDA		
Group	Mean	Parameter value	Group	Mean	Parameter value	Group	Mean	Parameter value
Eclipse								
a	0.238	A1 (title)	a	0.259	A3 (title+descr.)	a	0.131	A1 (title)
a	0.215	A3 (title+descr.)	b	0.227	A1 (title)	b	0.070	A3 (title+descr.)
b	0.180	A2 (descr.)	b	0.217	A2 (descr.)	b	0.062	A2 (descr.)
Jazz								
a	0.354	A1 (title)	a	0.399	A3 (title+descr.)	a	0.214	A1 (title)
a	0.351	A3 (title+descr.)	b	0.362	A1 (title)	a	0.206	A3 (title+descr.)
b	0.270	A2 (descr.)	c	0.314	A2 (descr.)	b	0.158	A2 (descr.)
Mozilla								
a	0.458	A1 (title)	a	0.555	A3 (title+descr.)	a	0.346	A1 (title)
ab	0.439	A3 (title+descr.)	a	0.542	A1 (title)	a	0.325	A3 (title+descr.)
b	0.402	A2 (descr.)	b	0.515	A2 (descr.)	b	0.282	A2 (descr.)

Bug Report Representation

Table 8.4 shows the HSD results for the bug report representation parameter. (We provide the HSD results for the remaining parameters online (Thomas, 2012).) A2 (description) always (i.e., for all three IR models and all three studied systems) belongs to the bottom group and never belongs to the top group, meaning that A2 is always significantly worse than at least one other value. A3 (title and description) almost always belongs to the top group: in one instance (LDA model, Eclipse), A3 is in the second group. A1 (title) is similar in that it is usually in the top group (2 exceptions: LSI model, Eclipse; and LSI model, Jazz). Since both A1 and A3 include the title of the bug report, and A2 does not, we theorize that including the title from the bug report representation is most important; whether you also include the description is of secondary importance. This result is likely because the bug descriptions introduce noise into the IR models, often because the descriptions include stacktraces. Stacktraces have the names of several files and methods, which can lead the IR model in the wrong direction. In contrast, the title is typically carefully constructed to exactly summarize the problem.

Entity Representation

The best choice of source code entity representation differs between the studied systems. In Mozilla and Jazz, B6 (all available data: identifiers, comments, and past bug reports) belongs to the top group no matter the IR model. Therefore, in these studied systems, more information is best. In addition, B4 (all past bug reports) and B5 (last 10 bug reports) do not belong to the top group in any of the IR models, indicating that identifiers and comments play a more important role in these studied systems.

In Eclipse, on the other hand, B4 and B5 belong to the top group for all three IR models. B6 appears in the top group for VSM, and the second of two groups in LDA, and the second of four groups for LSI. Therefore, in Eclipse, including past bug reports is most important; including identifiers and/or comments is of secondary importance.

One possible explanation for the difference between studied systems is the number of snapshots in our analysis. Our Eclipse dataset contains 16 snapshots, compared to the 8 and 10 of Jazz and Mozilla, respectively. Given the nature of our evaluation procedure, any configuration using PBRs only will achieve low performance during the first snapshot of the system, since by definition the source code entities haven't yet been linked to any past bug reports. The IR models are built on empty representations of the entities, and bug localization is simply random. While this is true for all three studied systems, Eclipse has more subsequent snapshots to mitigate the effects of the poor performance of the first snapshot, since the top-20 metric is averaged across all snapshots.

Overall, we conclude that B6 (all) is the best: For Mozilla and Jazz, B6 is always in the top group; in Eclipse, B6 is always in one of the top 2 groups. In addition, B6 is robust to cases when there are no past bug reports to which to link the entities, since identifiers and comments are also included.

Preprocessing Steps

C7 (stopping, stemming, and splitting) is in the top group for all studied systems and all IR models; it is the only parameter value that is. C4 (split and stop) is almost always in the top group (one exception: LDA, Jazz). C6 (stop and stem) is almost always in the top group (two exceptions). C2 (stop) is in the top group in all but four cases. C1 (split) is almost never in the top group; it is only twice. C3 (stem) is only in the top group once. Finally, C0 (none) is always in the bottom group, never in the top group. We conclude that performing all three preprocessing steps is most beneficial, and removing stop words is the most important of the three steps.

IR Model Parameters

VSM parameters For VSM, tf-idf (D1) weighting is always (i.e., for each studied system and each IR model) best, and sublinear tfi-df (D2) is always second best, and boolean (D3) weighting is always last. Cosine similarity (E1) is always better than overlap similarity (E2).

LSI parameters For LSI, sublinear tf-idf (F2) is always best, tf-idf (F1) is always best or second best, and Boolean (F3) is always last.

The best number of topics depends on the studied system. In Mozilla, 64 and 128 topics are best. In Eclipse, 256 topics is best. In Jazz, 128 and 256 topics are best. These roughly correlate with the number of LOC in each studied system, suggesting the the number of topics should be selected based on the size of the studied system.

LDA parameters For LDA, it appears like more topics are better, but there are no significant differences in any studied system between 64, 128, and 256 topics. Thirty-two topics, however, is always significantly worst.

The configuration of a classifier matters. We find that the VSM classifier, compared to LSI and LDA, achieves the best top-20 performance; using both the bug report's title and description is best; using the source code's identifiers, comments, and past bug reports is best; and all three preprocessing steps (stopping, stemming, and splitting) improve performance. Finally, tf-idf weighting and cosine similarity are best for VSM.

Discussion

No single classifier is best for all three studied systems. However, a configuration of the VSM classifier has the best performance for all studied systems, suggesting that VSM is the overall best classification technique for bug localization. For all studied systems, VSM $>$ LSI $>$ LDA, when considering each technique's best configuration.

Interestingly, for all studied systems, the LSI classification technique has the best worst performance, often significantly so. In Mozilla, for example, the worst LSI configuration has a performance of 23%, compared to VSM's 6% and LDA's 0%.

Mozilla is relatively easy compared to the other studied systems: the best top-20 for Mozilla is 80%, compared to Jazz's 69% and Eclipse's 55%. We also note that Mozilla is the smallest system, Jazz is the second smallest, and Eclipse is the largest.

In general, we find that more is better: take all the bug report data you can (A3); take all the entity data you can (B6); and do all the preprocessing steps you can (C7). Each of these parameter values had the best overall performance across the various IR models and studied systems.

We stress that the configuration of a classifier has a significant impact on its results. The difference between the best and worst configuration is huge; even small differences (e.g., deciding not to remove stop words) can result in large performance variations. Researchers and practitioners should be careful when configuring their bug localization classifiers. Overall, we recommend the VSM.A3.B6.C7.D1.E1 classifier (VSM using the bug report title and description, the source code entities' identifiers, comments, and past bug reports, preprocessed by splitting, stopping, and stemming, using tf-idf weighting, and using the cosine similarity metric), for two reasons. First, it achieves the best performance in Eclipse and the second best performance in Jazz and Mozilla. In addition, all the configuration settings (i.e., A3, B6, C7, D1, and E1) were shown to be optimal by Tukey's HSD statistical test.

8.4.1 Threats to Validity

This chapter shares potential threats to validity with Chapter 5, as the same datasets were used and systems were studied.

8.5 Conclusion

Solving the bug localization task is difficult, due to the unstructured nature of bug reports, but has major implications for developers, since it can dramatically reduce the time and effort required to maintain software. In this chapter, we empirically investigated how sensitive IR models are to data preprocessing and IR model parameters, i.e., the configuration of an IR classifier.

We found that the configuration of a classifier has a significant impact on its performance. In Eclipse, for example, the difference between a poorly configured IR classifier and a properly configured IR classifier is the difference between having a 1 in 100 chance of finding a relevant entity in the top 20 results and having better than a 1 in 2 chance. Fortunately, we found consistent results of the various configurations across all three studied systems, indicating that the proper configuration is not system specific.

We conclude with the following recommendations.

– The best IR-based classifier uses the Vector Space Model, trained with tf-idf term weighting on all available data in the source code entities (i.e., identifiers, comments, and past bug reports for each entity), which has been stopped, stemmed, and split, and queried with all available data in the bug report (i.e., title and description) with cosine similarity.
– The best EM-based classifier uses the new bug count metric to rank source code entities.

By identifying which classifier configurations are best, our results have substantially improved the state-of-the-art in bug localization. We conclude that even in the face of unstructured text in bug reports and unstructured linguistic data in source code, IR models can effectively localize bugs and reduce maintenance efforts and costs for developers.

The framework we used to define and analyze large sets of classifier configurations can be re-used in other software engineering domains, allowing researchers to better understand the effects of IR model configurations in those domains.

In future work, we wish to further validate our findings on additional high quality datasets. We would like to investigate finer-grained snapshots, better bug-to-entity linking algorithms, and training on methods rather than entire files. In addition, we wish to build explanation systems into the bug localization process, i.e., tell users *why* certain entities are relevant for a given bug report. Finally, we wish to conduct user studies to quantify how useful displaying related entities to the user will be in practice.

Part V

Epilogue

CHAPTER 9

Conclusion

The field of mining software repositories uses readily-available data to increase the productivity of developers and reduce project costs. Using all available data, both structured and unstructured, maximizes benefits. This thesis has shown how to advance the state-of-the-art of using IR models to mine unstructured software repositories, by moving towards three new paradigms: addressing new software engineering tasks; using advanced IR techniques; and opening the black box of IR models to better understand their assumptions and parameters.

Our overarching thesis is that the research and practice of using IR models to mine software repositories can be improved by going beyond the current state-of-the-art. Through empirical evaluations of our proposed techniques on real-world case studies, and through comparisons with existing techniques whenever possible, this thesis has provided strong evidence that software development practices can indeed be improved, providing support for our thesis.

IR models are not the only way to mine software repositories, and this thesis does not argue that existing methods should be removed or replaced. Rather, by using IR models in concert with existing methods, a much larger percentage of available data can be harnessed. In addition, by moving towards new software engineering applications, advanced

IR models, and a better understanding of IR models, researchers and practitioners can mine more of their repositories to infer as much actionable knowledge as possible.

9.1 Main Contributions of Thesis

Our overall goal in this thesis was to enhance the state-of-the-art by addressing the shortcomings in current research that uses IR models to mine unstructured software repositories, and to persuade future researchers and practitioners to follow our lead. We divide our specific contributions into three categories.

9.1.1 Applying IR Models to New Software Engineering Tasks

Proposing and evaluating a technique to help developers prioritize their test cases. We used LDA to determine the similarity between test cases, based on their underlying linguistic content. By prioritizing those test cases which were most dissimilar, we ensured that different parts of the system were being tested and that distinct faults would be detected. Our novel IR-based technique improved fault detection rates by up to 31% compared to state-of-the-art coverage-based techniques.

Proposing and evaluating a technique to analyze the interaction of source code and mailing lists. We used LDA to identify topics that were shared between the source code and the mailing list, and analyzed the behavior of those topics over time. Specifically, we defined four possible life cycles states that a topic can be in at any given time (inactive, discussion, implementation, and active) and identified and quantified patterns over time. Our results, the first of their kind, may be used as a first step towards software explanation, guiding documentation and training efforts, and monitoring project status.

9.1.2 Using Advanced IR Techniques on Software Repositories

Proposing and evaluating a framework for combining the results of disparate IR models. Within the context of bug localization, our framework can combine the results of any

IR model. We evaluated two different combination strategies: one based on the relevancy score of a document in the result set, and one based on the rank of a document in the result set. We found that the rank provides better performance that the relevancy score. Further, based on two experiments of combining individual models, we found that combining results almost always achieves better performance than the best individual model. Our framework can be used to significantly enhance bug localization performance.

Describing and evaluating an advanced IR technique to analyze source code histories. Using the Hall topic evolution model applied jointly to the linguistic data in several versions of source code, we approximated the conceptual evolution of the source code. We manually analyzed the resulting topic evolutions, comparing them with system release notes and commit messages, and determined that the evolutions correspond well (87–89%) with the actual changes to source code. Our results demonstrate the benefit of applying advanced IR models to unstructured software repositories.

9.1.3 Going Beyond the Black Box

Proposing and evaluating a technique that overcomes the data duplication problem in large source code histories. When versions of a repository are mostly identical, as is the case with source code versions, the assumptions made by off-the-shelf topic evolution models no longer hold. We proposed the *Diff* model to account for data duplication by removing any redundancy from the versions before applying a topic evolution model. Case studies found that the Diff model can infer more distinct topics as well as more sensitive and accurate topic evolutions, which are both useful for analyzing large source code histories.

Analyzing the sensitivity of IR models to data preprocessing and IR model parameters. We performed a large-scale empirical study to analyze the effects of data preprocessing and IR model parameters in the context of bug localization. We found that both matter greatly: a “good” IR model configuration can be 2–50 times more effective than a “bad” configuration. We also found that the VSM model achieves the best performance; both the bug report’s title and description are useful; using all source code linguistic data and past bug history is best;

and that stopping, stemming, and splitting identifiers results in the best performance.

9.2 Future Research Opportunities

The promising results presented in this thesis can be extended in several exciting ways.

Underused Repositories In Chapter 3, we showed that applying an IR model to a previously-unused repository, in this case a test suite, yielded new capabilities for developers. There remain several other software repositories that require more attention. For example, email archives, execution logs, and bug reports have rarely been studied, even though they are rich with information about a software system.

Underexplored Software Engineering Tasks Bug prediction, searching collections of software systems, and measuring the evolutionary trends of repositories are all underexplored tasks in the literature. In addition, traceability links are typically established between requirements documents and source code, although it would be also be useful to find links between other repositories, such as emails and source code, and between source code documents themselves. We have taken a first step towards establishing links between email and source code in Chapter 4, but additional work is required to provide clean, high-accuracy links, and to fully understand their interaction.

Additional IR Models The variants of LDA listed in Section 2.2 have promising features that may directly improve the results of several software engineering tasks. For example, the correlated topic model, which models dependencies between topics, may allow sets of dependent topics in need of refactoring to be found in the source code. Additionally, the cross-collection topic model might allow similar topics to be discovered from the source code of related systems, such as Mozilla Firefox and Google Chrome. In addition, lightweight models such as BM25 or BM25F may be useful for bug localization or test case prioritization.

Data Preprocessing and IR Model Parameters Now that we have shown the importance of data preprocessing and IR models parameters, previous studies from the literature can be performed again using our findings as a guideline.

Additional Preprocessing Steps In Chapter 8, we investigated three popular preprocessing steps: identifier splitting, stop word removal, and word stemming. A preprocessing step that is currently less popular, but may also provide benefits, is query expansion (Carpineto and Romano, 2012), i.e., automatically fixing spelling errors, finding synonyms, or using or WordNet (Miller, 1995) to find related concepts and themes. Query expansion can be applied, for example, to our bug localization dataset to reduce noise in the bug reports, and to help expand short or vague bug reports to provide more context. In addition, most preprocessing steps treat all words as equals, independent of their context. Considering context might allow the opportunity to give higher weights to important terms, technical terms, or system-specific terms. For example, it may be fruitful for the preprocessor to determine whether a bug report has an embedded code snippet and use this context to preserve identifier names in their entirety, so as to maximize the chance of linking the bug report to relevant source code entities that contain the same identifier names.

User Studies We have implemented research prototypes to evaluate our ideas throughout this thesis. With more sophisticated implementations, user studies can be conducted in the hands of practitioners to learn even more about how IR models can be useful in practice, and help guide future research in this area.

Treating Software as Natural Language Recent work by Hindle et al. (2012) has compared source code to natural language: both are created by humans, and while any given instance of either could theoretically be very complex, most of the time the instances are quite simple. The authors show that source code is indeed "natural", in that it is highly repetitive and predictable. As a consequence, models that deal with source code text, such as the ones presented in this thesis, can use this fact to construct more intelligent models.

Bibliography

B. Adams, Z. M. Jiang, and A. E. Hassan. Identifying crosscutting concerns using historical code changes. In *Proceedings of the 32nd International Conference on Software Engineering*, pages 305–314, 2010. [98]

S. N. Ahsan, J. Ferzund, and F. Wotawa. Automatic software bug triage system (BTS) based on Latent Semantic Indexing and Support Vector Machine. In *Proceedings of the 4th International Conference on Software Engineering Advances*, pages 216–221, 2009. [43, 46, 53, 55]

S. Ali, L. C. Briand, H. Hemmati, and R. K. Panesar-Walawege. A systematic review of the application and empirical investigation of search-based test case generation. *IEEE Transactions on Software Engineering*, 36(6):742–762, 2009. [61]

D. Andrzejewski, A. Mulhern, B. Liblit, and X. Zhu. Statistical debugging using latent topic models. In *Proceedings of the 18th European Conference on Machine Learning*, pages 6–17, 2007. [39, 50, 53, 55]

G. Anthes. Topic models vs. unstructured data. *Communications of the ACM*, 53:16–18, Dec. 2010. [22]

G. Antoniol, J. H. Hayes, Y. G. Gueheneuc, and M. Di Penta. Reuse or rewrite: Combining textual, static, and dynamic analyses to assess the cost of keeping a system up-to-date. In *Proceedings of the 24th International Conference on Software Maintenance*, pages 147–156, 2008. [7, 36, 53, 55]

Apache Foundation. HTTP Server. Accessed Aug. 8, 2012, 2012a. `http://httpd.apache.org`. [105]

Apache Foundation. Ant. Accessed Aug. 8, 2012, 2012b. `http://ant.apache.org`. [74]

Apache Foundation. Apache. Accessed Aug. 8, 2012, 2012c. `http://www.apache.org`. [74]

Apache Foundation. Derby. Accessed Aug. 8, 2012, 2012d. `http://db.apache.org/derby`. [74]

A. Arcuri and L. Briand. A practical guide for using statistical tests to assess randomized algorithms in software engineering. In *Proceedings of the 33rd International Conference on Software Engineering*, pages 1–10, 2011. [78]

H. U. Asuncion, A. U. Asuncion, and R. N. Taylor. Software traceability with topic modeling. In *Proceedings of the 32nd International Conference on Software Engineering*, pages 95–104, 2010. [7, 37, 53, 55]

A. Bacchelli, M. D'Ambros, M. Lanza, and R. Robbes. Benchmarking lightweight techniques to link e-mails and source code. In *Proceedings of the 16th Working Conference on Reverse Engineering*, pages 205–214, 2009. [98]

A. Bacchelli, M. Lanza, and V. Humpa. RTFM (Read The Factual Mails)-Augmenting program comprehension with Remail. In *Proceedings of the 15th European Conference on Software Maintenance and Reengineering*, pages 15–24, 2011. [117]

A. Bachmann, C. Bird, F. Rahman, P. T. Devanbu, and A. Bernstein. The missing links: Bugs and bug-fix commits. In *Proceedings of the 18th International Symposium on the Foundations of Software Engineering*, pages 97–106, 2010. [142]

R. Baeza-Yates and B. Ribeiro-Neto. *Modern information retrieval*, volume 463. ACM press New York, 1999. [23]

S. Bajracharya and C. Lopes. Mining search topics from a code search engine usage log. In *Proceedings of the 6th International Working Conference on Mining Software Repositories*, pages 111–120, 2009. [43, 47, 53, 55]

S. K. Bajracharya and C. V. Lopes. Analyzing and mining a code search engine usage log. *Empirical Software Engineering*, pages 1–43, Sept. 2010. [7, 43, 47, 53, 55]

P. F. Baldi, C. V. Lopes, E. J. Linstead, and S. K. Bajracharya. A theory of aspects as latent topics. *ACM SIGPLAN Notices*, 43(10):543–562, 2008. [33, 46, 53, 55, 152, 153, 154, 176]

K. Barnard, P. Duygulu, D. Forsyth, N. De Freitas, D. M. Blei, and M. I. Jordan. Matching words and pictures. *The Journal of Machine Learning Research*, 3:1107–1135, 2003. [7]

D. J. Bartholomew. *Latent variable models and factors analysis*. Oxford University Press, Inc., New York, NY, USA, 1987. [26]

G. Bavota, A. De Lucia, A. Marcus, and R. Oliveto. A two-step technique for extract class refactoring. In *Proceedings of the 25th International Conference on Automated Software Engineering*, pages 151–154, 2010. [7, 38, 53, 55]

O. Baysal and A. J. Malton. Correlating social interactions to release history during software evolution. In *Proceedings of the 4th International Workshop on Mining Software Repositories*, 2007. [98]

S. Bellon, R. Koschke, G. Antoniol, J. Krinke, and E. Merlo. Comparison and evaluation of clone detection tools. *IEEE Transactions on Software Engineering*, pages 577–591, 2007. [43]

B. Berliner. CVS II: Parallelizing software development. In *Proceedings of the USENIX Winter 1990 Technical Conference*, volume 341, page 352, 1990. [15]

N. Bettenburg and B. Adams. Workshop on Mining Unstructured Data (MUD) because "Mining Unstructured Data is like fishing in muddy waters"! In *Proceedings of the 17th Working Conference on Reverse Engineering*, pages 277–278, 2010. [16]

N. Bettenburg, E. Shihab, and A. E. Hassan. An empirical study on the risks of using off-the-shelf techniques for processing mailing list data. In *Proceedings of the 25th International Conference on Software Maintenance*, 2009. [15, 101, 118]

D. Binkley, M. Ceccato, M. Harman, F. Ricca, and P. Tonella. Tool-supported refactoring of existing object-oriented code into aspects. *IEEE Transactions on Software Engineering*, pages 698–717, 2006. [149]

C. Bird, A. Bachmann, E. Aune, J. Duffy, A. Bernstein, V. Filkov, and P. Devanbu. Fair and balanced?: Bias in bug-fix datasets. In *Proceedings of the 7th European Software Engineering Conference and the Symposium on the Foundations of Software Engineering*, pages 121–130, 2009a. [142]

C. Bird, P. C. Rigby, E. T. Barr, D. J. Hamilton, D. M. Germn, and P. T. Devanbu. The promises and perils of mining Git. In *Proceedings of the 6th Working Conference on Mining Software Repositories*, pages 1–10, 2009b. [15]

C. M. Bishop. Latent variable models. *Learning in graphical models*, 1998. [26]

D. M. Blei and J. D. Lafferty. Dynamic topic models. In *Proceedings of the 23rd international conference on Machine learning*, pages 113–120. ACM, 2006. [27]

D. M. Blei and J. D. Lafferty. Topic models. In *Text Mining: Classification, Clustering, and Applications*, pages 71–94. Chapman & Hall, London, UK, 2009. [6, 22, 25, 71, 117]

D. M. Blei, A. Y. Ng, and M. I. Jordan. Latent Dirichlet allocation. *Journal of Machine Learning Research*, 3:993–1022, 2003. [6, 7, 22, 25, 26, 76]

D. M. Blei, T. L. Griffiths, and M. I. Jordan. The nested chinese restaurant process and bayesian nonparametric inference of topic hierarchies. *Journal of the ACM*, 57(2):1–30, 2010. [190]

R. Blumberg and S. Atre. The problem with unstructured data. *DM Review*, 13:42–49, 2003. [6]

J. C. Bose and U. Suresh. Root cause analysis using sequence alignment and Latent Semantic Indexing. In *Proceedings of the 19th Australian Conference on Software Engineering*, pages 367–376, 2008. [39, 53, 55]

A. W. J. Bradley and G. C. Murphy. Supporting software history exploration. In *Proceedings of the 8th Working Conference on Mining Software Repositories*, pages 193–202, 2011. [146]

G. Capobianco, A. De Lucia, R. Oliveto, A. Panichella, and S. Panichella. Traceability recovery using numerical analysis. In *Proceedings of the 16th Working Conference on Reverse Engineering*, pages 195–204, 2009. [37, 53, 55]

C. Carpineto and G. Romano. A survey of automatic query expansion in information retrieval. *ACM Computing Surveys*, 44(1):1–50, Jan. 2012. [143, 225]

J. Chang. lda: Collapsed Gibbs sampling methods for topic models. Accessed Aug. 8, 2012, 2012. `http://cran.r-project.org/web/packages/lda`. [76]

J. Chang and D. M. Blei. Relational topic models for document networks. *Proceedings of the 12th International Conference on Artificial Intelligence and Statistics*, pages 81–88, 2009. [143]

S. Chen, Z. Chen, Z. Zhao, B. Xu, and Y. Feng. Using semi-supervised clustering to improve regression test selection techniques. In *Proceedings of the 4th International Conference on Software Testing, Verification and Validation*, pages 1–10, 2011. [62]

B. Cleary, C. Exton, J. Buckley, and M. English. An empirical analysis of information retrieval based concept location techniques in software comprehension. *Empirical Software Engineering*, 14(1):93–130, 2008. [7, 32, 53, 55]

P. Comon. Independent component analysis, a new concept? *Signal processing*, 36(3): 287–314, 1994. [24]

J. R. Cordy. The TXL source transformation language. *Science of Computer Programming*, 61(3):190–210, 2006. [73]

G. V. Cormack, C. L. Clarke, and S. Buettcher. Reciprocal Rank Fusion outperforms Condorcet and individual rank learning methods. In *Proceedings of the 32nd International Conference on Research and Development in Information Retrieval*, pages 758–759, 2009. [126, 143]

T. M. Cover and J. A. Thomas. *Elements of information theory*. John Wiley and sons, 2006. [160]

V. Dallmeier and T. Zimmermann. Extraction of bug localization benchmarks from history. In *Proceedings of the 22nd International Conference on Automated Software Engineering*, pages 433–436, 2007. [37]

M. D'Ambros, M. Lanza, and R. Robbes. Evaluating defect prediction approaches: A benchmark and an extensive comparison. *Empirical Software Engineering*, 17(4), 2012. [129]

B. de Alwis and G. C. Murphy. Answering conceptual queries with Ferret. In *Proceedings of the 30th International Conference on Software Engineering*, pages 21–30, 2008. [146]

R. C. de Boer and H. van Vliet. Architectural knowledge discovery with Latent Semantic Analysis: constructing a reading guide for software product audits. *Journal of Systems and Software*, 81(9):1456–1469, 2008. [7, 35, 47, 53, 55]

A. De Lucia, F. Fasano, R. Oliveto, and G. Tortora. Enhancing an artefact management system with traceability recovery features. In *Proceedings of the 20th International Conference on Software Maintenance*, pages 306–315, 2004. [7, 34, 53, 55]

A. De Lucia, F. Fasano, R. Oliveto, and G. Tortora. Can information retrieval techniques effectively support traceability link recovery? In *Proceedings of the 14th International Conference on Program Comprehension*, pages 307–316, 2006. [34, 53, 55]

A. De Lucia, F. Fasano, R. Oliveto, and G. Tortora. Recovering traceability links in software artifact management systems using information retrieval methods. *ACM Transactions on Software Engineering and Methodology*, 16(4), 2007. [7, 35, 53, 55]

S. Deerwester, S. T. Dumais, G. W. Furnas, T. K. Landauer, and R. Harshman. Indexing by Latent Semantic Analysis. *Journal of the American Society for Information Science*, 41(6): 391–407, 1990. [7, 23]

B. Dit, D. Poshyvanyk, and A. Marcus. Measuring the semantic similarity of comments in bug reports. In *Proceedings 1st International Workshop on Semantic Technologies in System Maintenance*, 2008. [43, 47, 53, 55]

H. Do, S. Elbaum, and G. Rothermel. Supporting controlled experimentation with testing techniques: An infrastructure and its potential impact. *Empirical Software Engineering*, 10(4):405–435, 2005. [74, 75, 92]

Eclipse Foundation. Eclipse. Accessed Aug. 8, 2012, 2011. http://www.eclipse.org. [130]

Edgewall Software. The Trac project. Accessed Aug. 8, 2012, 2012. http://trac.edgewall.org. [15]

S. Elbaum, A. Malishevsky, and G. Rothermel. Test case prioritization: A family of empirical studies. *IEEE Transactions on Software Engineering*, 28(2):159–182, 2002. [62, 66, 76, 83]

L. Erlikh. Leveraging legacy system dollars for e-business. *IT Professional*, 2(3):17–23, 2000. [5]

R. Feldt, R. Torkar, T. Gorschek, and W. Afzal. Searching for cognitively diverse tests: Towards universal test diversity metrics. In *Proceedings of the International Conference on Software Testing Verification and Validation Workshop*, pages 178–186, 2008. [66]

M. Fowler and K. Scott. *UML distilled: A brief guide to the standard object modeling language*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2000. ISBN 020165783X. [16]

T. Fritz and G. C. Murphy. Using information fragments to answer the questions developers ask. In *Proceedings of the 32nd International Conference on Software Engineering*, pages 175–184, 2010. [146]

C. S. Gall, S. Lukins, L. Etzkorn, S. Gholston, P. Farrington, D. Utley, J. Fortune, and S. Virani. Semantic software metrics computed from natural language design specifications. *Software, IET*, 2(1):17–26, 2008. [7, 38, 53, 55]

E. Gamma. JHotDraw. Accessed Aug. 8, 2012, 2012. `http://www.jhotdraw.org`. [26, 149, 186, 191]

Geeknet. SourceForge. Accessed Aug. 8, 2012, 2012. `http://sourceforge.net`. [16]

M. Gethers and D. Poshyvanyk. Using relational topic models to capture coupling among classes in object-oriented software systems. In *Proceedings of the 26th International Conference on Software Maintenance*, pages 1–10, 2010. [7, 39, 53, 55]

R. L. Glass. *Facts and fallacies of software engineering*. Addison-Wesley Professional, 2003. [5]

GNU. Diffutils. Accessed Aug. 8, 2012, 2010. `http://www.gnu.org/software/diffutils`. [188]

M. W. Godfrey, A. E. Hassan, J. Herbsleb, G. C. Murphy, M. Robillard, P. Devanbu, A. Mockus, D. E. Perry, and D. Notkin. Future of mining software archives: A roundtable. *IEEE Software*, 26(1):67–70, 2008. [5, 13, 14]

Google. Google code. Accessed Aug. 8, 2012, 2012. `http://code.google.com`. [16]

S. Grant and J. R. Cordy. Vector space analysis of software clones. In *Proceedings of the 17th International Conference on Program Comprehension*, pages 233–237, 2009. [24, 43, 47, 53, 55]

S. Grant and J. R. Cordy. Estimating the optimal number of latent concepts in source code analysis. In *Proceedings of the 10th International Working Conference on Source Code Analysis and Manipulation*, pages 65–74, 2010. [44, 47, 53, 55, 93]

S. Grant, J. R. Cordy, and D. Skillicorn. Automated concept location using independent component analysis. In *Proceedings of the 15th Working Conference on Reverse Engineering*, pages 138–142, 2008. [7, 32, 53, 55]

T. L. Griffiths and M. Steyvers. Finding scientific topics. *Proceedings of the National Academy of Sciences*, 101:5228–5235, 2004. [26, 93, 106, 150, 192]

T. L. Griffiths, M. Steyvers, and J. B. Tenenbaum. Topics in semantic representation. *Psychological Review*, 114(2):211–244, 2007. [6, 7, 76]

S. Grimes. Unstructured data and the 80 percent rule. *Clarabridge Bridgepoints*, 2008. [6]

D. Hall, D. Jurafsky, and C. D. Manning. Studying the history of ideas using topic models. In *Proceedings of the Conference on Empirical Methods in Natural Language Processing*, pages 363–371. ACL, 2008. [27]

A. E. Hassan. *Mining Software Repositories to Assist Developers and Support Managers*. PhD thesis, University of Waterloo, Waterloo, ON, Canada, 2004. [13]

A. E. Hassan. The road ahead for mining software repositories. In *Frontiers of Software Maintenance*, pages 48–57, 2008. [5, 6, 13, 14]

A. E. Hassan. Predicting faults using the complexity of code changes. In *Proceedings of the 31st International Conference on Software Engineering*, pages 78–88, 2009. [5]

A. E. Hassan and R. C. Holt. The top ten list: Dynamic fault prediction. In *Proceedings of the 21st International Conference on Software Maintenance*, pages 263–272, 2005. [13]

A. E. Hassan and T. Xie. Software intelligence: The future of mining software engineering data. In *Proceedings of the FSE/SDP workshop on Future of Software Engineering Research*, pages 161–166, 2010. [14]

J. H. Hayes, A. Dekhtyar, and S. K. Sundaram. Advancing candidate link generation for requirements tracing: The study of methods. *IEEE Transactions on Software Engineering*, pages 4–19, 2006. [7, 35, 53, 55]

H. Hemmati, A. Arcuri, and L. Briand. Reducing the cost of model-based testing through test case diversity. In *Proceedings of the 22nd International Conference on Testing Software and Systems*, pages 63–78, 2010a. [64, 66, 67, 68]

H. Hemmati, L. Briand, A. Arcuri, and S. Ali. An enhanced test case selection approach for model-based testing: An industrial case study. In *Proceedings of the 18th International Symposium on Foundations of Software Engineering*, pages 267–276, 2010b. [63, 66, 67, 68]

H. Hemmati, A. Arcuri, and L. Briand. Empirical investigation of the effects of test suite properties on similarity-based test case selection. In *Proceedings of the 4th International Conference on Software Testing, Verification and Validation*, pages 327–336, 2011. [63]

H. Hemmati, L. Briand, and A. Arcuri. Achieving scalable model-based testing through test case diversity. *ACM Transactions on Software Engineering and Methodology, to appear in 22(1)*, 2012. [62, 66, 67, 87]

A. Hindle, M. W. Godfrey, and R. C. Holt. What's hot and what's not: Windowed developer topic analysis. In *Proceedings of the 25th International Conference on Software Maintenance*, pages 339–348, 2009. [28, 41, 53, 55]

A. Hindle, M. W. Godfrey, and R. C. Holt. Software process recovery using recovered unified process views. In *Proceedings of the 26th International Conference on Software Maintenance*, pages 1–10, 2010. [41, 53, 55]

A. Hindle, E. T. Barr, Z. Su, M. Gabel, and P. T. Devanbu. On the naturalness of software. In *Proceedings of the 34th International Conference on Software Engineering*, pages 837–847, 2012. [225]

T. Ho, J. Hull, and S. Srihari. Decision combination in multiple classifier systems. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 16(1):66–75, 1994. [126]

T. Hofmann. Probabilistic Latent Semantic Indexing. In *Proceedings of the 22nd International Conference on Research and Development in Information Retrieval*, pages 50–57, 1999. [24, 25]

T. Hofmann. Unsupervised learning by probabilistic Latent Semantic Analysis. *Machine Learning*, 42(1):177–196, 2001. [24, 25]

IBM. Jazz. Accessed Aug. 8, 2012, 2012. http://www-01.ibm.com/software/rational/jazz. [131]

R. Ihaka and R. Gentleman. R: A language for data analysis and graphics. *Journal of Computational and Graphical Statistics*, pages 299–314, 1996. [76, 151]

B. Jiang, Z. Zhang, W. Chan, and T. Tse. Adaptive random test case prioritization. In *Proceedings of the 24th International Conference on Automated Software Engineering*, pages 233–244, 2009. [66]

H. Jiang, T. N. Nguyen, I. Chen, H. Jaygarl, and C. Chang. Incremental Latent Semantic Indexing for automatic traceability link evolution management. In *Proceedings of the 23rd International Conference on Automated Software Engineering*, pages 59–68, 2008. [7, 35, 53, 55]

J. Jones and M. Harrold. Test-suite reduction and prioritization for modified condition/decision coverage. *IEEE Transactions on Software Engineering*, 29(3):195–209, 2003. [66]

H. Kagdi, M. Gethers, D. Poshyvanyk, and M. Collard. Blending conceptual and evolutionary couplings to support change impact analysis in source code. In *Proceedings of the 17th Working Conference on Reverse Engineering*, pages 119–128, 2010. [7, 38, 53, 55]

Y. Kamei, S. Matsumoto, A. Monden, K. Matsumoto, B. Adams, and A. E. Hassan. Revisiting common bug prediction findings using effort-aware models. In *Proceedings of the 26th International Conference on Software Maintenance*, pages 1–10, 2010. [37]

S. Kawaguchi, P. K. Garg, M. Matsushita, and K. Inoue. Mudablue: An automatic categorization system for open source repositories. *Journal of Systems and Software*, 79(7): 939–953, 2006. [42, 53, 55]

J. Kittler, M. Hatef, R. P. Duin, and J. Matas. On combining classifiers. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 20(3):226–239, 1998. [126]

B. Korel, G. Koutsogiannakis, and L. Tahat. Model-based test prioritization heuristic methods and their evaluation. In *Proceedings of the 3rd International Workshop on Advances in Model-based Testing*, pages 34–43, 2007. [62, 66, 67]

R. Kuehl. *Design of experiments: Statistical principles of research design and analysis*. Brooks/Cole, 2000. [209]

A. Kuhn, S. Ducasse, and T. Girba. Enriching reverse engineering with semantic clustering. In *Proceedings of the 12th Working Conference on Reverse Engineering*, pages 133–142, 2005. [7, 41, 53, 55]

A. Kuhn, S. Ducasse, and T. Girba. Semantic clustering: Identifying topics in source code. *Information and Software Technology*, 49(3):230–243, 2007. [7, 14, 41, 53, 55, 63, 70, 71]

A. Kuhn, P. Loretan, and O. Nierstrasz. Consistent layout for thematic software maps. In *Proceedings of the 15th Working Conference on Reverse Engineering*, pages 209–218, 2008. [7, 42, 53, 55]

A. Kuhn, D. Erni, P. Loretan, and O. Nierstrasz. Software cartography: Thematic software visualization with consistent layout. *Journal of Software Maintenance and Evolution: Research and Practice*, pages 191–210, 2010. [7, 42, 53, 55]

S. Kullback and R. Leibler. On information and sufficiency. *The Annals of Mathematical Statistics*, 22(1):79–86, 1951. [71]

A. Kumar. Development at the speed and scale of Google. Presented at QCon 2010, San Francisco, CA, USA, 2010. [62]

Y. Ledru, A. Petrenko, and S. Boroday. Using string distances for test case prioritisation. In *Proceedings of the 24th International Conference on Automated Software Engineering*, pages 510–514, Nov. 2009. [66, 69]

Y. Ledru, A. Petrenko, S. Boroday, and N. Mandran. Prioritizing test cases with string distances. *Automated Software Engineering*, 19(1):65–95, 2011. [62, 66, 68, 69, 73, 75, 88]

M. M. Lehman. Programs, life cycles, and laws of software evolution. *Proceedings of the IEEE*, 68(9):1060–1076, 1980. [5, 40]

D. Leon and A. Podgurski. A comparison of coverage-based and distribution-based techniques for filtering and prioritizing test cases. In *Proceedings of the International Symposium on Software Reliability Engineering*, pages 442–456, 2003. [66]

T. C. Lethbridge, R. Laganiere, and C. King. *Object-oriented software engineering: practical software development using UML and Java*. McGraw-Hill, 2005. ISBN 0077109082. [14, 16]

M. Y. Lin, R. Amor, and E. Tempero. A Java reuse repository for Eclipse using LSI. In *Proceedings of the 2006 Australian Software Engineering Conference*, pages 351–362, 2006. [8, 41, 53, 55]

E. Linstead and P. Baldi. Mining the coherence of GNOME bug reports with statistical topic models. In *Proceedings of the 6th Working Conference on Mining Software Repositories*, pages 99–102, 2009. [7, 39, 53, 55]

E. Linstead, P. Rigor, S. Bajracharya, C. Lopes, and P. Baldi. Mining Eclipse developer contributions via author-topic models. In *Proceedings of the 4th International Workshop on Mining Software Repositories*, pages 30–33, 2007a. [33, 53, 55]

E. Linstead, P. Rigor, S. Bajracharya, C. Lopes, and P. Baldi. Mining concepts from code with probabilistic topic models. In *Proceedings of the 22nd International Conference on Automated Software Engineering*, pages 461–464, 2007b. [33, 53, 55]

E. Linstead, S. Bajracharya, T. Ngo, P. Rigor, C. Lopes, and P. Baldi. Sourcerer: mining and searching internet-scale software repositories. *Data Mining and Knowledge Discovery*, 18 (2):300–336, 2008a. [42, 53, 55]

E. Linstead, C. Lopes, and P. Baldi. An application of latent Dirichlet allocation to analyzing software evolution. In *Proceedings of the 7th International Conference on Machine Learning and Applications*, pages 813–818, 2008b. [27, 40, 53, 55, 146, 147]

E. Linstead, P. Rigor, S. Bajracharya, C. Lopes, and P. Baldi. Mining internet-scale software repositories. In *Advances in Neural Information Processing Systems*, volume 2007, pages 929–936, 2008c. [42, 53, 55]

E. Linstead, L. Hughes, C. Lopes, and P. Baldi. Software analysis with unsupervised topic models. In *NIPS Workshop on Application of Topic Models: Text and Beyond*, 2009. [53, 55]

Y. Liu, D. Poshyvanyk, R. Ferenc, T. Gyimothy, and N. Chrisochoides. Modeling class cohesion as mixtures of latent topics. In *Proceedings of the 25th International Conference on Software Maintenance*, pages 233–242, 2009. [7, 39, 53, 55]

J. C. Loehlin. *Latent variable models*. Erlbaum Hillsdale, NJ, 1987. ISBN 0898599636. [26]

F. Longo, R. Tiella, P. Tonella, and A. Villafiorita. Measuring the impact of different categories of software evolution. *Software Process and Product Measurement*, pages 344–351, 2008. [169]

M. Lormans. Monitoring requirements evolution using views. In *Proceedings of the 11th European Conference on Software Maintenance and Reengineering*, pages 349–352, 2007. [35, 54, 56]

M. Lormans and A. Van Deursen. Can LSI help reconstructing requirements traceability in design and test? In *Proceedings of 10th European Conference on Software Maintenance and Reengineering*, pages 47–56, 2006. [7, 35, 54, 56]

M. Lormans, H. G. Gross, A. van Deursen, and R. van Solingen. Monitoring requirements coverage using reconstructed views: An industrial case study. In *Proceedings of the 13th Working Conference on Reverse Engineering*, pages 275–284, 2006. [7, 35, 54, 56]

S. K. Lukins, N. A. Kraft, and L. H. Etzkorn. Source code retrieval for bug localization using latent Dirichlet allocation. In *Proceedings of the 15th Working Conference on Reverse Engineering*, pages 155–164, 2008. [36, 54, 56]

S. K. Lukins, N. A. Kraft, and L. H. Etzkorn. Bug localization using latent Dirichlet allocation. *Information and Software Technology*, 52(9):972–990, 2010. [36, 54, 56, 93, 125, 206, 207]

R. Madsen, S. Sigurdsson, L. Hansen, and J. Larsen. Pruning the vocabulary for better context recognition. In *Proceedings of the 17th International Conference on Pattern Recognition*, pages 483–488, 2004. [28]

J. I. Maletic and A. Marcus. Supporting program comprehension using semantic and structural information. In *Proceedings of the 23rd International Conference on Software Engineering*, pages 103–112, 2001. [8, 41, 54, 56]

J. I. Maletic and N. Valluri. Automatic software clustering via Latent Semantic Analysis. In *Proceeding of the 14th International Conference on Automated Software Engineering*, pages 251–254, 1999. [8, 41, 54, 56]

C. D. Manning, P. Raghavan, and H. Schutze. *Introduction to information retrieval*, volume 1. Cambridge University Press Cambridge, UK, 2008. [16, 17, 209]

A. Marcus. Semantic driven program analysis. In *Proceedings of the 20th International Conference on Software Maintenance*, pages 469–473, 2004. [54, 56]

A. Marcus and J. I. Maletic. Identification of high-level concept clones in source code. In *Proceedings of the 16th International Conference on Automated Software Engineering*, pages 107–114, 2001. [43, 47, 54, 56]

A. Marcus and J. I. Maletic. Recovering documentation-to-source-code traceability links using Latent Semantic Indexing. In *Proceedings of the 25th International Conference on Software Engineering*, pages 125–135, 2003. [7, 34, 54, 56]

A. Marcus, A. Sergeyev, V. Rajlich, and J. I. Maletic. An information retrieval approach to concept location in source code. In *Proceedings of the 11th Working Conference on Reverse Engineering*, pages 214–223, 2004. [7, 31, 54, 56]

A. Marcus, V. Rajlich, J. Buchta, M. Petrenko, and A. Sergeyev. Static techniques for concept location in object-oriented code. In *Proceedings of the 13th International Workshop on Program Comprehension*, pages 33–42, 2005. [7, 31, 54, 56]

A. Marcus, D. Poshyvanyk, and R. Ferenc. Using the conceptual cohesion of classes for fault prediction in object-oriented systems. *IEEE Transactions on Software Engineering*, 34(2): 287–300, 2008. [7, 38, 39, 54, 56]

G. Maskeri, S. Sarkar, and K. Heafield. Mining business topics in source code using latent Dirichlet allocation. In *Proceedings of the 1st conference on India software engineering conference*, pages 113–120, 2008. [33, 48, 54, 56, 71]

W. Masri, A. Podgurski, and D. Leon. An empirical study of test case filtering techniques based on exercising information flows. *IEEE Transactions on Software Engineering*, 33(7): 454–477, 2007. [66]

A. K. McCallum. MALLET: A machine learning for language toolkit. Accessed Aug. 8, 2012, 2012. `http://mallet.cs.umass.edu`. [30, 106, 150, 176, 192, 209]

S. McMaster and A. Memon. Call stack coverage for GUI test-suite reduction. *IEEE Transactions on Software Engineering*, 34(1):99–115, 2006. [66]

C. McMillan, D. Poshyvanyk, and M. Revelle. Combining textual and structural analysis of software artifacts for traceability link recovery. In *Proceedings of the ICSE Workshop on Traceability in Emerging Forms of Software Engineering*, pages 41–48, 2009. [7, 36, 54, 56]

C. McMillan, M. Grechanik, D. Poshyvanyk, Q. Xie, and C. Fu. Portfolio: Finding relevant functions and their usage. In *Proceedings of the 33rd International Conference on Software Engineering*, pages 111–120, 2011. [143]

H. Mei, D. Hao, L. Zhang, L. Zhang, J. Zhou, and G. Rothermel. A static approach to prioritizing JUnit test cases. *IEEE Transactions on Software Engineering*, 2011. [62, 66, 68]

Q. Mei and C. X. Zhai. Discovering evolutionary theme patterns from text: an exploration of temporal text mining. In *Proceedings of the 11th International Conference on Knowledge Discovery in Data Mining*, pages 198–207, 2005. [28]

G. A. Miller. WordNet: A lexical database for english. *Communications of the ACM*, 38(11): 39–41, 1995. [21, 225]

A. T. Misirli, A. B. Bener, and B. Turhan. An industrial case study of classifier ensembles for locating software defects. *Software Quality Journal*, 19(3):515–536, 2011. [50, 126]

J. Moad. Maintaining the competitive edge. *Datamation*, 36(4):61–66, 1990. [5]

Mozilla Foundation. Bugzilla. Accessed Aug. 8, 2012, 2012a. http://www.bugzilla.org. [15]

Mozilla Foundation. Mozilla. Accessed Aug. 8, 2012, 2012b. http://www.mozilla.org. [131]

S. Neuhaus and T. Zimmermann. Security trend analysis with CVE topic models. In *Proceedings of the 21st International Symposium on Software Reliability Engineering*, pages 111–120, 2010. [15, 41, 54, 56]

A. T. Nguyen, T. T. Nguyen, J. Al-Kofahi, H. V. Nguyen, and T. N. Nguyen. A topic-based approach for narrowing the search space of buggy files from a bug report. In *Proceedings of the 26th International Conference on Automated Software Engineering*, pages 263–272, 2011. [36, 136, 143, 206, 207, 208]

T. H. Nguyen, B. Adams, and A. E. Hassan. A case study of bias in bug-fix datasets. In *Proceedings of the 17th Working Conference on Reverse Engineering*, pages 259–268, 2010. [38, 142]

R. Oliveto, M. Gethers, D. Poshyvanyk, and A. De Lucia. On the equivalence of information retrieval methods for automated traceability link recovery. In *Proceedings of the 18th International Conference on Program Comprehension*, pages 68–71, 2010. [37, 54, 56]

J. Ossher, S. Bajracharya, E. Linstead, P. Baldi, and C. Lopes. Sourcererdb: An aggregated repository of statically analyzed and cross-linked open source java projects. In *Proceedings of the 6th Working Conference on Mining Software Repositories*, pages 183–186, 2009. [54, 56]

L. Page, S. Brin, R. Motwani, and T. Winograd. The PageRank citation ranking: Bringing order to the web. *Stanford InfoLab*, 1999. [143]

D. E. Perry and A. L. Wolf. Foundations for the study of software architecture. *ACM SIGSOFT Software Engineering Notes*, 17(4):40–52, 1992. [146]

S. Pestov. jEdit. Accessed Aug. 8, 2012, 2012. http://www.jedit.org. [149]

X. H. Phan, L. M. Nguyen, and S. Horiguchi. Learning to classify short and sparse text & web with hidden topics from large-scale data collections. In *Proceeding of the 17th International Conference on World Wide Web*, pages 91–100, 2008. [49]

C. M. Pilato, B. Collins-Sussman, and B. W. Fitzpatrick. *Version Control with Subversion*. O'Reilly Media, 2008. ISBN 0596510330. [15]

I. Porteous, D. Newman, A. Ihler, A. Asuncion, P. Smyth, and M. Welling. Fast collapsed Gibbs sampling for latent Dirichlet allocation. In *Proceeding of the 14th International Conference on Knowledge Discovery and Data Mining*, pages 569–577, 2008. [7, 76, 87]

M. Porter. An algorithm for suffix stripping. *Program*, 14:130, 1980. [28]

D. Poshyvanyk and M. Grechanik. Creating and evolving software by searching, selecting and synthesizing relevant source code. In *Proceedings of the 31st International Conference on Software Engineering*, pages 283–286, 2009. [42, 47, 54, 56]

D. Poshyvanyk and A. Marcus. Combining formal concept analysis with information retrieval for concept location in source code. In *Proceedings of the 15th International Conference on Program Comprehension*, pages 37–48, 2007. [7, 32, 54, 56, 143]

D. Poshyvanyk, A. Marcus, V. Rajlich, et al. Combining probabilistic ranking and Latent Semantic Indexing for feature identification. In *Proceedings of the 14th International Conference on Program Comprehension*, pages 137–148, 2006. [7, 32, 54, 56]

D. Poshyvanyk, Y. Gueheneuc, A. Marcus, G. Antoniol, and V. Rajlich. Feature location using probabilistic ranking of methods based on execution scenarios and information retrieval. *IEEE Transactions on Software Engineering*, 33(6):420–432, 2007. [29, 49, 54, 56, 143]

PostgreSQL. Accessed Aug. 8, 2012, 2012. `http://www.postgresql.org`. [105, 186, 191]

F. Rahman, C. Bird, and P. T. Devanbu. Clones: what is that smell? *Empirical Software Engineering*, 17(4-5):503–530, 2012. [43]

V. Rajlich and N. Wilde. The role of concepts in program comprehension. In *Proceedings of the 10th International Workshop on Program Comprehension*, pages 271–278, 2002. ISBN 0769514952. [31]

D. Ramage, E. Rosen, J. Chuang, C. D. Manning, and D. A. McFarland. Topic modeling for the social sciences. In *NIPS 2009 Workshop on Applications for Topic Models: Text and Beyond*, 2009. [7]

M. K. Ramanathan, M. Koyuturk, A. Grama, and S. Jagannathan. PHALANX: A graph-theoretic framework for test case prioritization. In *Proceedings of the 23rd ACM Symposium on Applied Computing*, pages 667–673, 2008. [66]

S. Rao and A. Kak. Retrieval from software libraries for bug localization: a comparative study of generic and composite text models. In *Proceeding of the 8th Working Conference on Mining Software Repositories*, pages 43–52, 2011. [37, 206, 207, 211]

M. Revelle and D. Poshyvanyk. An exploratory study on assessing feature location techniques. In *Proceedings of the 17th International Conference on Program Comprehension*, pages 218–222, 2009. [32, 54, 56]

M. Revelle, B. Dit, and D. Poshyvanyk. Using data fusion and web mining to support feature location in software. In *Proceedings of the 18th International Conference on Program Comprehension*, pages 14–23, 2010. [7, 33, 54, 56, 143]

S. Robertson, H. Zaragoza, and M. Taylor. Simple BM25 extension to multiple weighted fields. In *Proceedings of the 13th International Conference on Information and Knowledge Management*, pages 42–49, 2004. [143]

M. Robillard and G. Murphy. Concern graphs: Finding and describing concerns using structural program dependencies. In *Proceedings of the 24th International Conference on Software Engineering*, pages 406–416, 2002. [98]

M. P. Robillard and G. C. Murphy. Representing concerns in source code. *ACM Transactions on Software Engineering and Methodology*, 16(1), Feb. 2007. [149]

G. Robles, J. Gonzalez-Barahona, D. Izquierdo-Cortazar, and I. Herraiz. Tools for the study of the usual data sources found in libre software projects. *International Journal of Open Source Software and Processes*, 1(1):24–45, 2009. [101]

M. Rosen-Zvi, T. Griffiths, M. Steyvers, and P. Smyth. The author-topic model for authors and documents. In *Proceedings of the 20th conference on Uncertainty in artificial intelligence*, pages 487–494, 2004. ISBN 0974903906. [33]

G. Rothermel, R. Untch, C. Chu, and M. Harrold. Prioritizing test cases for regression testing. *IEEE Transactions on Software Engineering*, 27(10):929–948, 2001. [61, 62, 63, 66, 73, 78, 87]

G. Rothermel, M. Harrold, J. Von Ronne, and C. Hong. Empirical studies of test-suite reduction. *Software Testing, Verification and Reliability*, 12(4):219–249, 2002. [85]

C. K. Roy, J. R. Cordy, and R. Koschke. Comparison and evaluation of code clone detection techniques and tools: A qualitative approach. *Science of Computer Programming*, 74(7): 470–495, 2009. [43]

G. Salton and M. J. McGill. Introduction to modern information retrieval. *New York*, 1983. [23]

G. Salton, A. Wong, and C. S. Yang. A vector space model for automatic indexing. *Communications of the ACM*, 18(11):620, 1975. [7, 21]

S. Sampath, R. C. Bryce, G. Viswanath, V. Kandimalla, and A. G. Koru. Prioritizing user-session-based test cases for web applications testing. In *Proceedings of the 1st International Conference on Software Testing, Verification, and Validation*, pages 141–150, 2008. [66, 67]

T. Savage, B. Dit, M. Gethers, and D. Poshyvanyk. TopicXP: Exploring topics in source code using latent Dirichlet allocation. In *Proceedings of the 26th International Conference on Software Maintenance*, pages 1–6, 2010. [34, 54, 56]

R. L. Scheaffer and J. T. McClave. *Probability and statistics for engineers*. Duxbury Press Boston, Massachusetts, USA, 1994. [162]

R. W. Selby. Enabling reuse-based software development of large-scale systems. *IEEE Transactions on Software Engineering*, 31(6):495–510, 2005. [5]

N. Serrano and I. Ciordia. Bugzilla, ITracker, and other bug trackers. *Software, IEEE*, 22(2): 11–13, 2005. [14]

C. E. Shannon. A mathematical theory of communication. *ACM SIGMOBILE Mobile Computing and Communications Review*, 5(1):3–55, 2001. [154]

E. Shihab, Z. M. Jiang, and A. E. Hassan. Studying the use of developer irc meetings in open source projects. In *Proceedings of the 25th International Conference on Software Maintenance*, 2009a. [15]

E. Shihab, Z. M. Jiang, and A. E. Hassan. On the use of IRC channels by developers of the GNOME GTK+ open source project. In *Proceedings of the 6th IEEE Working Conference on Mining Software Repositories*, 2009b. [15]

E. Shihab, N. Bettenburg, B. Adams, and A. Hassan. On the central role of mailing lists in open source projects: An exploratory study. *New Frontiers in Artificial Intelligence*, 6284: 91–103, 2010a. [15]

E. Shihab, Z. M. Jiang, W. M. Ibrahim, B. Adams, and A. E. Hassan. Understanding the impact of code and process metrics on post-release defects: a case study on the eclipse project. In *Proceedings of the International Symposium on Empirical Software Engineering and Measurement*, 2010b. [38]

A. Simao, R. F. de Mello, and L. J. Senger. A technique to reduce the test case suites for regression testing based on a self-organizing neural network architecture. In *Proceedings of the 30th Annual International Computer Software and Applications Conference*, pages 93–96, 2006. [62, 66]

J. Sliwerski, T. Zimmerman, and A. Zeller. When do changes induce fixes? In *Proceedings of the 2nd Working Conference on Mining Software Repositories*, 2005. [134, 142]

Software Freedom Conservancy. Git. Accessed Aug. 8, 2012, 2012. http://www.git-scm.com. [15]

M. Steyvers and T. Griffiths. Probabilistic topic models. In *Latent Semantic Analysis: A Road to Meaning*. Laurence Erlbaum, 2007. [22, 26, 160]

S. W. Thomas. Mining software repositories using topic models. In *Proceedings of the 33rd International Conference on Software Engineering*, pages 1138–1139, 2011. [5]

S. W. Thomas. Replication package. Accessed Dec. 3, 2012, 2012. http://sailhome.cs.queensu.ca/replication/sthomas/PHD2012. [11, 29, 84, 138, 211, 214]

S. W. Thomas, B. Adams, A. E. Hassan, and D. Blostein. DiffLDA: Topic evolution in software projects. Technical Report 2010-574, School of Computing, Queen's University, 2010a. [201]

S. W. Thomas, B. Adams, A. E. Hassan, and D. Blostein. Validating the use of topic models for software evolution. In *Proceedings of the 10th International Working Conference on Source Code Analysis and Manipulation*, pages 55–64, 2010b. [27, 40, 54, 56, 105, 145, 152, 176, 185, 189]

S. W. Thomas, B. Adams, A. E. Hassan, and D. Blostein. Modeling the evolution of topics in source code histories. In *Proceedings of the 8th Working Conference on Mining Software Repositories*, pages 173–182, 2011. [102, 178, 185]

S. W. Thomas, B. Adams, D. Blostein, and A. E. Hassan. Studying software evolution using topic models. *Science of Computer Programming*, pages 1–23, 2012a. [145]

S. W. Thomas, N. Bettenburg, D. Blostein, and A. E. Hassan. Talk and work: Recovering the relationship between mailing list discussions and development activity, 2012b. Submitted to *Empirical Software Engineering*. [97]

S. W. Thomas, H. Hemmati, A. E. Hassan, and D. Blostein. Static test case prioritization using topic models. *Empirical Software Engineering*, pages 1–31, 2012c. [61]

S. W. Thomas, M. Nagappan, D. Blostein, and A. E. Hassan. The impact of classifier configuration and classifier combination on bug localization, 2012d. Submitted to *IEEE Transactions on Software Engineering*. [125, 205]

K. Tian, M. Revelle, and D. Poshyvanyk. Using latent Dirichlet allocation for automatic categorization of software. In *Proceedings of the 6th International Working Conference on Mining Software Repositories*, pages 163–166, 2009. [7, 42, 54, 56]

W. Tichy. An interview with Prof. Andreas Zeller: Mining your way to software reliability. *Ubiquity*, 2010, Apr. 2010. URL http://doi.acm.org/10.1145/1880066.1883621. [5, 14]

J. Tukey. The philosophy of multiple comparisons. *Statistical Science*, 6(1):100–116, 1991. [210]

J. Tukey and H. Braun. *The collected works of John W. Tukey: Multiple comparisons, 1948-1983*, volume 8. Chapman & Hall/CRC, 1994. [210]

B. Ujhazi, R. Ferenc, D. Poshyvanyk, and T. Gyimothy. New conceptual coupling and cohesion metrics for object-oriented systems. In *Proceedings of the 10th International Working Conference on Source Code Analysis and Manipulation*, pages 33–42, 2010. [7, 38, 54, 56]

P. van der Spek, S. Klusener, and P. van de Laar. Towards recovering architectural concepts using Latent Semantic Indexing. In *Proceedings of the 12th European Conference on Software Maintenance and Reengineering*, pages 253–257, 2008. [7, 32, 54, 56]

M. Van Erp and L. Schomaker. Variants of the Borda Count method for combining ranked classifier hypotheses. In *Proceedings of the 7th International Workshop on Frontiers in Handwriting Recognition*, pages 443–452, 2000. [126, 128]

A. Vargha and H. D. Delaney. A critique and improvement of the CL common language effect size statistics of McGraw and Wong. *Journal of Educational and Behavioral Statistics*, 25 (2):101–132, 2000. [78]

H. M. Wallach, I. Murray, R. Salakhutdinov, and D. Mimno. Evaluation methods for topic models. In *Proceedings of the 26th International Conference on Machine Learning*, pages 1105–1112, 2009. [76, 92, 151]

X. Wang and A. McCallum. Topics over time: a non-markov continuous-time model of topical trends. In *Proceedings of the 12th international conference on Knowledge discovery and data mining*, pages 424–433. ACM, 2006. [27]

X. Wei and W. B. Croft. LDA-based document models for ad-hoc retrieval. In *Proceedings of the 29th International ACM SIGIR Conference on Research and Development in Information Retrieval*, pages 178–185, 2006. [209]

W. Wong, J. Horgan, S. London, and H. Agrawal. A study of effective regression testing in practice. In *Proceedings of the 8th International Symposium On Software Reliability Engineering*, pages 264–274, 1997. [63, 66]

C. Wu, E. Chang, and A. Aitken. An empirical approach for semantic web services discovery. In *Proceedings of the 19th Australian Conference on Software Engineering*, pages 412–421, 2008. [44, 47, 54, 56]

S. Yoo and M. Harman. Regression testing minimization, selection and prioritization: A survey. *Software Testing, Verification and Reliability*, 22(2):67–120, 2010. [64, 65, 83]

S. Yoo, M. Harman, P. Tonella, and A. Susi. Clustering test cases to achieve effective and scalable prioritisation incorporating expert knowledge. In *Proceedings of the 18th International Symposium on Software Testing and Analysis*, pages 201–212, 2009. [66]

H. Zawawy, K. Kontogiannis, and J. Mylopoulos. Log filtering and interpretation for root cause analysis. In *Proceedings of the 26th International Conference on Software Maintenance*, pages 1–5, 2010. [40, 54, 56]

C. X. Zhai. Statistical language models for information retrieval. *Synthesis Lectures on Human Language Technologies*, 1(1):1–141, 2008. [18, 22, 23, 25]

L. Zhang, J. Zhou, D. Hao, L. Zhang, and H. Mei. Prioritizing JUnit test cases in absence of coverage information. In *Proceedings of the 25th International Conference on Software Maintenance*, pages 19–28, 2009. [62, 66, 68, 83]

X. Zhou, X. Zhang, and X. Hu. Dragon toolkit: Incorporating auto-learned semantic knowledge into large-scale text retrieval and mining. In *Proceedings of the 19th International Conference on Tools with Artificial Intelligence*, volume 2, pages 197–201, 2007. [49]

T. Zimmermann, R. Premraj, and A. Zeller. Predicting defects for Eclipse. In *Proceedings of the 3rd International Workshop on Predictor Models in Software Engineering*, 2007. [127]

T. Zimmermann et al. Mining version histories to guide software changes. *IEEE Transactions on Software Engineering*, pages 429–445, 2005. [14]