STUDYING SOFTWARE QUALITY USING TOPIC MODELS

by

TSE-HSUN CHEN

A thesis submitted to the
School of Computing
in conformity with the requirements for
the degree of Master of Science

Queen's University
Kingston, Ontario, Canada
January 2013

Abstract

Software is an integral part of our everyday lives, and hence the quality of software is very important. However, improving and maintaining high software quality is a difficult task, and a significant amount of resources is spent on fixing software defects. Previous studies have studied software quality using various measurable aspects of software, such as code size and code change history. Nevertheless, these metrics do not consider all possible factors that are related to defects. For instance, while lines of code may be a good general measure for defects, a large file responsible for simple I/O tasks is likely to have fewer defects than a small file responsible for complicated compiler implementation details. In this thesis, we address this issue by considering the *conceptual concerns* (or *features*). We use a statistical topic modelling approach to approximate the conceptual concerns as *topics*. We then use topics to study software quality along two dimensions: *code quality* and *code testedness*. We perform our studies using three versions of four large real-world software systems: Mylyn, Eclipse, Firefox, and NetBeans.

Our proposed topic metrics help improve the defect explanatory power (i.e., fitness of the regression model) of traditional static and historical metrics by 4–314%. We compare one of our metrics, which measures the cohesion of files, with other

topic-based cohesion and coupling metrics in the literature and find that our metric gives the greatest improvement in explaining defects over traditional software quality metrics (i.e., lines of code) by 8–55%.

We then study how we can use topics to help improve the testing processes. By training on previous releases of the subject systems, we can predict not well-tested topics that are defect prone in future releases with a precision and recall of 0.77 and 0.75, respectively. We can map these topics back to files and help allocate code inspection and testing resources. We show that our approach outperforms traditional prediction-based resource allocation approaches in terms of saving testing and code inspection efforts.

The results of our studies show that topics can be used to study software quality and support traditional quality assurance approaches.

Co-authorship

Earlier versions of the work in the thesis were published as listed below:

1. *Explaining Software Defects Using Topic Models (Chapter 3)*

   Tse-Hsun Chen, Stephen W. Thomas, Meiyappan Nagappan, Ahmed E. Hassan, 9th Working Conference on Mining Software Repositories (MSR). Zurich, Switzerland. June 2-3, 2012 (acceptance rate: 18/64 (28%)).

   My contribution: Drafting the research plan, collecting the data, analyzing the data, writing and polishing the paper drafts, and presenting the paper.

2. *Explaining Software Defects and Studying Cohesion and Coupling Using Topic Models (Chapter 3)*

   Tse-Hsun Chen, Stephen W. Thomas, Meiyappan Nagappan, Ahmed E. Hassan, to be submitted for the Journal of Empirical Software Engineering. Springer Press (Impact Factor 1.854).

   My contribution: Drafting the research plan, collecting the data, analyzing the data, and writing and polishing the paper drafts.

3. *Studying the Effect of Testing on Code Quality using Topic Models (Chapter 4)*
   Tse-Hsun Chen, Stephen W. Thomas, Hadi Hemmati, Meiyappan Nagappan, Ahmed E. Hassan, under review for the Journal of Empirical Software Engineering. Springer Press (Impact Factor 1.854).

   My contribution: Drafting the research plan, collecting the data, analyzing the data, and writing and polishing the paper drafts.

Acknowledgments

First, I would like to thank my dearest supervisor Dr. Ahmed E. Hassan for his great supervision and support. Without his suggestions and advice, this thesis cannot be possible. Ahmed, I really enjoy working under your supervision, and I will remember all the skills that I learned from you for the rest of my life. I greatly appreciate your help during my MSc study.

I would also like to thank all the members at the Software Analysis and Intelligence Lab (SAIL), who help me throughout my thesis. I feel very honored to have the chance to work in this wonderful lab.

Thank you MSR for giving me the chance to explore the world, and to meet someone special in my life.

I would like to dedicate my work to my parents and family for their continuous support through out my life. Without them, this thesis would not have been possible.

List of Notations and Abbreviations

α: Dirichlet prior for document-topic distributions in LDA

β: Dirichlet prior for topic-word distributions in LDA

δ: Topic membership cut-off

θ: a document-topic topic membership value matrix

$\theta_i j$: the topic membership value of topic j in document i

ϕ: word distribution matrix for topics

p^2: Pearson correlation coefficient

z: a topic

D^2: deviance explained

D_{pre}: pre-release defect density

D_{post}: post-release defect density

f: a file

tf: term frequency

idf: inverse document frequency

II: number of iterations used for sampling LDA topics

K: number of topics in LDA

R^2: coefficient of determination

W ratio: proportion of a topic's weight found in test and source code files

W_{source}: a topic's source code weight

W_{test}: a topic's test code weight

AIC: Akaike information criterion

CCBO: Conceptual Coupling between Object classes

CLCOM5: Lack of Cohesion on Methods

CVS: Concurrent Versions System

DTM: Defect-prone topic membership

GoF: Goodness of fit

GUI: Graphical user interface

HTHD: High testedness and high defect density

HTLD: High testedness and low defect density

LDA: Latent Dirichlet Allocation

LOC: Lines of Code

LSI: Latent Semantic Indexing

LTHD: Low testedness and high defect density

LTLD: Low testedness and low defect density

LM: Linear regression model

MAS: Maximum Asymmetry Score

MEV: Maximum Edge Value

MWE: Maximal Weighted Entropy

MIC: Maximal information coefficient

MINE: Maximal information-based nonparametric exploration

NDT: Number of defect-prone topics

NT: Number of topics

PC: Principal Components

PCA: Principal Component Analysis

POST: Post-release defects

PRE: Pre-release defects

RTC: Relational topic-based coupling

RTM: Relational topic model

SVD: Singular Value Decomposition

SVN: Subversion

TM: Topic membership

UI: User interface

VIF: Variance Inflation Factor

XML: Extensible Markup Language

Table of Contents

List of Tables

List of Figures

Chapter 1

Introduction

Software quality is an important issue in software engineering because the cost of fixing software defects can be prohibitively expensive (Slaughter et al., 1998). As a result, researchers have tried to uncover the possible reasons for software defects using different classes of software metrics, such as product metrics (i.e., size), process metrics (i.e., time to fix a defect), and project metrics (i.e., team size) (Hall et al., 2011; Kan, 2002). Indeed, such approaches have shown some success in explaining the defect-proneness of certain software entities (e.g., methods, classes, files, or modules) (Hall et al., 2011). However, these classes of metrics do not take into account the actual *conceptual concerns* of the software system—the main software requirements and design decisions (i.e., high level software features) that may affect implementation details (Liu et al., 2009; Robillard and Murphy, 2007). For instance, while lines of code may be a good general measure for defects, it is not always so: the largest file in the Mylyn project (Foundation, 2012), for example, has 2,771 lines of code but no defects. A much smaller file, with 23 lines of code, does contain a defect in version 1.0.

In addition to using software metrics for identifying defects, testing is also widely used for detecting defects prior to the release of a software system. Software developers use test cases to uncover defects in software systems before a release to ensure software quality. Previous research has studied code coverage/test coverage (i.e., the proportion of the source code files that have been tested) using traditional testing criteria, such as statement coverage, path coverage, and function coverage (Huang, 1975; Myers et al., 2004; Nagappan et al., 2007; Ntafos, 1988). These metrics analyze code coverage from the code structure. However, these code coverage approaches do not consider the *conceptual concerns* in the source code files. Since testers test software at the level of concerns (Weyuker, 1998), by identifying which concerns require more testing, testers may gain insight about where to allocate testing resources and how to improve code quality.

Recent research in Software Engineering proposes a new class of approaches for predicting defects based on conceptual concerns (Chen et al., 2012; Linstead et al., 2008; Liu et al., 2009; Maskeri et al., 2008; Nguyen et al., 2011). These studies approximate concerns as topics using *statistical topic models*, such as latent Dirichlet allocation (LDA) (Blei et al., 2003). Topics are a set of related words in a given document, in our case source code or test files are examples of documents. These works provide initial evidence that topics in software systems are related to the defect-proneness of source code files, opening a new perspective for explaining why some files are more defect-prone than others.

In this thesis, we build on this line of previous research. We study the relationship between software quality and topics in source code files. In order to do this, we propose using topics to study software quality along two dimensions: *code quality*

and *code testedness*.

In the first dimension, we propose a set of topic-based metrics. We want to study if our new topic-based metrics can better explain software quality. Our metrics outperform state-of-the-art topic-based cohesion and coupling metrics in explaining the quality of code for several large open source software systems.

In the second dimension, we examine how we can use topics to support the software testing process. We measure topic testedness by comparing the prevalence of a topic in source code files to its prevalence in test files, and study the effect of testing on software quality, at the abstraction level of *topics*. By testing the defect prone topics that require more testing and that are discovered by our approach, additional defects can possibly be located. In addition, because topics can be linked back to source code files, managers and software developers can allocate more testing resources on the files related to these topics, and thus improve software quality and reduce maintenance costs. Our topic-based approach outperforms and complements traditional prediction-based testing resource allocation approaches, and helps uncover new sets of defect-prone files that may require more testing.

Although our results show that topics can be used to study software quality, the approaches proposed in this thesis are dependent on the quality of the generated topics. Hence it is important to use them in cases where the software systems have sufficient linguistic data such as identifier names and comments in them for the statistical topic models to generate good quality topics.

1.1 Research Statement

Prior research studies software quality from the perspective of structure, development process, and developer interaction of a software system. However, these software metrics do not consider the underlying conceptual concerns in source code files. In this thesis, we approximate concerns as topics using statistical topic models, and we study the relationship between topics and software quality.

Topics, which are approximations of software concerns, can be used to study software quality by better explaining the quality of code and helping allocate software quality assurance efforts effectively.

1.2 Thesis Overview

1. *Using Topic Models to Study Code Quality (Chapter 3)?*

   We measure topic defect-proneness by considering the defect history of topics, and study the defect-proneness of topics across different versions. We propose a number of topic-based metrics, and study whether our metrics can help explain software defects. We show that our metrics outperform and complement other topic-based cohesion and coupling metrics.

2. *Using Topic Models to Study Code Testedness (Chapter 4)?*

   We measure topic testedness (i.e., how well a topic is tested) and use topics to help improve the software testing process and ensure software quality. We show that when a topic is well tested, it is less likely to be defect prone. To help allocate testing resources, we show that we can obtain good precision and

recall when predicting less tested and defect prone topics for future releases of the software. We also show that our approach can find other sets of defect prone files that are not captured by traditional approaches.

1.3 Thesis Contributions

In this thesis, we empirically validate our results through case studies on many versions of multiple large software systems. Our contributions are as follows:

1. Proposing a different view of studying software quality – using topics.
2. Proposing topics metrics which help explain software defects and comparing them with current state-of-the-art metrics.
3. Proposing a topic-based technique to help improve the effectiveness of the testing process in finding defects.

1.4 Organization of the Thesis

The rest of the thesis is organized as follows: Chapter 2 briefly introduces some topic modelling, source code preprocessing approaches, and regression analysis, and discusses related works. Chapter 3 presents the first dimension of our study, which is using topics to study code quality. We use topic defect information to define *topic defect density*, and show that it is possible to study defects using topics. We propose our topic-based metrics and examine if our metrics help traditional static and historical metrics explain defects. We also compare our metric with other state-of-the-art topic-based cohesion and coupling metrics. In Chapter 4, we use topic

models to study software quality from the perspective of code testedness. We show how one can use our proposed approach to prioritize file inspection and testing efforts. Finally, Chapter 5 concludes the thesis with a discussion of the limitations and potential future research directions.

Chapter 2

Background and Related Work

2.1 Topic Models

Our goal is to determine which concerns are in each source code file. This information is often not easily available, since developers do not often manually categorize each file and a file may continue several concerns (Maskeri et al., 2008). In this thesis, we approximate concerns using statistical topics, following the work of previous research (Baldi et al., 2008; Maskeri et al., 2008; Thomas et al., 2010). In particular, we extract the *linguistic data* from each source code file, i.e., the identifier names and comments, which helps to determine the functionality of a file (Kuhn et al., 2007). We then treat the linguistic data as a corpus of textual documents, which we use as a basis for topic modeling. In this section, we provide a brief description of some topic modelling techniques that are used in this thesis.

	Top words
z_1	*os, cpu, memory, kernel*
z_2	*network, speed, bandwidth*
z_3	*button click mouse right*

(a) Topics (Z).

	z_1	z_2	z_3
f_1	0.3	0.7	0.0
f_2	0.0	0.9	0.1
f_3	0.5	0.0	0.5

(b) Topic memberships (θ).

Figure 2.1: Example topic model in which three topics are discovered from three source code files (not shown). (a) The three discovered topics (z_1, z_2, z_3) are defined by their top (i.e., highest probable) words. (b) The three source code files (f_1, f_2, f_3) can now be represented by a topic membership vector.

2.1.1 Latent Dirichlet Allocation

In latent Dirichlet allocation (LDA), a *topic* is a collection of frequently co-occurring words in the corpus. Given a corpus of n documents $f_1, ..., f_n$, topic modeling approaches automatically discover a set Z of topics, $Z = \{z_1, ..., z_K\}$, as well as the mapping θ between topics and documents (see Figure 2.1). The number of topics, K, is an input that controls the granularity of the topics. We use the notation θ_{ij} to describe the topic membership value of topic z_i in document f_j.

Intuitively, the top words of a topic are semantically related and represent some real-world concept. For example, in Figure 2.1a, the three topics represent the concepts of "operating systems," "computer networks," and "user input." The topic membership of a document then describes which concepts are present in that document: document f_1 is 30% about operating systems and 70% about computer networks.

More formally, each topic is defined by a probability distribution over all of the unique words in the corpus. Given two Dirichlet priors, α and β, a topic model will

generate a topic distribution θ_j for f_j based on α, and generate a word distribution ϕ_i for z_i based on β. Choosing the right parameter values for K, α, and β is more of an art than a science, and depends on the size of the corpus and the desired granularity of the topics (Wallach et al., 2009).

The topic membership values define links between topics and source code files. In Figure 2.1b, the source code file f_1 belongs to topic z_1 and z_2, because its membership values of these two topics are larger than 0. This is because f_1 contains words from topics z_1 and z_2, such as *os, cpu, and network*.

2.1.2 Latent Semantic Indexing

Latent Semantic Indexing (LSI) is computed by applying singular value decomposition (SVD) on the term-document matrix. Each row of the term-document matrix corresponds to a document (i.e., software entity), and the columns represent the unique words in all the documents. For example, the index $[i, j]$ in the matrix represents the number of occurrences of the word j in the document i. In addition to word frequency count, other weight measures such as $tf * idf$ (term frequency-inverse document frequency) are also often used. $Tf * idf$ measures the weight (i.e., importance) of a word in a document given the whole corpus, where tf is the term frequency in a document, and idf is the inverse document frequency. In this thesis, we use $tf * idf$ instead of the raw word frequency count. SVD reduces the dimensions in the term-document matrix by selecting the top K dimensions (i.e., K topics) with the largest singular values. This reduction helps remove synonymy and polysemy in the term-document matrix (Witten et al., 2011). Finally, by computing the cosine distance between each row (document), we obtain a similarity score of

the documents.

2.1.3 Relational Topic Models

Chang et al. propose a variant of LDA, called Relational Topic Models (RTM), which is able to predict whether two documents are related using their underlying topics (Chang and Blei, 2009). RTM is able to give a linking probability between two documents (how likely these two documents are related). RTM has recently been used by Gethers et al. (Gethers and Poshyvanyk, 2010) to measure the coupling between source code files.

2.2 Source Code Preprocessing

Source code files contain many human-readable information in identifier names and comments. However, we cannot use the raw source code files as our text corpus for running topic models, because programming language syntax and control structures do not contain much information about the concerns in a file. Therefore, we first collect the source code files from each version of each subject system, and then preprocess the files using the preprocessing steps proposed by Kuhn et al. (Kuhn et al., 2007). Namely, we first extract comments and identifier names from each file. Next, we split the identifier names according to common naming conventions, such as camel case and underscores. Finally, we stem the words and remove common English-language stop words.

2.3 Defect Modeling

2.3.1 Types of Regression Models

Researchers in Empirical Software Engineering often use regression analysis to predict or understand the factors that lead to software defects. Various types of regression models are commonly used, and in particular, linear and logistic regressions are the most popular ones (Golberg and Cho, 2004; Kutner et al., 2004). Given some software quality metrics (i.e., dependent variables), linear regression models the number of defects (i.e., independent variable) in files. Logistic regression, on the other hand, models the probability of a file being defective or not. Lines of code, number of pre-release defects, and code changes before release (code churn) are commonly used to predict software defects (DAmbros et al., 2010; Gyimothy et al., 2005; Oram and Wilson, 2010). Research on several open and commercial systems shows that these three baseline metrics outperform most other more complex metrics in the literature today. Studies which propose new metrics must compare the performance of their new metrics against these baseline metrics (Oram and Wilson, 2010).

2.3.2 Typical Steps in Regression Analysis

Removing Collinear Variables

Regression analysis is a type of statistical analysis, where the regression model is trying to fit the value of the independent variables according some distributions (e.g., normal distribution or Bernoulli distribution), given the dependent variables.

Therefore, if the dependent variables are collineared together (also called multicollinearity problem), then the results of the regression analysis (i.e., statistical significance and confidence interval) may be inaccurate.

The Variance Inflation Factor (VIF) is often used to detect and remove collinear variables. Variables that have a VIF score larger than 10 are removed, since these variables might have multicollinearity problems (Kutner et al., 2004). Another commonly used approach is Principal Component Analysis (PCA). PCA transforms the variables into a set of uncorrelated variables, called principal components (PCs) (Jolliffe, 2002). PCs can replace independent variables in regression models, and such approach is called principal component regression analysis (Jolliffe, 2002).

Finding the Most Effective Variables and Examining Goodness of Fit

In some situations where we need to find a subset of the dependent variables that are most effective for predicting (or understanding) the independent variable, researchers use stepwise regression to do variable selection (Golberg and Cho, 2004; Kutner et al., 2004). Stepwise regression in each step will add/remove a variable to/from the regression model, while trying to optimize the goodness of fit.

Goodness of fit (GoF) measures how well a set of independent variables together describe the dependent variable. For example, adjusted R^2 and D^2 are used to measure GoF of linear and logistic regression models, respectively. The values of adjusted R^2 and D^2 range from 0 to 1, where 0 indicates there is no relationship between the dependent and the independent variables, and 1 indicates there is a strong correlation between the dependent and the independent variables. The problem with adjusted R^2 and D^2 is that their values increase when the number

of independent variables increases, which will give a more biased GoF measure. Another GoF measuring criterion called Akaike information criterion (AIC) assigns a penalty to regression models with more independent variables.

Base Model and the Explanatory Power of a Metric

To examine whether a new metric is adding any new information to a regression model, we can add the metric to a regression model, and study the increase in GoF. In this thesis, we call this increase in GoF the explanatory power of the new metric, since it represents how this new metric improves the overall predictive power of the regression model. The model that the new metric is added to is called the base model.

2.4 Related Work

Approximating Concerns using Topic Models

Recently, many researchers used topic modeling approaches to understand software systems from a different point of view than from the traditional structural and historical views. For example, Kuhn et al. used Latent Semantic Indexing (LSI) to cluster the files in a software system according to the similarity of word usage (Kuhn et al., 2007). Maskeri et al. were the first to apply LDA to source code to uncover its conceptual concerns (Maskeri et al., 2008). Linstead et al. and Thomas et al. used topics to study the evolution of concerns in the source code (Linstead et al., 2008; Thomas et al., 2010, 2011).

Predicting Defects using Topic Models

A few recent studies have tried to establish a link between topics and defects. For example, Liu et al. propose a new metric, called Maximal Weighted Entropy (MWE) (Liu et al., 2009), to measure the level of cohesion in a software system. MWE, for each topic, captures the topic occupancy and distribution of each file, i.e., how many different topics a file contains. While this metric focuses on the cohesiveness of topics in a file, our proposed metrics focus on the defect-prone topics in a file.

Nguyen et al. use LDA to predict defects (Nguyen et al., 2011). The authors first apply LDA to the subject systems using $K=5$ topics, and for each source code file they multiply the topic memberships by the file's LOC. As a result, the authors obtain five topic variables for each file, and use these variables to build a prediction model. In this way, the authors provide initial evidence, that it is possible to explain defects using topic-based metrics. In this thesis, we are interested in explaining defects while also controlling for the standard defect explainers, i.e., LOC, churn, and pre-release defects. In addition, we consider a larger number of topics in order to capture more accurate and detailed conceptual concerns. We use Principal component analysis (PCA) (Jolliffe, 2002) to extract the most effective topics and avoid the possible problem of multicollinearity and minimize the effects of overfitting.

Capturing Software Cohesion and Coupling using Topic Models

Marcus et al. capture cohesion using LSI (Marcus and Poshyvanyk, 2005; Marcus et al., 2008), and show that it is possible to predict defects using the level of cohesion in a file (Marcus et al., 2008). Poshyvanyk et al. also use LSI to capture level

of coupling by the cosine similarity score among files (Poshyvanyk and Marcus, 2006). Ujhazi et al. propose two new cohesion and coupling metrics based on previous works by Marcus et al. and Poshyvanyk et al. (Marcus and Poshyvanyk, 2005; Marcus et al., 2008; Poshyvanyk and Marcus, 2006), and provide a parametric version (i.e., calculate the metrics based on a given parameter) of the metrics (Ujhazi et al., 2010). In addition, Gethers et al. use relational topic models (Chang and Blei, 2009) to uncover coupling among files (Gethers and Poshyvanyk, 2010), and use the level of coupling for prioritizing file inspection work.

Other Uses of Topic Models in Software Engineering

Other uses of topic models in software engineering tasks include concept location (Cleary et al., 2008; Lukins et al., 2010; Poshyvanyk et al., 2007; Rao and Kak, 2011; Revelle et al., 2011), traceability link recovering (Asuncion et al., 2010), and building source code search engines (Tian et al., 2009).

2.5 Chapter Summary

In this chapter, we briefly introduce topic models, and different topic modelling approaches. We also define how we preprocess the source code files in order to apply topic models on them. We introduce basic background about model fitting and regression analysis. We finally survey related works focused on four different branches: capturing concerns using topic models, predicting defect using topic models, measuring software cohesion and coupling using topic models, and other

uses of topic models in software engineering. In the next chapter, we perform a preliminary study and examine whether we can use topics to study software quality.

Chapter 3

Using Topic Models to Study Code Quality

Researchers have proposed various metrics based on measurable aspects of the source code entities (e.g., methods, classes, files, or modules) of a software project in an effort to explain the relationships between software development and software defects. However, these metrics largely ignore the actual functionality, i.e., the *conceptual concerns*, of a software system, which are the main technical concepts that reflect the business logic or domain of the system. For instance, while lines of code may be a good general measure for defects, a large file responsible for simple I/O tasks is likely to have fewer defects than a small file responsible for complicated compiler implementation details. In this chapter, we study the effect of conceptual concerns on code quality.

We use LDA to approximate software concerns as *topics*; we then propose various metrics based on these topics to help explain the defect-proneness (i.e., quality) of the files. Paramount to our proposed metrics is that they take into account the

defect history of each topic.

In particular, we aim to answer the following four research questions using case studies on multiple versions of Mozilla Firefox, Eclipse, Mylyn, and NetBeans:

RQ1: Are some topics more defect-prone than others?

We find that some topics, such as those related to new features and the core functionality of a system, have a much higher defect density than others (average skewness 7.25, where a skewness of 1 is already considered highly skewed).

RQ2: Do defect-prone topics remain defect-prone over time?

We find that defect-prone topics remain so over time, indicating that prior defect-proneness of a topic can be used to explain the future behavior of topics and their associated files (average Spearman correlation is 0.54).

RQ3: Can our proposed topic-based metrics help explain why some files are more defect-prone than others?

We find that including our proposed topic-based metrics provides additional explanatory power (4 – 314% improvement) about the defect-proneness of files over existing product and process metrics.

RQ4: How do our metrics compare with other topic-based cohesion and coupling metrics?

We find that our metrics outperform other topic-based cohesion and coupling metrics. Our metric can give improvement in defect explanatory power over baseline metric (i.e., lines of code) by 8–55%, where other metrics give 8–50% of improvement. Practitioners may benefit from including our metric

when analyzing code quality using cohesion and coupling.

Chapter Overview

Section 3.1 talks about the subject systems that we use to answer the research questions, and Section 3.2 describes the design of our case studies. Section 3.3 shows the results of our case studies, and Section 3.4 shows the result of our the parameter sensitivity analysis. We compare our metric with other topic-based cohesion and coupling metrics in Section 3.3.4. Section 3.5 discusses potential threats to validity, and Section 3.7 concludes the chapter with a brief summary of our findings.

3.1 Choice of Subject Systems

We focus on four large, real-world subject systems: Mylyn, Eclipse, Firefox, and NetBeans (Table 3.1). For each system, we look at three different versions (versions 1.0, 2.0, and 3.0 of Mylyn, versions 2.0, 2.1, and 3.0 of Eclipse, versions 1.0, 1.5, and 2.0 of Firefox, and versions 4.0, 5.0, and 5.5.1 of NetBeans). Eclipse is a popular IDE (integrated development environment), which has an extensive plugin architecture. Mylyn is a popular plugin for Eclipse that implements a task management system. Firefox is a well-known open source web browser that is used by millions of users. Finally, NetBeans is a popular module-based IDE which is implemented in Java.

Table 3.1: Statistics of the subject systems.

	Total lines of code (K)	No. of files	Pre-release defects	Post-release defects	Programming language
Mylyn 1.0	127	833	1,047	712	Java
Mylyn 2.0	136	923	2,015	1,012	Java
Mylyn 3.0	165	1,115	2,045	480	Java
Firefox 1.0	2,841	5,523	638	454	C/C++
Firefox 1.5	3,111	5,879	716	946	C/C++
Firefox 2.0	3,205	5,942	1,134	453	C/C++
Eclipse 2.0	797	6,716	7,634	1,691	Java
Eclipse 2.1	987	7,799	4,975	1,182	Java
Eclipse 3.0	1,305	10,496	7,421	2,679	Java
NetBeans 4.0	915	4,253	630	311	Java
NetBeans 5.0	1,957	8,849	1,339	217	Java
NetBeans 5.5.1	3,302	16,383	883	795	Java

3.2 Case Study Design

In this section, we describe our analysis process, depicted in Figure 3.1. We use MALLET (McCallum, 2002) as our LDA implementation, which uses Gibbs sampling to approximate the joint distribution of topics and words. We run MALLET with 10,000 sampling iterations (II), and 1,000 of the iterations are used to optimize α and β using hyperparameter optimization. In addition, we build the topics using both unigrams (single words) and bigrams (pairs of adjacent words), since bigrams help to improve the performance for word assignments in topic modeling the best (Brown et al., 1992).

We apply LDA to all versions of the preprocessed files of a system at the same time, an approach proposed by Linstead et al. (Linstead et al., 2008). For this study, we use K=500 topics for all subject systems. Lukins et al. found that 500 topics is a

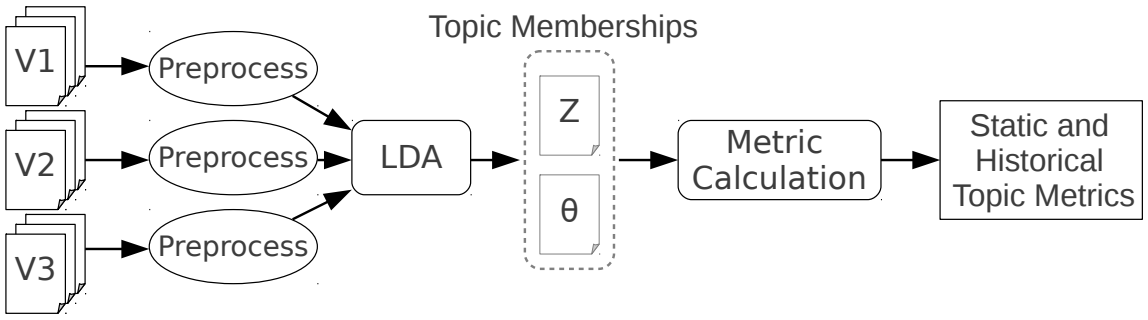


Figure 3.1: Process of calculating topic-based metrics. After preprocessing the source code, we run LDA on all versions of the source code files together. Using the topics and topic memberships that LDA returns, we calculate the topic-based metrics.

good number for Eclipse and Mozilla (Lukins et al., 2010), and we also feel this is a reasonable choice for Mylyn and NetBeans (more discussions about the parameter choices in Section 3.4).

3.2.1 Proposed Topic-based Metrics

To help explain the defect-proneness of source code files, we propose two categories of topic-based metrics: *static* and *historical*. Static topic metrics use only a single snapshot of the software system, while historical metrics use the defect history of topics. In the formulation of our topic-based metrics, we also consider traditional software metrics:

- ***LOC(*** f_j***)*** The lines of code of file f_j.
- ***PRE(*** f_j***)*** The number of pre-release defects of file f_j, which are those defects related to f_j up to six months before a given version.
- ***POST(*** f_j***)*** The number of post-release defects of file f_j, which are those defects found up to six months after a given version.

Using these software metrics and the results from topic modeling (as explained in Section 2.1.1), we propose the following topic-based metrics.

Topic Defect Density

The defect density of a source code file is a well-known software metric, defined as the ratio of the number of defects in the file to its size. Using this ratio as motivation, we define the ***pre-release defect density*** (***D_{PRE}***) of a topic z_i as

$$D_{\text{PRE}}(z_i) = \sum_{j=1}^{n} \theta_{ij} * \left(\frac{\text{PRE}(f_j)}{\text{LOC}(f_j)} \right), \tag{3.1}$$

where n is the total number of source code files and θ_{ij} is the topic membership of topic z_i in source code file f_j. Similarly, we define ***post-release defect density*** (***D_{POST}***) of a topic z_i as

$$D_{\text{POST}}(z_i) = \sum_{j=1}^{n} \theta_{ij} * \left(\frac{\text{POST}(f_j)}{\text{LOC}(f_j)} \right). \tag{3.2}$$

Since the topic membership value represents the probability that a source code file belongs to a certain topic, the topic defect density represents the possible number of defects in the topic per line of code across all files that contain the topic.

Static Topic-based Metrics

We propose static topic-based metrics to capture the number of topics a file contains, and the topic membership of each file. We define the ***Number of Topics*** (***NT***) of a

file f_j as

$$\text{NT}(f_j) = \sum_{i=1}^{K} I(\theta_{ij} \geq \delta) \tag{3.3}$$

where I is the indicator function that returns 1 if its argument is true, and 0 otherwise. δ is a cut-off threshold that determines if a topic plays an important role in a given file. The NT metric measures the level of cohesion in a file: files with a large number of topics may be poorly designed or implemented, and thus may have higher chances to have defects (Liu et al., 2009).

We define the ***Topic Membership*** (***TM***) of a file f_j as the topic membership values returned by the topic modeling approach:

$$\text{TM}(f_j) = \theta_j. \tag{3.4}$$

The intuition behind this metric is that we assume different topics have different effects on the defect-proneness of a file. Some topics (e.g., a compiler-related topic) may increase the defect-proneness of a file, but other topics (e.g., an I/O-related topic) may actually decrease the defect-proneness. By using all the topic membership values, the TM metric captures the full behavior of a file.

Historical Topic-based Metrics

We extend the static topic-based metric by considering the defect history of each topic. In order to calculate the number of defect-prone topics in a file, we define a *defect-prone topic* as a topic that has more defects than the average of all topics.

The set of defect-prone topics, B, is defined by

$$B = \{z_i \in Z \text{ s.t. } D_{\text{PRE}}(z_i) > \mu(D_{\text{PRE}}(Z))\}, \tag{3.5}$$

where $\mu(D_{\text{PRE}}(Z))$ is the mean of the topic defect densities of all topics.

We define the ***Number of Defect-prone Topics*** (***NDT)*** in file f_j by

$$\text{NDT}(f_j) = \sum_{i=1}^{K} I((z_i \in B) \wedge ((\theta_{ij}) \geq \delta)). \tag{3.6}$$

We define the ***Defect-prone Topic Membership*** (***DTM***) metric of file f_j as the topic memberships of defect-prone topics:

$$\text{DTM}(f_j) = \theta_{ij} \text{ where } z_i \in B. \tag{3.7}$$

DTM is the same as TM, except it only contains the topic memberships of defect-prone topics.

We set the membership threshold δ in Equations 3.3 and 3.6 to 1%. This value prevents topics with small, insignificant memberships in a file from being counted in that file's metrics.

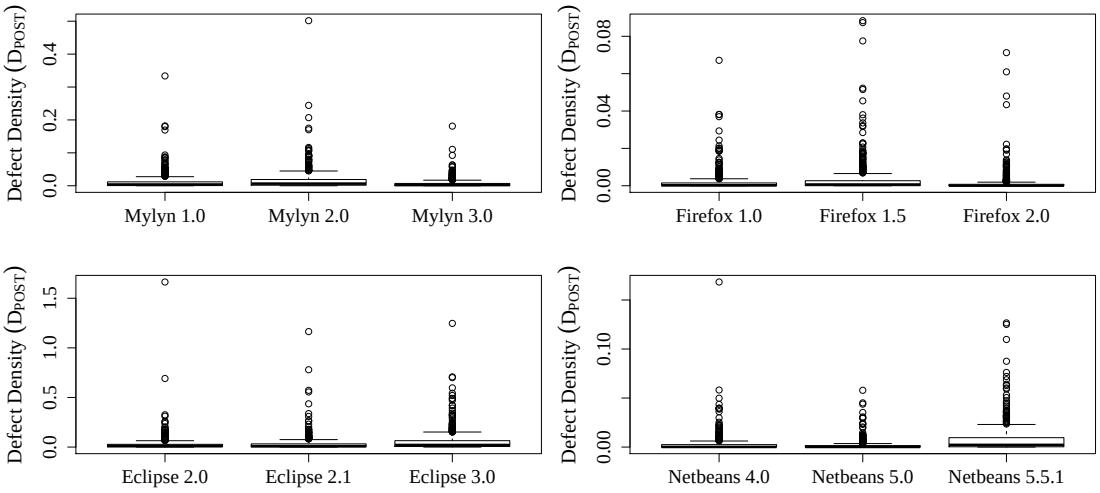


Figure 3.2: Box plots of the topic defect density of three versions of Mylyn, Firefox, Eclipse, and NetBeans. The y-axis represent the topic defect density.

3.3 Results of Case Studies

3.3.1 Are some topics more defect-prone than other topics?

Approach

We use Equation 3.2 to calculate the topic defect density (D_{POST}) for each topic in the software system. We visualize the distribution of defect densities using box plots, and we provide a table of the five number summary and skewness of the densities. We then perform Kolmogorov-Smirnov non-uniformity tests to statistically determine if there is a significant difference between the defect densities of the various topics.

Table 3.2: Five-number summary and skewness of defect densities of all subject systems. The defect densities are highly skewed, and most of the topics have a density value close to zero.

	Min.	1st Qu.	Median	3rd Qu.	Max.	Skewness
Mylyn 1.0	0.00	0.00	0.00	0.01	0.33	7.18
Mylyn 2.0	0.00	0.00	0.01	0.02	0.50	7.41
Mylyn 3.0	0.00	0.00	0.00	0.01	0.18	6.00
Eclipse 2.0	0.00	0.00	0.01	0.03	1.66	13.28
Eclipse 2.1	0.00	0.00	0.01	0.03	1.16	7.92
Eclipse 3.0	0.00	0.01	0.02	0.06	1.25	4.90
Firefox 1.0	0.00	0.00	0.00	0.00	0.07	6.44
Firefox 1.5	0.00	0.00	0.00	0.00	0.09	5.88
Firefox 2.0	0.00	0.00	0.00	0.00	0.07	7.96
NetBeans 4.0	0.00	0.00	0.00	0.00	0.17	10.29
NetBeans 5.0	0.00	0.00	0.00	0.00	0.06	5.62
NetBeans 5.5.1	0.00	0.00	0.00	0.01	0.13	4.08

Results

The box plots of the density values of each software system are shown in Figure 3.2. Box plots show outliers and the five-number summary of the data (minimum, first quartile, median, third quartile, maximum). The actual values of the five-number summary and skewness of the defect densities is shown in Table 3.2. The outliers in Figure 3.2 are the defect-prone topics, which indicate that some topics have much higher defect densities than others. Table 3.2 further indicates that most topics have a low (almost zero) defect density value, and the values are significantly positively skewed.

The number of topics and defect-prone topics for each system is consistent across versions (Table 3.3). We find that Mylyn has more defect-prone topics than the

Table 3.3: For each system, we show the mean defect density value across all topics ($\mu(D_{\text{POST}})$), the number and percentage of defect-prone topics (NDT), the median number of topics in each file (Med. NT), and the median number of defect-prone topics in each file (Med NDT).

	K	$\mu(D_{\text{POST}})$	NDT	Med. (NT)	Med. (NDT)
Mylyn 1.0	500	0.01	139 (27.8%)	9	7
Mylyn 2.0	500	0.02	137 (27.4%)	9	7
Mylyn 3.0	500	0.01	128 (25.6%)	9	7
Eclipse 2.0	500	0.03	122 (24.4%)	9	5
Eclipse 2.1	500	0.03	124 (24.8%)	9	5
Eclipse 3.0	500	0.06	136 (27.2%)	10	6
Firefox 1.0	500	0.00	106 (21.2%)	5	3
Firefox 1.5	500	0.00	111 (22.2%)	5	3
Firefox 2.0	500	0.00	82 (16.4%)	5	2
NetBeans 4.0	500	0.00	106 (21.2%)	9	5
NetBeans 5.0	500	0.00	103 (20.6%)	9	5
NetBeans 5.5.1	500	0.01	147 (29.4%)	9	6

other three systems, while Firefox has the least number of defect-prone topics, and Eclipse has the highest mean defect density among four systems. NetBeans, on the other hand, has a similar median NT and NDT to that of Eclipse.

Finally, we apply the Kolmogorov-Smirnov test on the topic defect density values of each version of each subject system to verify the non-uniformity illustrated by our visualizations. If the p-value computed using Kolmogorov-Smirnov test is high, then the data is more likely to be uniformly distributed. However, we find that the p-values for all systems are significantly small (< 0.001), indicating that the distribution of defect density values is indeed not uniform (Stapleton, 2008).

Discussion

To better understand why some topics are more defect-prone than others, we investigate the relevant words of the top three most and least defect-prone topics (Table 3.4–3.7).

Mylyn. Mylyn, previously known as Mylar, is an Eclipse plugin for task management. We find that the topics with the highest defect densities are (i) those dealing with the Eclipse integration (topic 421), likely because the Eclipse plugin API changes so often; (ii) those that are related to the core functionality of the system, i.e., tasks and the task UI (topics 164 and 168); and (iii) those dealing with the test suite of Mylyn (topic 400), likely because of adding test cases for new defect fixes.

On the other hand, the least defect-prone topics deal with images and color (topics 405 and 178) and data compression (topic 175). We postulate that the logic behind these functions may be simpler and better defined than that of the core functionality topics.

Eclipse. Regarding Eclipse, two of the most defect-prone topics (topics 496 and 492) in Eclipse 2.0 are about CVS plug-ins. The build notes for this release indicate that the plug-ins supporting CVS-related functionalities were first introduced in this version, making it an active area of development. (In fact, according to Eclipse's defect repository, 17 defects relating to CVS remained unfixed after the 2.0 release.) A similar story holds for Eclipse 2.1, when integration for the Apache Ant build system was actively developed, leading to many defects in topic 131.

Another set of defect-prone topics in Eclipse deals with low-level details such as

memory operations and message passing (topics 462, 169, and 233). We hypothesize that the logic needed to implement these topics are more complex, leading to more defects.

The least defect-prone topics in Eclipse include those about bit-wise operations (topic 116), arrays (topic 182), and parameter parsing (topic 192). One reason that these might contain fewer defects is that errors in these topics may be observed during run time (e.g., ”array out-of-bounds”) and are thus more easily detected by developers during the testing phase of the project.

Firefox. One of the most defect-prone topics in Firefox 1.0 and 1.5 deals with event handing (topic 101), which is responsible for dispatching events according to network protocol responses. The topic is likely more defect-prone because the network stack has been modified several times to enable the dynamic re-rendering of complex webpages as they are being loaded.

Another defect-prone topic in Firefox 2.0 deals with accessing saved states (topic 80). The release notes for this version indicate that new features were introduced that allow the browser to restore previous sessions, and that the tabbed browsing functionality is updated.

Scanner Access Now Easy (SANE), an API that enables a scanner/digital camera application to be created with JavaScript, is one of the least defect-prone topics in all versions of Firefox (topic 280). Another topic that is not defect-prone deals with Base64 encoding (topic 359—the characters are segments of encoded characters), a known character standard.

NetBeans. NetBeans 4.0 released several new features, such as: project system based on Apache Ant (topic 219), code refactoring functionality, which uses a tree-like structure to manipulate changes (topic 182), and GUI for controlling debug and build operations (topic 372). These topics appear to be the most defect prone topics. Topics related to sending queries (topic 484), code completion (topic 57), parsing and manipulating Java code (topic 389), and using NetBeans to develop Mobile Information Device Profile (MIDP) application (topic 97) are also more defect prone. From the release notes of NetBeans, we find a possible reason these topic are more defect prone: these features are relatively new in early versions of NetBeans and so they have more defects reported after release of the software.

The least defect prone topics in NetBeans are about handling XML files (topic 236), handing database metadata (dmd) and metadata adaptor (topic 455), and providing support for Java Platform (J2EE) customizer (topic 211). We hypothesize these topics are related to simpler tasks, where defects may be observed during development. In addition, handling and storing software properties (topic 14), parser generator that produces syntax trees (topic 73), and helper classes for creating NetBeans GUI tests (topic 39) are less defect prone.

We find that different topics have different defect density, and most topics in a system are not defect prone (average skewness 7.25, and a skewness of 1 is already considered highly skewed). We find that topics that are related to new features and the core functionality of a system tend to have a much higher defect density than others.

Table 3.4: Top words and defect densities of the most/least defect-prone topics in our subject systems. The number on the left-hand side of each column represents the topic ID.

Most Defect Prone			**Least Defect Prone**		
Topic ID	Top words	Density	Topic ID	Top words	Density
Mylyn 1.0					
421	mylar, eclips, eclips_mylar, mylar_intern, mylar_task	0.334	405	src, dest, base, imag, fragment, imag_pattern	<0.001
164	task, list, task_list, task_ui, ui, plugin	0.182	178	lower_color, part, put_light green_lower, jface, medium	<0.001
400	test, suit, test_suit, add_test, add, suit_add	0.180	175	monitor, gzip, configur, key, bugzilla_attribut, iter	<0.001
Mylyn 2.0					
143	task, eclips, eclips_mylyn, mylyn, ui, task_ui	0.502	405	src, dest, base, imag, fragment, imag_pattern	<0.001
457	eclips, mylyn, eclips_mylyn, intern, mylyn_intern, core	0.244	178	lower_color, part, put_light, green_lower, jface, medium	<0.001
164	task, list, task_list task_ui, ui, plugin	0.207	175	monitor, gzip, configur, key, bugzilla_attribut, iter	<0.001
Mylyn 3.0					
143	task, eclips, eclips_mylyn, mylyn, ui, task_ui	0.181	178	lower_color, part, put_light, green_lower, jface, medium	<0.001
457	eclips, mylyn, eclips_mylyn, intern, mylyn_intern, core	0.111	310	aa, comparison_check, comparison, check, check_aa, aa_comparison	<0.001
168	repositori, task_repositori, task_core, repositori	0.092	6	select, caller, calle, editor, part, foo	<0.001

Continued in Table 3.5.

Table 3.5: Continued from Table 3.4. Top words and defect densities of the most/least defect-prone topics in our subject systems. The number on the left-hand side of each column represents the topic ID.

Most Defect Prone			**Least Defect Prone**		
Topic ID	Top words	Density	Topic ID	Top words	Density
Eclipse 2.0					
174	express, method, declar, ast, node, astnod	1.663	116	0xff, 0xff_0xff, src, dst, 0xa, 0xf	<0.001
496	option, local, seccion, folder, ccv_core, local option	0.691	146	printer, data, printer_data, code, error, dispos	<0.001
492	team, eclips_team, eclips, ccv, intern_ccv, team_intern	0.325	316	run, line, offset, style, length, item	<0.001
Eclipse 2.1					
143	form, toolkit, dfm, nfm, ui, eclips_ui	1.164	192	arg, vtbl, arg_arg, guid, iidfrom, system	<0.001
131	ant, eclips, task, eclips_ant, ui, intern	0.779	325	token, scribe, align, print, scribe_print, space	<0.001
233	bundl, recourc, resourc_bundl, kei, bundl_resourc, messag	0.572	116	0xff, 0xff_0xff, src, dst, 0xa, 0xf	<0.001
Eclipse 3.0					
462	memori, block,render, memori_block, view, address	1.247	330	packet, print, id, command, stream, spy	<0.001
169	transfer,data,code, transfer_data, java, object	0.708	182	array, constant, array_dim, dim, pixbuf, paramet	<0.001
131	ant, eclips, task, eclips_ant, ui, intern	0.700	270	pt, ph, pt_arg, arg, pg, wm	<0.001

Continued in Table 3.6.

Table 3.6: Continued from Table 3.5. Top words and defect densities of the most/least defect-prone topics in our subject systems. The number on the left-hand side of each column represents the topic ID.

Most Defect Prone			Least Defect Prone		
Topic ID	Top words	Density	Topic ID	Top words	Density
Firefox 1.0					
462	list, val, isvgvalu, modifi, observ, imethodimp	0.067	359	ghhd, sbz_yxkgd, yxkgd, sbz, yxkgd_ghhd, vghle_sbz	<0.001
381	frame, svgframe, comptr, queri, kid, add	0.038	280	sane, plugin, zoom, sane_plugin, instanc, error	<0.001
101	rv, rv_rv, comptr, nsresult, fail, fail_rv	0.038	361	child, border, spec, num, color, col	<0.001
Firefox 1.5					
305	elem, rv, length, map, rv_rv, comptr	0.088	359	ghhd, sbz_yxkgd, yxkgd, sbz, yxkgd_ghhd, vghle_sbz	<0.001
413	xform, elem, model, wrapper, instanc, xform_xpath	0.087	280	sane, plugin, zoom, sane_plugin, instanc, error	<0.001
101	rv, rv_rv, comptr, nsresult, fail, fail_rv	0.078	335	ck, rv, pr, log, modlog, log_modlog	<0.001
Firefox 2.0					
168	param, info, pruint, xpttype, val, count	0.071	359	ghhd, sbz_yxkgd, yxkgd, sbz, yxkgd_ghhd, vghle_sbz	<0.001
305	elem, rv, length, map, rv_rv, comptr	0.061	100	frame, pfd, span, psd, width, line	<0.001
80	access, state, retval, shell, node, comptr	0.048	280	sane, plugin, zoom, sane_plugin, instanc, error	<0.001

Continued in Table 3.7.

Table 3.7: Continued from Table 3.6. Top words and defect densities of the most/least defect-prone topics in our subject systems. The number on the left-hand side of each column represents the topic ID.

Most Defect Prone			**Least Defect Prone**		
Topic ID	Top words	Density	Topic ID	Top words	Density
NetBeans 4.0					
182	model, node, type, tree, tree_model, unknown	0.168	236	node, layout, properti, form, code, buf	<0.001
372	grid, bag, grid_bag, constraint, bag_constraint, awt	0.058	211	level, sourc_level, platform_kei, modul, chang, version	<0.001
219	project, helper, ant, properti, project_helper, evalu	0.050	455	prop, set, adaptor, dmd, sqlexcept, max	<0.001
NetBeans 5.0					
57	path, cp, classpath, recourc, implement, java	0.058	236	node, layout, properti, form, code, buf	<0.001
484	queri, javadoc, binari, root, file, binari_queri	0.045	211	level, sourc_level, platform_kei, modul, chang, version	<0.001
375	artifact, path, librari, project, ant, ant_artifact	0.044	455	prop, set, adaptor, dmd, sqlexcept, max	<0.001
NetBeans 5.5.1					
97	midp, compon, present, vmd, modul_vmd, modul	0.127	14	properti, pw, tf, properti_sheet, sheet, text	<0.001
389	token, token_token, offset, sequenc, lexer, token_id	0.125	73	jj, kind, cur, state, activ, po	<0.001
142	test, suit, junit, test_test, test_suit, netbean	0.110	39	jelli, mbean, constant, jelli_constant, nfwo, helper	<0.001

3.3.2 RQ2: Do defect-prone topics remain defect-prone over time?

Approach

In RQ1, we found that some topics are more defect-prone than other topics. In order to verify that these topics are consistently defect-prone over time, we compute the Spearman rank correlation of the topic defect density values among different versions. Spearman rank correlation computes the correlation on the ranks of the topic defect density, so a high correlation value will imply that the ranks of the topic defect proneness are consistent across versions (i.e., the most defect topic is still most defect prone in the next release).

Results

Table 3.8 shows the correlation of topic defect density among different versions of a system. The correlation values are consistently medium to high between different versions, which indicates that a defect-prone topic is still likely to be defect-prone in the later versions. We also see evidence of this in Table 3.4, as several of the top defect-prone topics are listed for each of the versions of a software system.

Therefore, it would be better to allocate more testing resources to previously-identified defect-prone topics, as they are likely to remain defect-prone in later releases.

Table 3.8: Spearman correlation coefficients of each topic's defect density across software versions.

	Mylyn 1.0	Mylyn 2.0	Mylyn 3.0
Mylyn 1.0	1.000	–	–
Mylyn 2.0	0.673	1.000	–
Mylyn 3.0	0.483	0.493	1.000

	Eclipse 2.0	Eclipse 2.1	Eclipse 3.0
Eclipse 2.0	1.000	–	–
Eclipse 2.1	0.529	1.000	–
Eclipse 3.0	0.438	0.530	1.000

	Firefox 1.0	Firefox 1.5	Firefox 2.0
Firefox 1.0	1.000	–	–
Firefox 1.5	0.536	1.000	–
Firefox 2.0	0.473	0.564	1.000

	NetBeans 4.0	NetBeans 5.0	NetBeans 5.5.1
NetBeans 4.0	1.000	–	–
NetBeans 5.0	0.612	1.000	–
NetBeans 5.5.1	0.588	0.563	1.000

The correlation of the topic defect density among different versions of a system is consistently from mid to high, which implies that defect prone topics are very likely to be defect prone in later versions. This information may help practitioners allocate testing resources more effectively.

3.3.3 RQ3: Can our proposed topic-based metrics help explain why some files are more defect-prone than others?

In the previous research questions, we have shown that topics have different levels of defect proneness, and defect prone topics tend to remain so over time. To provide evidence for practitioners that topics can help quality assurance processes, we examine the amount of additional deviance in post-release defects that our topic-based metrics can explain, with respect to traditional baseline metrics (LOC, PRE, and code churn) in this research question. This analysis allows us to verify empirically our theory that topic-based metrics provide additional explanatory power over post-release software defects.

Approach

Explaining Software Defects. As previously mentioned, software metrics can be classified as *static* or *historical*. Static metrics, such as lines of code (LOC), are obtained from a single snapshot of the system (Crawford et al., 1985). On the other hand, historical metrics require past information about the system, and include pre-release defects (PRE) and code churn (i.e, changes to the code) (Bird et al., 2011). As such, in this research question, we build two sets of models: those based on

static topic-based metric, and those based on historical topic-based metrics. (For a more detailed discussion about the analysis approach that is used in this research question, please refer to Section 2.3.) For a baseline static metric, we choose LOC, because LOC is a good general software metric and has been used for benchmarking (DAmbros et al., 2010; Rosenberg, 1997). For baseline historical metrics, we choose PRE and code churn because they are a good measurement for defects (, n.d.; Biyani and Santhanam, 1998), and have also been used as a baseline model for comparing metrics (Bird et al., 2011). Moreover, LOC, PRE, and code churn have been shown to be the best metrics for explaining defects, and researchers in Empirical Software Engineering should compare their metrics with them (DAmbros et al., 2010; Gyimothy et al., 2005; Oram and Wilson, 2010).

Our goal here is not to predict post-release defects. Instead, we want to see how much improvement on explaining deviance (i.e., model fitness) in defects our topic-based metrics can bring to the baseline metrics.

We use logistic regression with post-release defects as our dependent variable, and report the percent deviance explained (D^2) for each combination of independent variables (i.e., metric combinations). (To eliminate any skew in the metric values, we apply a log transformation on the metrics.) Here, the D^2 measure is similar to the adjusted R^2 measure in linear regression, except that D^2 quantifies how much deviance a logistic regression model can explain.

Interpreting Results. A higher D^2 value generally indicates a better model fit, but when the number of independent variables is large, D^2 may not be a good measure. As the number of independent variables increases, D^2 will always increase regardless of the quality of the model. Thus, we also use another measure called

the Akaike information criterion (AIC). AIC can be used to compare the fitness of different models, and it penalizes more complex models (Burnham Kenneth P., 2004; Raftery, 1995). Models with lower AIC scores are better.

Recall that by the definition of our TM and DTM metrics (Equations 3.4 and 3.7), each metric will produce many values for each file (K values in the case of TM, and $|B|$ values in the case of DTM). To avoid the problems of overfitting and multicollinearity, we use Principal Component Analysis (PCA) to reduce the dimensionality of the metrics (Jolliffe, 2002). PCA transforms the data into a smaller set of uncorrelated variables while still capturing the patterns of the original data (Jolliffe, 2002). We choose the principal components (PCs) until either 90% of the variances are explained, or when the increase in variance explained by adding a new PC is less than the mean variance explained of all PCs (Jolliffe, 2002).

We perform stepwise regression on the PCs of TM and DTM metrics to make our model more robust (Cureton and D'Agostino, 1993; Haan, 1977). Stepwise regression is a variable selection approach, which adds or removes variables to the model according to some criteria, which, in this thesis, we choose to use the AIC score.

Results

We present the results in Tables 3.9–3.10 and 3.11–3.12. Table 3.9 shows the results for static topic-based metric. We find that adding NT gives a significant improvement in the deviance explained. All models have statistically significant (p-value $\leq$ 0.05) improvement when NT is added to the model. In all the versions of Mylyn, Firefox, and NetBeans, NT gives at least an 18% increase in D^2 (Figure 3.3 and

Table 3.9: D^2 improvement and AIC scores for static software metrics. The higher the D^2 the better the explanatory power; the lower the AIC scores the better the explanatory power. Numbers in the parentheses are the D^2 improvement or AIC score decreasement in percentage of the base model. The best model of each version of the software is marked in bold.

System	Model	D^2	AIC
Mylyn 1.0	Base(LOC)	0.09	1047.36
	Base+NT	0.14 (+56%)	990.85 (-5%)
	Base+TM	***0.21 (+133%)***	***957.83 (-9%)***
Mylyn 2.0	Base(LOC)	0.14	1078.35
	Base+NT	0.19 (+36%)	1020.46 (-5%)
	Base+TM	***0.27 (+93%)***	***956.73 (-11%)***
Mylyn 3.0	Base(LOC)	0.13	1159.74
	Base+NT	0.20 (+54%)	1071.71 (-8%)
	Base+TM	***0.34 (+162%)***	***949.17 (-18%)***
Firefox 1.0	Base(LOC)	0.12	2374.85
	Base+NT	0.16 (+33%)	2256.41 (-5%)
	Base+TM	***0.20 (+67%)***	***2168.37 (-9%)***
Firefox 1.5	Base(LOC)	0.15	3474.84
	Base+NT	0.21 (+40%)	3241.09 (-7%)
	Base+TM	***0.28 (+87%)***	***2969.09 (-15%)***
Firefox 2.0	Base(LOC)	0.14	2224.76
	Base+NT	0.17 (+18%)	2152.19 (-3%)
	Base+TM	***0.24 (+71%)***	***1973.27 (-11%)***

Continued in Table 3.10.

Table 3.10: Continued from Table 3.9. D^2 improvement and AIC scores for static software metrics. The higher the D^2 the better the explanatory power; the lower the AIC scores the better the explanatory power. Numbers in the parentheses are the D^2 improvement or AIC score decreasement in percentage of the base model. The best model of each version of the software is marked in bold.

System	Model	D^2	AIC
Eclipse 2.0	Base(LOC)	0.18	4584.26
	Base+NT	0.18 (+0%)	4575.72 (-0%)
	Base+TM	***0.29 (+61%)***	***4003.58 (-13%)***
Eclipse 2.1	Base(LOC)	0.11	4804.87
	Base+NT	0.11 (+0%)	4793.48 (-0%)
	Base+TM	***0.28 (+87%)***	***2969.09 (-15%)***
Eclipse 3.0	Base(LOC)	0.14	7591.93
	Base+NT	0.14 (+0%)	7589.50 (-0%)
	Base+TM	***0.24 (+71%)***	***6800.06 (-10%)***

System	Model	D^2	AIC
NetBeans 4.0	Base(LOC)	0.07	1529.27
	Base+NT	0.10 (+43%)	1476.20 (-3%)
	Base+TM	***0.29 (+314%)***	***1196.67 (-22%)***
NetBeans 5.0	Base(LOC)	0.07	1650.94
	Base+NT	0.10 (+43%)	1594.86 (-3%)
	Base+TM	***0.19 (+171%)***	***1477.02 (-11%)***
NetBeans 5.5.1	Base(LOC)	0.07	4916.00
	Base+NT	0.11 (+57%)	4727.63 (-4%)
	Base+TM	***0.17 (+143%)***	***4417.73 (-10%)***

Table 3.11: D^2 improvement and AIC scores for historical software metrics. The higher the D^2 the better the explanatory power; the lower the AIC scores the better the explanatory power. Numbers in the parentheses are the D^2 improvement or AIC score decreasement in percentage of the base model. The best model of each version of the software is marked in bold. NDT is statistically significant in all systems except Mylyn 3.0, Eclipse 2.1, and Eclipse 3.0.

System	Model	D^2	AIC
Mylyn 1.0	Base(PRE+Churn)	0.21	917.38
	Base+NDT	0.24 (+14%)	885.72 (-3%)
	Base+DTM	***0.30 (+43%)***	***824.24 (-10%)***
Mylyn 2.0	Base(PRE+Churn)	0.22	987.04
	Base+NDT	0.23 (+4%)	971.01 (-2%)
	Base+DTM	***0.34 (+55%)***	***882.19 (-11%)***
Mylyn 3.0	Base(PRE+Churn)	0.28	957.31
	Base+NDT	0.29 (+4%)	955.98 (-0%)
	Base+DTM	***0.36 (+29%)***	***909.83 (-5%)***

System	Model	D^2	AIC
Firefox 1.0	Base(PRE+Churn)	0.14	2299.70
	Base+NDT	0.18 (+29%)	2204.47 (-4%)
	Base+DTM	***0.20 (+43%)***	***2152.08 (-6%)***
Firefox 1.5	Base(PRE+Churn)	0.20	3255.54
	Base+NDT	0.25 (+25%)	3081.35 (-5%)
	Base+DTM	***0.27 (+35%)***	***3005.55 (-8%)***
Firefox 2.0	Base(PRE+Churn)	0.23	2008.67
	Base+NDT	0.25 (+9%)	1951.41 (-3%)
	Base+DTM	***0.28 (+22%)***	***1892.53 (-6%)***

Continued in Table 3.12.

Table 3.12: Continued from Table 3.11. D^2 improvement and AIC scores for historical software metrics. The higher the D^2 the better the explanatory power; the lower the AIC scores the better the explanatory power. Numbers in the parentheses are the D^2 improvement or AIC score decreasement in percentage of the base model. The best model of each version of the software is marked in bold. NDT is statistically significant in all systems except Mylyn 3.0, Eclipse 2.1, and Eclipse 3.0.

System	Model	D^2	AIC
Eclipse 2.0	Base(PRE+Churn)	0.17	4605.37
	Base+NDT	0.20 (+18%)	4477.51 (-3%)
	Base+DTM	***0.30 (+76%)***	***3930.40 (-15%)***
Eclipse 2.1	Base(PRE+Churn)	0.15	4586.50
	Base+NDT	0.15 (+0%)	4586.19 (-0%)
	Base+DTM	***0.19 (+27%)***	***4366.09 (-5%)***
Eclipse 3.0	Base(PRE+Churn)	0.17	7310.04
	Base+NDT	0.17 (+0%)	7309.30 (-0%)
	Base+DTM	***0.24 (+41%)***	***6729.03 (-8%)***

System	Model	D^2	AIC
NetBeans 4.0	Base(PRE+Churn)	0.28	1182.89
	Base+NDT	***0.31 (+11%)***	***1138.08 (-4%)***
	Base+DTM	0.31 (+11%)	1145.90 (-3%)
NetBeans 5.0	Base(PRE+Churn)	0.14	1524.13
	Base+NDT	0.17 (+21%)	1470.69 (-4%)
	Base+DTM	***0.21 (+50%)***	***1418.24 (-7%)***
NetBeans 5.5.1	Base(PRE+Churn)	0.13	4612.55
	Base+NDT	0.17 (+31%)	4366.69 (-5%)
	Base+DTM	***0.19 (+46%)***	***4306.96 (-7%)***

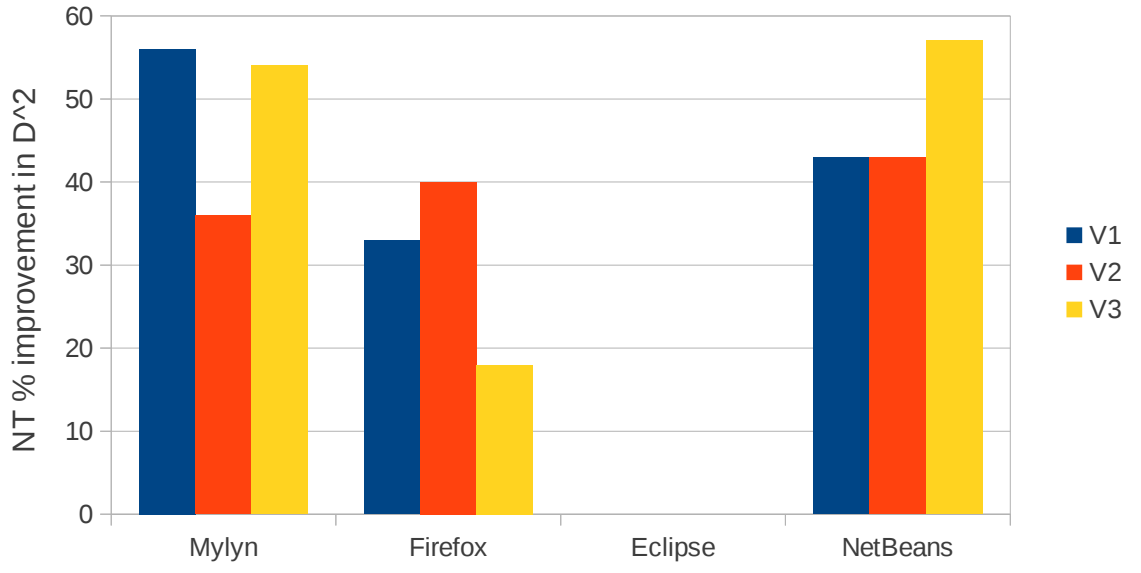


Figure 3.3: Percentage improvement in D^2 when NT is added to the base model.

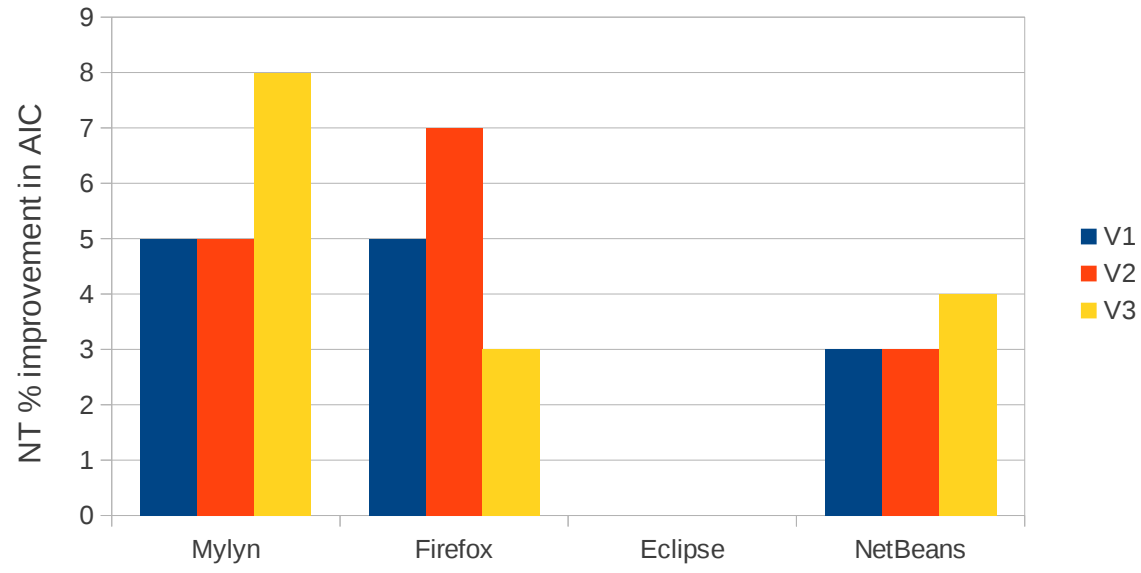


Figure 3.4: Percentage improvement in AIC when NT is added to the base model.

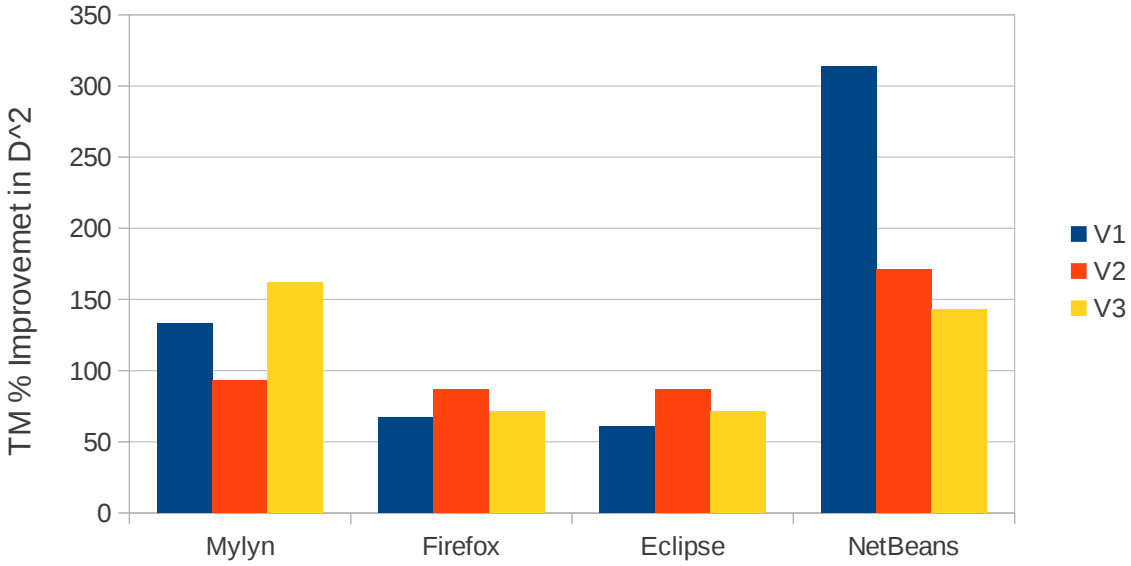


Figure 3.5: Percentage improvement in D^2 when TM is added to the base model.

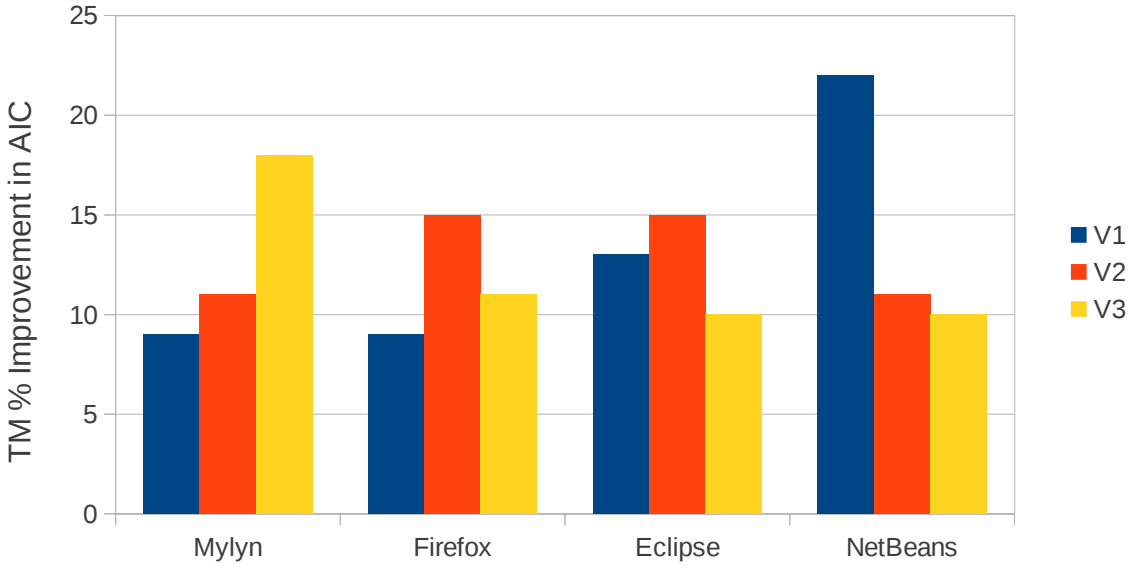


Figure 3.6: Percentage improvement in AIC when TM is added to the base model.

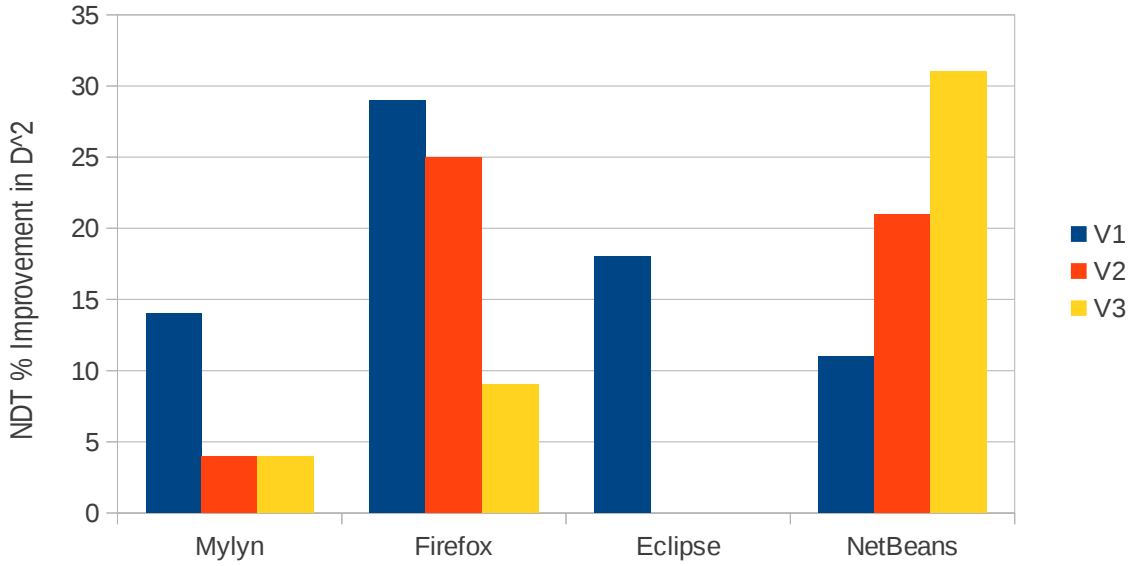


Figure 3.7: Percentage improvement in D^2 when NDT is added to the base model.

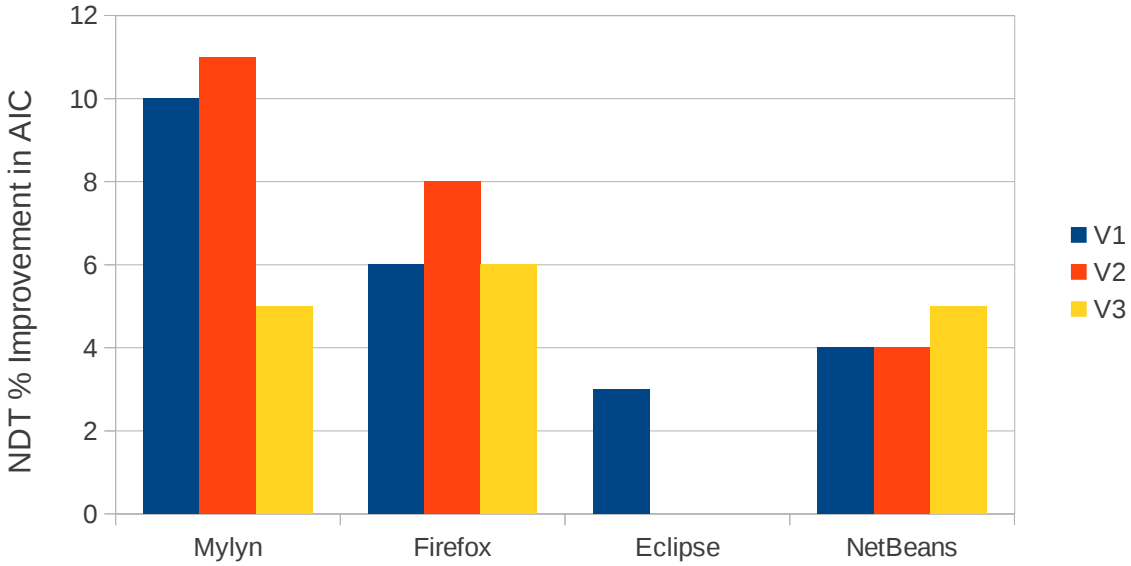


Figure 3.8: Percentage improvement in AIC when NDT is added to the base model.

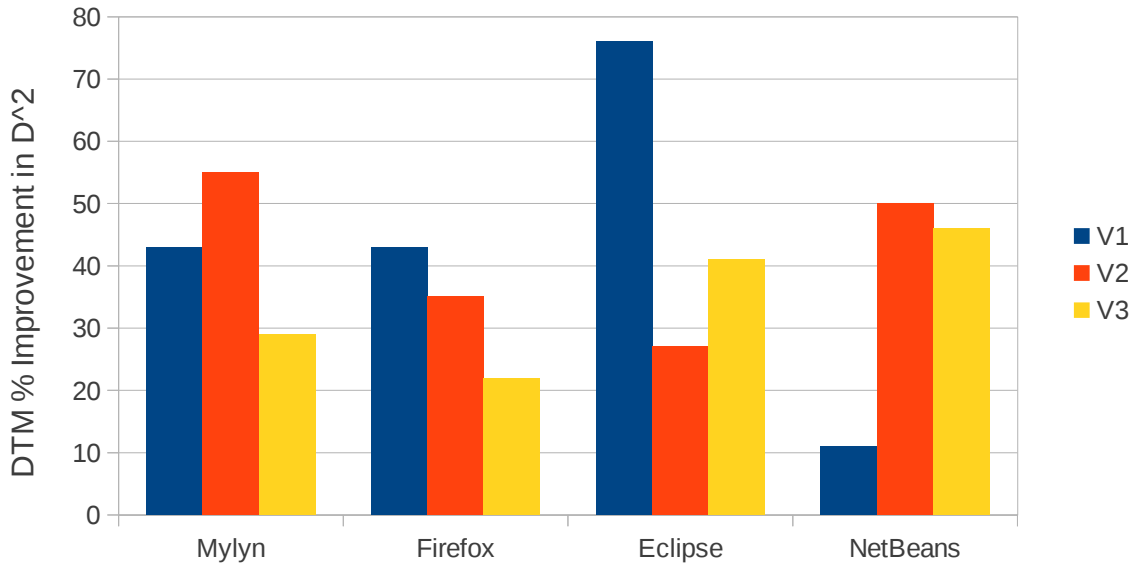


Figure 3.9: Percentage improvement in D^2 when DTM is added to the base model.

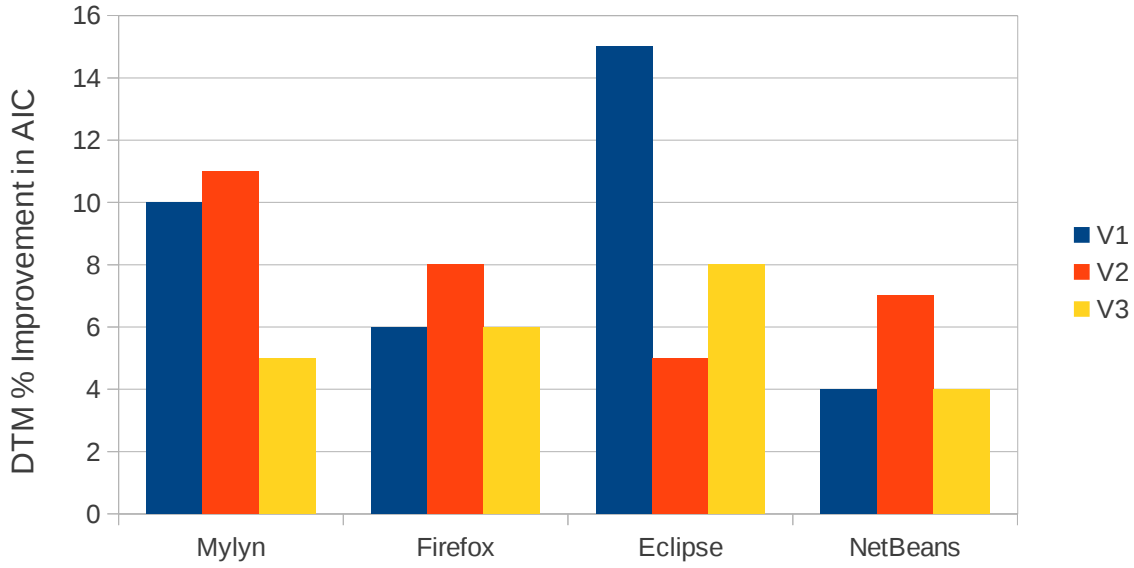


Figure 3.10: Percentage improvement in AIC when DTM is added to the base model.

3.4). However, we find that the performance of NT is not as high in Eclipse.

Both TM and DTM, on the other hand, give significant improvements in all three subject systems. We find that the TM metric improves the deviance explained by 61–314%, compared to the baseline model (Figure 3.5 and 3.6).

Table 3.11 – 3.12 shows the results for historical topic-based metrics. We find that NDT gives a promising improvement in D^2 over the base model (Figure 3.7 and 3.8). This implies that topics with high pre-release defects are more likely to have post-release defects, and having more defect-prone topics may have negative effects on the code quality. The improvement of NDT is not as large for Eclipse 2.1 and 3.0. However, the improvements achieved by the DTM metric are consistent across all versions of all systems. DTM also help explain defects more than NDT, except for NetBeans 4.0 where NDT has a lower AIC score than that of DTM. We find that, overall, DTM improves the explanatory power over the baseline model by 11–76% (Figure 3.9 and 3.10).

Discussion

One possible explanation as to why the improvement of NDT in Eclipse is not as high as in the other systems is because topics in Eclipse have higher defect density (Table 3.3). Since topics are generally more defect-prone, the overall explanatory power of NDT decreases. On the other hand, DTM contains a more general information about all the defect-prone topics, which better explains defects.

To see the effects of NT and NDT in our logistic regression models, Table 3.13 shows the average coefficients of these metrics across the three versions of each

Table 3.13: Average value of the regression coefficients of NT and NDT metrics.

	Mylyn	Firefox	Eclipse	NetBeans
NT	1.73	1.26	0.06	1.32
NDT	1.71	0.85	0.24	0.90

subject system. Both NT and NDT have positive coefficients in all the subject systems, which implies that as the number of topics or defect-prone topics increases, a file will have higher chances to be defect-prone. However, the coefficients of NDT in Firefox, Eclipse, and NetBeans, and NT in Eclipse are smaller than one, which means the effects are relatively minor.

Our findings show that the number of topics in a file has a strong relationship with defects, and files having more defect-prone topics will more likely to be defect-prone.

All of our proposed topic-based metrics can help improve the baseline metrics in terms of defect explanatory power. We find that, the more topics a file has, the higher the chance that the file may be defect-prone; and the more defect prone topics a file has, the higher the chance that the file may be defect prone.

3.3.4 RQ4: How do our metrics compare with other topic-based cohesion and coupling metrics?

Maintaining a high cohesion and low coupling among software files during development can help reduce maintenance costs and improve reliability of a software system (Fenton, 1991; Macro and Buxton, 1987). Researchers have used various

software structures, such as interactions among variables and methods, to measure cohesion and coupling in software systems (Allen and Khoshgoftaar, 1999; Bieman and Kang, 1998; Briand et al., 1998; Chae et al., 2000). Recently, many studies have measured cohesion and coupling using a different approach, namly topic models (Chen et al., 2012; Gethers and Poshyvanyk, 2010; Liu et al., 2009; Marcus and Poshyvanyk, 2005; Marcus et al., 2008; Poshyvanyk and Marcus, 2006; Ujhazi et al., 2010). These approaches measure cohesion and coupling using the concern similarity or scattering in source code files. The NT metric that we propose in Section 3.2.1 also measures the level of cohesion of a source code file, i.e., more topics implies low cohesion, and in RQ3 we find that this metric is statistical significant when explaining software defects.

In this RQ, we compare the defect explanatory power of state-of-the-art topic-based cohesion and coupling metrics with our metric, NT. We do not include NDT in the comparison because NDT is considering the defect history of topics, and current state of the art only considers static information (i.e., source code). In addition, TM and DTM contain more than one variable. If we compare these three topic based metrics with current state of the art, the comparison will be unfair.

By identifying which metrics perform best, and whether the metrics are highly correlated with each other, practitioners can use the most effective combination of metrics and avoid possible problems of overfitting and multicollinearity. State-of-the-art topic-based cohesion and coupling metrics are already outperforming other traditional cohesion and coupling metrics, so we do not include traditional cohesion and coupling metrics in our study (Gethers and Poshyvanyk, 2010; Liu et al., 2009; Marcus and Poshyvanyk, 2005; Marcus et al., 2008; Poshyvanyk and Marcus, 2006;

Table 3.14: Summary of the topic-based cohesion and coupling metrics that we compare in our study. Granularity indicates at which level the metric is computed.

Metric	Type	Granularity	Topic Model	Preferred Metric Value	Cited Paper
CLCOM5	cohesion	method	LSI	low	(Ujhazi et al., 2010)
CCBO	coupling	file	LSI	low	(Ujhazi et al., 2010)
RTC_s	coupling	file	RTM	low	(Gethers and Poshyvanyk, 2010)
MWE	cohesion	method	LDA	high	(Liu et al., 2009)
NT	cohesion	file	LDA	low	This Chapter

Ujhazi et al., 2010). In addition, we can study whether NT can explain more defects than the current state of the art. The goal of this research question is to compare the defect explanatory power of our NT metric with existing topic-based cohesion and coupling metrics, and help practitioners decide which metrics should be included when analyzing cohesion and coupling of software systems using topic models.

Approach

We want to compare NT (Equation 3.3) with the cohesion and coupling metrics proposed by Gethers et al. (Gethers and Poshyvanyk, 2010), Liu et al. (Liu et al., 2009), and Ujhazi et al. (Ujhazi et al., 2010), since these metrics use different topic modeling approaches. Moreover, because these metrics analyse cohesion and coupling based on topics, the results might have some overlap and some metrics might be highly correlated to others. Therefore, we examine whether there are correlated, and how these metrics differ in defect explanatory power. Table 3.14 shows the summary of these metrics, and we briefly describe these metrics below.

Measuring Cohesion and Coupling using Latent Semantic Indexing. Ujhazi et al. measure conceptual cohesion using a topic modelling approach called Latent Semantic Indexing (LSI) (Ujhazi et al., 2010) (for more information about LSI, see Section 2.1.2).

Ujhazi et al. propose a cohesion metric called CLCOM5, which is defined by computing the cosine similarity among all the *methods* within a source code file (i.e., each document in LSI is a method, and the corpus is the file). If a method is very similar to many other methods in the same file, then this file has low cohesion. If the similarity score between each pair of methods within the same file is larger than some threshold, then a score of one is added to the file (i.e., the file's coupling score plus one). The authors find the optimal threshold for the cosine similarity score to be between 0.7 to 0.8 through empirical analysis. In this thesis, we use the average of these thresholds, 0.75.

Ujhazi et al. also propose a coupling metric called CCBO at the *file level* (i.e., each document in LSI is a file, and the corpus is all the files in the software system). If a file is similar to many other files in the same software system, then this file is highly coupled. The authors compute the cosine similarity among all the source code files in a software system, and assign a coupling score of one if the score is larger than a threshold (same threshold range as that in CLCOM5).

Measuring Coupling using Relational Topic Models Gethers et al. (Gethers and Poshyvanyk, 2010) propose a coupling metric using a variant of LDA, called relational topic model (RTM) (Chang and Blei, 2009) (see Section 2.1.3). RTM predicts the linking probability among documents using the underlying topics, so if the topics in a file is very similar to many other files in the same system (i.e., have high

linking probability), then this file is highly coupled. Let RTC(f_1, f_2) (Relational Topic-based Coupling between f_1 and f_2) be the probability that a link exists between file f_1 and f_2. Gethers et al. then define RTC$_s$ for a source code file f_i as

$$\text{RTC}_s(f_i) = 1/n * \sum_{j \in F} \text{RTC}(f_i, f_j), \tag{3.8}$$

where n is the number of files in the system, and F is the set of all source code files.

Measuring Cohesion using Topic Distributions and Occupancies Liu et al. propose a cohesion metric using LDA (see Section 2.1.1), called Maximal Weighted Entropy (MWE) (Liu et al., 2009), to measure the level of cohesion in software systems at the *method level*. For each topic, Liu et al. compute the topic occupancy and distribution in a source code file, and the MWE is the product of the maximum occupancy and distribution among all topics. The occupancy of topic z_i in *all the methods* of a source code file is defined as

$$O(z_i) = 1/m(f) * \sum_{j=1}^{m(f)} \theta_{ij}, \tag{3.9}$$

where $m(f)$ is the number of methods in the source code file f, and θ_{ij} is the topic membership value of topic z_i in method j. Occupancy measures the average of a topic's membership value across all methods in a file (on average, how much does this topic exist in this file).

Liu et al. use information entropy to measure the distribution of topics in a source code file. Information entropy measures the uncertainty of the topics (Ihara, 1993), so if a topic is distributed uniformly in the methods of a file, then the entropy

value will be low; otherwise, the entropy value will be high. Topic distribution of topic z_i in a file is defined as

$$D(t_i) = 1/log(m(f)) * E(z_i), \tag{3.10}$$

where $E(z_i)$ is the information entropy value for z_i computed across all the methods in a file.

Therefore, MWE of a source code file f is defined as

$$\text{MWE}(f) = max_{i \in Z}(O(z_i) * D(z_i)), \tag{3.11}$$

where Z is the set of topics in a file.

Implementation and Experiment Procedures We implement and compute all of the above-mentioned metrics using K = 500 and II = 10,000, the same as our previous research questions. These parameters may slightly affect the results, but we want to do a controlled experiment where the parameters are the same. For implementing RTC$_s$, we use the recommended α (50/K)and β (0.1) values as in the software package that Gethers et al. use. We use the MALLET optimized α and β for implementing MWE. The experiment procedures are as follows:

- We first examine the pair-wise correlation between all topic-based metrics and LOC, and study whether these topic-based metrics are capturing different information than LOC. We use LOC as our baseline metric and remove any metrics that are highly correlated to LOC, because most software complexity metrics are correlated with LOC, and LOC is one of the best metrics for

explaining defects (Oram and Wilson, 2010).

- We study how much D^2 and AIC improvement each metric gives over the base model. Again, we use LOC as the baseline metric, since these metrics are all using a single snapshot of the current software system.

- We perform PCA analysis on LOC and all the topic-based cohesion and coupling, except NT. We use the resulting PCs as the base model, and examine the D^2 improvement when NT is added to the model. By doing this experiment, we can examine whether NT can bring any additional improvements to all of LOC and state-of-the-art topic-based cohesion and coupling metrics when explaining defects.

Results and Discussion

Correlation Analysis. LOC is one of the most effective metrics for explaining code quality, and all software complexity metrics are highly correlated with LOC (Oram and Wilson, 2010). Prior research also recommend further study to compare their metrics with LOC (Oram and Wilson, 2010). Therefore, we use LOC as a baseline metric, and study if the topic-based metrics are bringing new information to LOC. We perform a correlation analysis to remove any metrics that are highly correlated with LOC (0.6) to ensure that all the topic-based metrics are capturing different information than LOC.

In Table 3.15 – 3.18, we see that CLCOM5 is highly correlated with LOC in every subject system (0.65 – 0.77). Due to such high correlation with the baseline metric, we remove CLCOM5 in the future analysis. We also find that NT has a relatively high correlation with MWE (-0.01 – -0.65). However, these two metrics

are not highly correlated with LOC. In addition, even if they are highly correlated, one metric may still outperform the other when explaining defects, so we will keep these two metrics and perform a more detailed comparison. Furthermore, NT is considerably simpler metric to measure compared to MWE, and we are curious about NT's overall performance compared to MWE, the more complex metric.

Improvement in Defect Explanatory Power of Each Topic-base Cohesion and Coupling Metric. Table 3.19 – 3.22 shows the improvement of each metric over the base model (LOC) in all the subject systems. Our NT metric generally gives the greatest improvement (on average 30%), except for Eclipse 2.0 and 3.0, and NetBeans 4.0; however, no topic-based cohesion and coupling metrics give improvement in these systems. Note that, since some of the metrics are computed at the method level (then aggregated to obtain a file level metric), source code files that do not have methods (i.e., interfaces) are excluded, and as a result the D^2 and AIC score for the base model are slightly different from that of RQ3 (dataset is slightly different due to exclusion of some files that do not have methods). We find that 364 files were excluded from all versions of Mylyn (a total of 37 defects are excluded); 790 files were excluded from all versions of Firefox (17 defects); 6,236 files were excluded from all versions of Eclipse (330 defects); and 9,143 files were excluded from all versions of NetBeans (469 defects).

MWE gives the second best improvement over the base model (12%). RTC$_s$, which gives 6% improvement on average, is not statistically different from the base model in most of the subject systems: low level of statistical significance indicates that the effect of RTC$_s$ likely happened by chance. CCBO gives about 3% improvement, but is not statistically significant in 7 out of 12 versions of the four software

Table 3.15: Pair-wise correlation between all topic-based cohesion and coupling metrics and LOC. "*" indicates a p-value < 0.05. "**" indicates a p-value < 0.01. "***" indicates a p-value < 0.001.

	Mylyn 1.0				
	NT	MWE	RTCs	CCBO	CLCOM5
NT	—	—	—	—	—
MWE	-0.48***	—	—	—	—
RTCs	0.35***	-0.34***	—	—	—
CCBO	-0.16***	0.23***	-0.07	—	—
CLCOM5	0.12**	-0.04	-0.04	-0.12**	—
LOC	0.33***	-0.39***	0.01	-0.19***	0.65***
	Mylyn 2.0				
	NT	MWE	RTCs	CCBO	CLCOM5
NT	—	—	—	—	—
MWE	-0.48***	—	—	—	—
RTCs	-0.01	-0.03	—	—	—
CCBO	-0.18***	0.17***	0.05	—	—
CLCOM5	0.10**	-0.04	-0.07*	-0.09**	—
LOC	0.33***	-0.39***	-0.04	-0.18***	0.66***
	Mylyn 3.0				
	NT	MWE	RTCs	CCBO	CLCOM5
NT	—	—	—	—	—
MWE	-0.51***	—	—	—	—
RTCs	0.00	-0.02	—	—	—
CCBO	-0.31***	0.17***	0.01	—	—
CLCOM5	0.11***	-0.01	0.00	-0.08*	—
LOC	0.37***	-0.36***	0.01	-0.17***	0.68***

Continued in Table 3.16.

Table 3.16: *Continued from Table 3.15*. Pair-wise correlation between all topic-based cohesion and coupling metrics and LOC. "*" indicates a p-value < 0.05. "**" indicates a p-value < 0.01. "***" indicates a p-value < 0.001.

	Firefox 1.0				
	NT	MWE	RTCs	CCBO	CLCOM5
NT	—	—	—	—	—
MWE	-0.65***	—	—	—	—
RTCs	-0.24***	0.39***	—	—	—
CCBO	-0.56***	0.55***	0.47***	—	—
CLCOM5	0.27***	-0.25***	-0.33***	-0.21***	—
LOC	0.44***	-0.53***	-0.47***	-0.38***	0.77***
	Firefox 1.5				
	NT	MWE	RTCs	CCBO	CLCOM5
NT	—	—	—	—	—
MWE	-0.65***	—	—	—	—
RTCs	-0.02	0.02	—	—	—
CCBO	-0.55***	0.54***	0.02	—	—
CLCOM5	0.26***	-0.24***	-0.01	-0.19***	—
LOC	0.41***	-0.49***	0.00	-0.33***	0.77***
	Firefox 2.0				
	NT	MWE	RTCs	CCBO	CLCOM5
NT	—	—	—	—	—
MWE	-0.65***	—	—	—	—
RTCs	0.01	-0.01	—	—	—
CCBO	-0.55***	0.54***	-0.01	—	—
CLCOM5	0.26***	-0.24***	-0.01	-0.19***	—
LOC	0.41***	-0.49***	-0.01	-0.32***	0.77***

Continued in Table 3.17.

Table 3.17: *Continued from Table 3.16*. Pair-wise correlation between all topic-based cohesion and coupling metrics and LOC. "*" indicates a p-value < 0.05. "**" indicates a p-value < 0.01. "***" indicates a p-value < 0.001.

	Eclipse 2.0				
	NT	MWE	RTCs	CCBO	CLCOM5
NT	—	—	—	—	—
MWE	-0.48***	—	—	—	—
RTCs	0.33***	-0.36***	—	—	—
CCBO	-0.49***	0.33***	-0.12***	—	—
CLCOM5	0.02	-0.04**	0.04**	0.04**	—
LOC	0.27***	-0.41***	0.22***	-0.15***	0.72***
	Eclipse 2.1				
	NT	MWE	RTCs	CCBO	CLCOM5
NT	—	—	—	—	—
MWE	-0.50***	—	—	—	—
RTCs	0.00	0.00	—	—	—
CCBO	-0.48***	0.32***	0.01	—	—
CLCOM5	0.06***	-0.03*	0.01	0.04**	—
LOC	0.32***	-0.39***	0.02	-0.15***	0.73***
	Eclipse 3.0				
	NT	MWE	RTCs	CCBO	CLCOM5
NT	—	—	—	—	—
MWE	-0.49***	—	—	—	—
RTCs	0.00	0.00	—	—	—
CCBO	-0.40***	0.35***	0.03*	—	—
CLCOM5	0.08***	-0.02	0.02	0.05***	—
LOC	0.33***	-0.38***	0.01	-0.15***	0.73***

Continued in Table 3.18.

Table 3.18: *Continued from Table 3.17.* Pair-wise correlation between all topic-based cohesion and coupling metrics and LOC. "*" indicates a p-value < 0.05. "**" indicates a p-value < 0.01. "***" indicates a p-value < 0.001.

	NetBeans 4.0				
	NT	MWE	RTCs	CCBO	CLCOM5
NT	—	—	—	—	—
MWE	0.01	—	—	—	—
RTCs	0.03	-0.30***	—	—	—
CCBO	0.00	0.18***	-0.01	—	—
CLCOM5	0.05**	-0.01	0.08***	0.04	—
LOC	0.07***	-0.32***	0.22***	-0.06**	0.69***
	NetBeans 5.0				
	NT	MWE	RTCs	CCBO	CLCOM5
NT	—	—	—	—	—
MWE	-0.40***	—	—	—	—
RTCs	0.01	-0.01	—	—	—
CCBO	-0.33***	0.10***	-0.01	—	—
CLCOM5	0.10***	-0.02	0.00	0.00	—
LOC	0.31***	-0.34***	-0.01	0.01	0.69***
	NetBeans 5.5.1				
	NT	MWE	RTCs	CCBO	CLCOM5
NT	—	—	—	—	—
MWE	-0.47***	—	—	—	—
RTCs	-0.02	0.00	—	—	—
CCBO	-0.34***	0.15***	-0.01	—	—
CLCOM5	0.11***	0.01	0.01	-0.02*	—
LOC	0.35***	-0.31***	0.01	-0.05***	0.69***

Table 3.19: D^2 improvement and AIC scores for topic-based cohesion and coupling metrics. Numbers in the parentheses are the D^2 increase or AIC score decrease in percentage of the base model. The best model of each version of the software is marked in bold. * indicates the metric is statistically significant (i.e., p-value < 0.05).

System	Model	D^2	AIC
Mylyn 1.0	Base(LOC) *	0.06	958.06
	***Base+NT** *	***0.09 (+50%)***	***924.94 (-3%)***
	Base+CCBO	0.06 (+0%)	959.25 (-0%)
	Base+RTC$_s$ *	0.09 (+50%)	930.94 (-3%)
	Base+MWE *	0.08 (+33%)	940.42 (-2%)
Mylyn 2.0	Base(LOC) *	0.10	947.11
	***Base+NT** *	***0.14 (+40%)***	***908.48 (-4%)***
	Base+CCBO	0.10 (+0%)	949.02 (-0%)
	Base+RTC$_s$	0.10 (+0%)	949.10 (-0%)
	Base+MWE *	0.12 (+20%)	922.96 (-3%)
Mylyn 3.0	Base(LOC) *	0.11	1075.17
	***Base+NT** *	***0.17 (+55%)***	***1004.44 (-7%)***
	Base+CCBO *	0.13 (+18%)	1059.42 (-1%)
	Base+RTC$_s$	0.11 (+0%)	1077.05 (-0%)
	Base+MWE *	0.13 (+18%)	1057.65 (-2%)

Continued in Table 3.20.

Table 3.20: Continued from Table 3.19. D^2 improvement and AIC scores for topic-based cohesion and coupling metrics. Numbers in the parentheses are the D^2 increase or AIC score decrease in percentage of the base model. The best model of each version of the software is marked in bold. * indicates the metric is statistically significant (i.e., p-value < 0.05).

System	Model	D^2	AIC
Firefox 1.0	Base(LOC) *	0.12	2330.13
	***Base+NT** *	***0.17 (+42%)***	***2189.26 (-6%)***
	Base+CCBO *	0.13 (+8%)	2299.17 (-0%)
	Base+RTC$_s$ *	0.13 (+8%)	2300.87 (-0%)
	Base+MWE *	0.15 (+25%)	2253.79 (-3%)
Firefox 1.5	Base(LOC) *	0.15	3399.67
	***Base+NT** *	***0.22 (+47%)***	***3114.63 (-8%)***
	Base+CCBO	0.15 (+0%)	3399.25 (-0%)
	Base+RTC$_s$	0.15 (+0%)	3400.12 (-0%)
	Base+MWE *	0.17 (+13%)	3305.37 (-3%)
Firefox 2.0	Base(LOC) *	0.14	2166.52
	***Base+NT** *	***0.18 (+29%)***	***2069.98 (-4%)***
	Base+CCBO	0.14 (+0%)	2168.13 (-0%)
	Base+RTC$_s$	0.14 (+0%)	2167.94 (-0%)
	Base+MWE *	0.17 (+21%)	2099.11 (-3%)

Continued in Table 3.21.

Table 3.21: Continued from Table 3.20. D^2 improvement and AIC scores for topic-based cohesion and coupling metrics. Numbers in the parentheses are the D^2 increase or AIC score decrease in percentage of the base model. The best model of each version of the software is marked in bold. * indicates the metric is statistically significant (i.e., p-value < 0.05).

System	Model	D^2	AIC
Eclipse 2.0	Base(LOC) *	0.12	4197.38
	Base+NT *	0.13 (+8%)	4174.70 (-0%)
	***Base+CCBO** *	***0.13 (+8%)***	***4155.80 (-0%)***
	Base+RTC$_s$ *	0.13 (+8%)	4194.66 (-0%)
	Base+MWE *	0.13 (+8%)	4196.62 (-0%)
Eclipse 2.1	Base(LOC) *	0.10	4110.59
	Base+NT *	0.10 (+0%)	4106.49 (-0%)
	Base+CCBO	0.10 (+0%)	4112.54 (-0%)
	Base+RTC$_s$	0.10 (+0%)	4111.78 (-0%)
	***Base+MWE** *	***0.10 (+0%)***	***4096.07 (-0%)***
Eclipse 3.0	Base(LOC) *	0.11	6717.16
	Base+NT	0.11 (+0%)	6719.12 (-0%)
	Base+CCBO *	0.11 (+0%)	6710.12 (-0%)
	Base+RTC$_s$	0.11 (+0%)	6718.76 (-0%)
	***Base+MWE** *	***0.11 (+0%)***	***6680.56 (-0%)***

Continued in Table 3.22.

Table 3.22: Continued from Table 3.21. D^2 improvement and AIC scores for topic-based cohesion and coupling metrics. Numbers in the parentheses are the D^2 increase or AIC score decrease in percentage of the base model. The best model of each version of the software is marked in bold. * indicates the metric is statistically significant (i.e., p-value < 0.05).

System	Model	D^2	AIC
NetBeans 4.0	Base(LOC) *	0.04	1062.50
	Base+NT	0.04 (+0%)	1064.35 (-0%)
	***Base+CCBO** *	***0.04 (+0%)***	***1058.13 (-0%)***
	Base+RTC$_s$	0.04 (+0%)	1064.50 (-0%)
	Base+MWE *	0.04 (+0%)	1064.35 (-0%)
NetBeans 5.0	Base(LOC) *	0.06	871.55
	***Base+NT** *	***0.08 (+33%)***	***849.41 (-3%)***
	Base+CCBO	0.06 (+0%)	871.37 (-0%)
	Base+RTC$_s$	0.06 (+0%)	873.39 (-0%)
	Base+MWE	0.06 (+0%)	867.02 (-0%)
NetBeans 5.5.1	Base(LOC) *	0.04	3616.50
	***Base+NT** *	***0.08 (+50%)***	***3474.94 (-4%)***
	Base+CCBO *	0.04 (+0%)	3613.90 (-0%)
	Base+RTC$_s$	0.04 (+0%)	3618.25 (-0%)
	Base+MWE *	0.04 (+0%)	3605.71 (-0%)

systems.

From the results, we can see that NT outperforms other metrics whenever topic-based cohesion and coupling metrics can help explain software defects. Also, NT is statistically significant in most of the systems. This implies the effect of NT most likely does not happen by chance, and is more stable than other topic-based cohesion and coupling metrics. Another advantage of NT is that NT can be applied to source code files that do not have any methods, whereas MWE can only work on the files that have methods. In our case study, hundreds of defects are removed in Eclipse and NetBeans because these defects are in the files that do not have any methods. If we use MWE, then these defects will be ignored. NT is also more intuitive and much simpler to measure than other metrics. By studying how many topics a source code file has, practitioners can determine if redesign of such file is required to increase file cohesion and decrease maintenance difficulty.

NT gives the best improvement over the base model in terms of helping explain defects, and is statistically significant in most systems. NT is easier to interpret and much simpler to implement than other metrics, which can also help practitioners during maintenance (i.e., identify and redesign source code files that contain more topics).

Does NT give improvements to state-of-the-art when explaining defects?

In Table 3.19 – 3.22, we have shown how each metric helps explain defects, and NT generally gives the best improvement to the base model. However, since some metrics have some overlapping information (i.e., moderate correlation), we want to study whether NT gives additional improvement in explaining defects to all other

state-of-the-art topic-based cohesion and coupling metrics and LOC.

We apply PCA analysis on all the metrics except NT (i.e., apply on LOC and other state-of-the-art topic-based cohesion and coupling, except CLCOM5), and use the resulting PCs as independent variables in the regression model. We use this model as our base model, and examine the improvement in explaining defects when NT is added to the model. PCA transforms the metrics into a set of uncorrelated PCs, so all the PCs are not correlated with each other. This solves the problem of multicollinearity and the order of the metrics does not matter in regression models anymore (Golberg and Cho, 2004; Kutner et al., 2004).

We report the D^2 improvement in explaining defects when NT is added to the base model in Table 3.23. We also report the regression coefficient and the level of statistical significance (p-value). We find that adding NT gives statistically significant improvement to the combined model (LOC and state-of-the-art topic-based cohesion and coupling metrics), and NT gives 2.9 – 97% improvement over the base model. The coefficient of NT is all positive, except in Eclipse 2.0. Positive coefficients imply that if a file has more topics, then it is more likely to be defect-prone. We do not report the coefficients for the system that NT is not statistically significant.

Since NT and MWE have a moderate correlation relationship, we perform the same analysis using MWE, and study how much improvement MWE gives. Table 3.24 shows the result of the study. The column *Base + MWE* has the same result has the column *Base + NT*, because these two columns report the D^2 of the full model. We include this column in the table for the ease of comparison.

We can see that the base models in Table 3.24 have higher D^2 values than that

Table 3.23: D^2 improvement when NT is added to the base model that is composed of PCs of LOC and other state-of-the-art topic-based cohesion and coupling metrics. "*" indicates a p-value < 0.05. "**" indicates a p-value < 0.01. "***" indicates a p-value < 0.001.

System	D^2 of Base	D^2 of Base+NT	% D^2 Inc.	NT Coeff.	p-Val
Mylyn 1.0	0.095	0.109	14.737	0.893	***
Mylyn 2.0	0.124	0.145	16.935	1.202	***
Mylyn 3.0	0.140	0.180	28.571	1.565	***
Firefox 1.0	0.163	0.181	11.043	0.995	***
Firefox 1.5	0.173	0.227	31.214	1.666	***
Firefox 2.0	0.170	0.192	12.941	1.014	***
Eclipse 2.0	0.137	0.141	2.920	-0.409	***
Eclipse 2.1	0.103	0.103	0.000	—	
Eclipse 3.0	0.118	0.118	0.000	—	
Netbeans 4.0	0.053	0.053	0.000	—	
Netbeans 5.0	0.067	0.092	37.313	1.232	***
Netbeans 5.5.1	0.043	0.085	97.674	1.762	***

in Table 3.23, which implies that including NT in the model explains more defects than including MWE in the model. In addition, MWE gives less improvement to the base models when compared to NT in Table 3.23. However, MWE gives a better improvement in Eclipse 2.1, 3.0, and NetBeans 4.0 (but these software systems also exclude many source code files that do not have methods). We find that although MWE and NT have a moderate correlation, NT still gives more improvements to the base model compared to MWE. Moreover, NT is much simpler to implement and works on files without methods.

Table 3.24: D^2 improvement when MWE is added to the base model that is composed of PCs of LOC, CCBO, NT, and RTC_s. "*" indicates a p-value $<$ 0.05. "**" indicates a p-value $<$ 0.01. "***" indicates a p-value $<$ 0.001. The column *Base+MWE* has the same number as the column *Base+NT* in Table 3.23, because they both refer to the same full model.

	Base	Base+MWE	% Inc.	MWE Coeff.	p-Val
Mylyn 1.0	0.106	0.109	2.830%	—	
Mylyn 2.0	0.138	0.145	5.072%	-1.084	**
Mylyn 3.0	0.180	0.180	0.00%	—	
Firefox 1.0	0.177	0.181	2.256%	-1.153	***
Firefox 1.5	0.226	0.227	0.442%	-0.654	*
Firefox 2.0	0.183	0.192	4.918%	-1.600	***
Eclipse 2.0	0.137	0.141	2.920%	-0.890	***
Eclipse 2.1	0.100	0.103	3.000%	-0.884	***
Eclipse 3.0	0.111	0.118	6.306%	-1.480	***
Netbeans 4.0	0.041	0.053	29.268%	-1.770	***
Netbeans 5.0	0.091	0.092	1.099%	—	
Netbeans 5.5.1	0.084	0.085	1.190%	0.727	*

NT gives improvements to LOC and all other state-of-the-art topic-based cohesion and coupling metrics. Although NT has a moderate correlation with MWE, NT gives more improvement to MWE. In addition, our previous finding (if a file has more topics then it is more likely to be defect prone) still holds when controlling for LOC and other cohesion and coupling measures.

3.4 Sensitivity Analysis for the Parameters of Our Approach

In our approach, we use several parameter values for LDA and the metrics that we define: two Dirichlet priors for smoothing (α and β), the number of iterations

(II), the number of topics (K), and δ in NT and NDT. We perform a parameter sensitivity analysis to see how these parameters affect the defect explanatory power of our topic-based metrics. We do not change the parameters of the topic-based cohesion and coupling metrics in 3.3.4, since we are interested in comparing how these metrics can help explain defects when controlling for the parameters.

In particular, we use our previous setting as a baseline (II=10,000, K=500, and δ=1%), and we change the value of each parameter to a lower and higher value (9,000 and 11,000 for II, 400 and 600 for K, and 0.5% and 2% for δ), and report the D^2 of NT and NDT. In this study, for α and β, we use the values as optimized by MALLET (McCallum, 2002).

We choose PCs until either 90% of variances are explained or the increase in variance explained by the next PC is less than the mean of the variance explained by all PCs (for Equation 3.4 and 3.7). We choose the parameter based on previous study (Jolliffe, 2002), and different numbers may give slightly different results. However, in our case studies, the 90% cut-off is almost never met, because we always first encounter a PC whose variance explained is smaller than the mean. Therefore, we keep this PCA cut-off the same throughout this section.

Table 3.25 – 3.26 shows the results of the sensitivity analysis. We list the baseline D^2 of the regression models, and report the new D^2 when the parameter is changed. We can see that D^2 is very stable across the parameters changes. In most cases, D^2 values remain unchanged, and in the worst case, D^2 only changes by 0.02 (10%). We find that the results of RQ3 are consistent when the parameters change, which implies that our results are not particular sensitive to the parameters that we choose in the thesis.

We also find that the result of our RQ1 and RQ2 still holds (most topics are not defect-prone, and defect-prone topics tend to be defect prone in future releases). The skewness and the correlation of topic defect densities across different release remain high. We manually check the topic label of the most defect-prone topics when K and II change, and we notice that the topic labels are still very similar to the labels in Table 3.4–3.7.

3.5 Threats to Validity

3.5.1 Parameter Sensitivity Analysis

Our approach involves the choice of several parameters, and there is no automated technique to find the optimal values for them. However, as shown in Section 3.4, our results are not sensitive to the parameters that we choose. Nevertheless, further research is required to understand the effects of these parameters on the results (e.g., change all the parameters at the same time).

3.5.2 Representing Concerns as Topics

Topic models are based on machine learning techniques, which involves some randomized algorithms. Therefore, each computation may result in slightly different topic distributions. We use a relatively large number of Gibbs sampling iterations (10,000) to approximate topics distributions, which hopefully gives us more stable results.

Table 3.25: Results of the parameter sensitivity analysis of the parameters. The baseline parameters and their values are shown in the table. For each system, we show the D^2 when the parameter changes. The values in the parentheses indicate the increase/decrease from the baseline D^2 score. * indicates the metric is statistically significant (p-value < 0.05).

Baseline Values: K=500, II=10,000, δ=0.01			
NT		*NDT*	
Lower	Higher	Lower	Higher
Mylyn 1.0 (Baseline D^2: LOC+NT = 0.14, PRE+CHURN+NDT = 0.24)			
K=400 0.14* (—)	K=600 0.14* (—)	K=400 0.23* (-0.01)	K=600 0.24* (—)
II=9K 0.14* (—)	II=11K 0.14* (—)	II=9K 0.24* (—)	II=11K 0.24* (—)
δ=0.005 0.14* (—)	δ=0.02 0.14* (—)	δ=0.005 0.24* (—)	δ=0.02 0.22* (-0.02)
Mylyn 2.0 (Baseline D^2: LOC+NT = 0.19, PRE+CHURN+NDT = 0.23)			
K=400 0.19* (—)	K=600 0.18* (-0.01)	K=400 0.23* (—)	K=600 0.23* (—)
II=9K 0.19* (—)	II=11K 0.19* (—)	II=9K 0.23* (—)	II=11K 0.23* (—)
δ=0.005 0.19* (—)	δ=0.02 0.19* (—)	δ=0.005 0.24* (+0.01)	δ=0.02 0.22* (-0.01)
Mylyn 3.0 (Baseline D^2: LOC+NT = 0.20, PRE+CHURN+NDT = 0.29)			
K=400 0.21* (+0.01)	K=600 0.20* (—)	K=400 0.29* (—)	K=600 0.29* (—)
II=9K 0.20* (—)	II=11K 0.20* (—)	II=9K 0.29 (—)	II=11K 0.29 (—)
δ=0.005 0.20* (—)	δ=0.02 0.20* (—)	δ=0.005 0.29* (—)	δ=0.02 0.29 (—)
Firefox 1.0 (Baseline D^2: LOC+NT = 0.16, PRE+CHURN+NDT = 0.18)			
K=400 0.16* (—)	K=600 0.16* (—)	K=400 0.18* (—)	K=600 0.18* (—)
II=9K 0.16* (—)	II=11K 0.16* (—)	II=9K 0.18* (—)	II=11K 0.18* (—)
δ=0.005 0.16* (—)	δ=0.02 0.15* (-0.01)	δ=0.005 0.18* (—)	δ=0.02 0.17* (-0.01)
Firefox 1.5 (Baseline D^2: LOC+NT = 0.21, PRE+CHURN+NDT = 0.25)			
K=400 0.21* (—)	K=600 0.21* (—)	K=400 0.25* (—)	K=600 0.25* (—)
II=9K 0.20* (-0.01)	II=11K 0.20* (-0.01)	II=9K 0.25* (—)	II=11K 0.25* (—)
δ=0.005 0.21* (—)	δ=0.02 0.20* (-0.01)	δ=0.005 0.26* (+0.01)	δ=0.02 0.24* (-0.01)
Firefox 2.0 (Baseline D^2: LOC+NT = 0.17, PRE+CHURN+NDT = 0.25)			
K=400 0.17* (—)	K=600 0.17* (—)	K=400 0.25* (—)	K=600 0.24* (-0.01)
II=9K 0.17* (—)	II=11K 0.17* (—)	II=9K 0.25* (—)	II=11K 0.25* (—)
δ=0.005 0.17* (—)	δ=0.02 0.17* (—)	δ=0.005 0.25* (—)	δ=0.02 0.24* (-0.01)

Continued in Table 3.26.

Table 3.26: Continued from Table 3.25. Results of the parameter sensitivity analysis of the parameters. The baseline parameters and their values are shown in the table. For each system, we show the D^2 when the parameter changes. The values in the parentheses indicate the increase/decrease from the baseline D^2 score. * indicates the metric is statistically significant (p-value < 0.05).

Baseline Values: K=500, II=10,000, δ=0.01			
NT		*NDT*	
Lower	Higher	Lower	Higher
Eclipse 2.0 (Baseline D^2: LOC+NT = 0.18, PRE+CHURN+NDT = 0.20)			
K=400 0.18 (—)	K=600 0.18* (—)	K=400 0.18* (-0.02)	K=600 0.18* (-0.02)
II=9K 0.18* (—)	II=11K 0.18* (—)	II=9K 0.20* (—)	II=11K 0.20* (—)
δ=0.005 0.18* (—)	δ=0.02 0.18* (—)	δ=0.005 0.19* (-0.01)	δ=0.02 0.21* (+0.01)
Eclipse 2.1 (Baseline D^2: LOC+NT = 0.11, PRE+CHURN+NDT = 0.15)			
K=400 0.11* (—)	K=600 0.11* (—)	K=400 0.15 (—)	K=600 0.15 (—)
II=9K 0.11* (—)	II=11K 0.11* (—)	II=9K 0.15 (—)	II=11K 0.15 (—)
δ=0.005 0.11* (—)	δ=0.02 0.11* (—)	δ=0.005 0.15* (—)	δ=0.02 0.15 (—)
Eclipse 3.0 (Baseline D^2: LOC+NT = 0.14, PRE+CHURN+NDT = 0.17)			
K=400 0.14 (—)	K=600 0.14* (—)	K=400 0.17* (—)	K=600 0.17 (—)
II=9K 0.14* (—)	II=11K 0.14 (—)	II=9K 0.17 (—)	II=11K 0.17 (—)
δ=0.005 0.14* (—)	δ=0.02 0.14 (—)	δ=0.005 0.17 (—)	δ=0.02 0.18* (+0.01)
NetBeans 4.0 (Baseline D^2: LOC+NT = 0.10, PRE+CHURN+NDT = 0.31)			
K=400 0.10* (—)	K=600 0.10* (—)	K=400 0.31* (—)	K=600 0.30* (-0.01)
II=9K 0.10* (—)	II=11K 0.10* (—)	II=9K 0.30* (-0.01)	II=11K 0.30* (-0.01)
δ=0.005 0.10* (—)	δ=0.02 0.09* (-0.01)	δ=0.005 0.30* (-0.01)	δ=0.02 0.30* (-0.01)
NetBeans 5.0 (Baseline D^2: LOC+NT = 0.10, PRE+CHURN+NDT = 0.17)			
K=400 0.11* (+0.01)	K=600 0.10* (—)	K=400 0.18* (+0.01)	K=600 0.17* (—)
II=9K 0.10* (—)	II=11K 0.10* (—)	II=9K 0.17* (—)	II=11K 0.17* (—)
δ=0.005 0.11* (+0.01)	δ=0.02 0.10* (—)	δ=0.005 0.18* (+0.01)	δ=0.02 0.17* (—)
NetBeans 5.5.1 (Baseline D^2: LOC+NT = 0.11, PRE+CHURN+NDT = 0.17)			
K=400 0.10* (-0.01)	K=600 0.11* (—)	K=400 0.17* (—)	K=600 0.18* (+0.01)
II=9K 0.11* (—)	II=11K 0.11* (—)	II=9K 0.18* (+0.01)	II=11K 0.18* (+0.01)
δ=0.005 0.11* (—)	δ=0.02 0.10* (-0.01)	δ=0.005 0.18* (+0.01)	δ=0.02 0.16* (-0.01)

3.5.3 Using Defects as an Indicator of Code Quality

In this chapter, we study software quality from the perspective of code quality. We use defects as an indicator of code quality. However, this indicator may be subjective, and other indicators of code quality such as performance may also be included in future research.

3.5.4 Subject Systems

We considered three versions of Mylyn, Firefox, Eclipse, and NetBeans, and answer our research questions based on these systems. However, the results that we found on these systems may not necessarily generalize to all software systems.

3.5.5 Results in Eclipse

We find that topic-based cohesion and coupling metrics give less improvement in Eclipse compared to other studied software systems. Moreover, although the coefficients of NT is small in Eclipse, they are negative, which contradicts with our finding in other studied software systems. A further study on Eclipse is needed to discover the reason these topic-based cohesion and coupling metrics are not working as well.

3.6 Guideline for Implementing Our Approach

In this section, we provide a step-by-step guideline to help practitioners implement our approach and use it on their software systems.

1. Obtain the source code of the software system.

2. Obtain the defect history (i.e., how many pre-release and post-release defects a file has) for the source code files of the older versions of the software system.

3. Preprocess the source code. Take only the identifier names and comments from the source code files, remove common English stopwords, and finally stem the words (Section 2.2).

4. Use the MALLET tool (McCallum (2002)) to apply LDA on the preprocessed source code (Section 3.2).

5. Compute the topic metrics (NT, NDT, TM, and DTM) proposed in Section 3.3.3. The defect information that is used by NDT and DTM is from the previous version of the software system.

6. Sort the files according to their NT and NDT values. Investigate the files with the highest values, and decide whether a refactoring of such files is necessary to increase file cohesion and decrease maintenance difficulty.

7. Train the regression model using TM and DTM as independent variables (other metrics such as LOC can also be added to the model), and the post-release defect information from the previous version. Apply the model on the current version of the software system to predict which files are more defect-prone. More testing resources may be allocated to these files.

3.7 Chapter Summary

In this chapter, we aim to understand the relationship between the *conceptual concerns* in source code files, i.e., their technical content, with their defect-proneness.

To do so, we approximated the concerns in each file with *statistical topics*, and proposed new metrics on these topics. In particular, we considered the defect history of each topic, which we hypothesized would help better explain the defect-proneness of the files.

To evaluate our new metrics, we performed a detailed case study on four large, real-world systems: Mylyn, Mozilla Firefox, Eclipse, and NetBeans. The highlights of our study results include:

- Some topics are much more defect-prone than others.
- A topic's defect-proneness holds over time.
- The more topics a file has, the higher the chances it has defects.
- The more *defect-prone* topics a file has, the higher are still the chances that it has defects.
- Our proposed topic-based metrics provide better defect explanatory power over existing static (i.e., LOC) and historical (i.e., PRE and churn) metrics, suggesting that our metrics provide additional information about the quality of the code. Further study should consider using such metrics alongside traditional metrics for building defect prediction models.
- Our NT metric, which measures the level of cohesion in a file, outperforms other topic-based cohesion and coupling metrics, and practitioners may benefit from including our metric when studying software cohesion and coupling using topic models.

Chapter 4

Using Topic Models to Study Code Testedness

In previous chapter, we have shown that it is possible to study software quality using topics from the perspective of code quality. We show that topics bring benefits to current static and historical metrics when explaining defects. In this chapter, we want to study software quality in another dimension using topics, namely code testedness.

Testing is one of the most practical approaches for detecting defects prior to the release of a software system. Therefore, through testing, the quality of a system can be improved. Previous research has proposed many approaches to help determine which *files* need more testing. However, testers typically create test cases based on the *features* (or *conceptual concerns*) of the system. In this chapter, we examine the relationship between the conceptual concerns in source code and those in test files. Same as the previous chapter, we use a statistical topic modeling approach (LDA) to approximate the conceptual concerns from the source code files as *topics*.

We define testedness by how much a piece of source code is tested in test files. We measure how *well-tested* a topic is, and how *defect-prone* it is. Our overarching hypothesis is that when a topic is highly tested, it will be less defect-prone. Our study helps to understand at a high level which concerns need more testing by identifying topics that likely contain defects, but are not well-tested. By testing the topics discovered by our approach, additional defects can possibly be located. In addition, because topics can be linked back to source code files (see Section 2.1.1), managers and software developers can allocate more testing resources on the files related to these topics, and thus improve code quality and reduce maintenance costs. To evaluate our approach, we perform an in-depth case study on three large, real-world software systems, focusing on the following research questions.

RQ1: How shared are the topics between source code files and test files?

To answer this question, we measure the proportion of a topic found in test code and source code. We find that in all three systems under study, between 48% and 78% of topics are shared between source code and test files.

RQ2: Are more tested topics less defect-prone?

We find four classes of topics: (i) those that are highly tested and have low defect-density (25% of the topics); (ii) those that are less tested and have high defect density (21%); (iii) those that are less tested and have low defect-density (54%); and (iv) those that are better tested and have higher defect-density (4%). We find that well-tested topics are likely to be less defect-prone. We also describe the relationships between topic testedness and defect density as *non-coexistence* (well-tested topics are less defect-prone).

RQ3: Can we automatically identify defect-prone topics that are not well-tested?

To answer this question, we use a naive Bayes classifier to do cross-release prediction, and identify defect-prone topics that are not well-tested. We find that by training on previous releases of a system, we can obtain, on average, a precision of 0.77 and a recall of 0.75 when classifying topics according to their testedness and defect density in later releases. By linking topics to source code files, practitioners can allocate additional testing resources to these parts of the code more effectively.

Chapter Overview

Section 4.1 talks about the subject systems that we use to answer the research questions, and Section 4.2 describes the design of our case studies. Section 4.3 shows the results of our case studies, and Section 4.4 shows the results of our parameter sensitivity analysis. In Section 4.5, we discuss potential threats to validity, and finally conclude the chapter with a summary in Section 4.7.

4.1 Choice of Subject Systems

We use the same three subject systems as before: Mylyn, Eclipse, and NetBeans (Table 4.1). Firefox is not included in this chapter because Firefox is implemented in C/C++ and do not have heuristically identifiable unit test files. The numbers are slightly different from Table 3.1, because we also check out the test files in the repositories, and separate source code and test files according to some heuristics

Table 4.1: Statistics of the subject systems, after preprocessing.

	Total lines of source code (K)	No. of source files	No. of test files	Total lines of test code (K)	Post-release defects
Mylyn 1.0	126	830	165	20	712
Mylyn 2.0	135	921	173	21	1,012
Mylyn 3.0	165	1,115	217	29	480
Eclipse 2.0	797	6,722	1,011	237	1,692
Eclipse 2.1	987	7,845	1,290	430	1,182
Eclipse 3.0	1,305	10,545	1,835	600	2,679
NetBeans 4.0	840	3,874	502	95	287
NetBeans 5.0	1,758	7,880	1,246	234	194
NetBeans 5.5.1	2,913	14,522	2,238	438	739

(more about the heuristics in Section 4.2.1). We choose these three systems because they all have unit tests for testing. Unit test files are usually large enough to contain enough linguistic data (i.e., identifier names and comments) from which we can extract conceptual concerns. If the test files were too small (for example, simple one-line scripts), then our methodology may not have enough information to capture meaningful topics (Titov and McDonald, 2008). In addition, these three systems differ in size and number of defects.

4.2 Case Study Design

In this section, we describe our analysis process, depicted in Figure 4.1. Again, we use MALLET (McCallum, 2002) as our LDA implementation, and we run MALLET with the same setting as in the previous chapters (10,000 sampling iterations, 500 topics). We also build the topics using both unigrams (single words) and bigrams

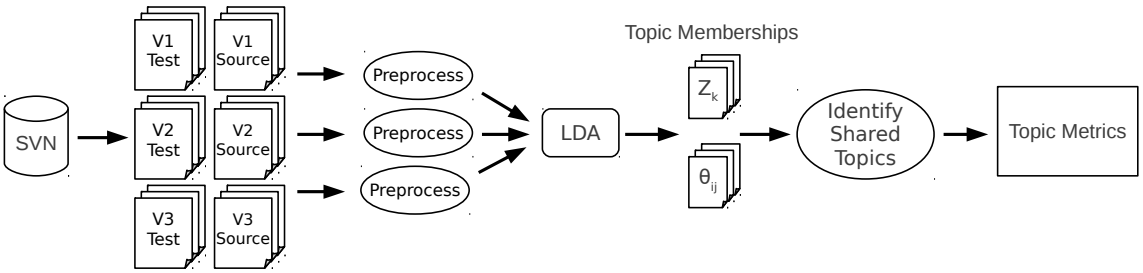


Figure 4.1: Our process of applying topic models and calculating topic-based metrics. We preprocess test file and source code files and run LDA on all versions of the data together. We first obtain shared topics using the topic and topic membership values returned by LDA. We then calculate the topic-based metrics on *shared topics* only.

(pairs of adjacent words). We apply LDA to all versions of the preprocessed source code and test files of a system at the same time.

In this section we describe how we identify test files, and describe Maximal Information Coefficient (MIC), an approach we use to uncover the relationship between topic testedness and defect-proneness.

4.2.1 Identifying Test Files

Unfortunately, it is difficult to automatically identify which source code files are in fact test files, as test files are not explicitly marked in any way. To do so, we use a heuristic similar to that of Zaidman et al. (Zaidman et al., 2008). First we check out the entire source code repository. We extract all source code files that have a file path which contains the test keywords *junit* or *test*, since the subject systems use the JUnit testing framework. For example, we would extract the source code files under the folder *src/test/*, but not *src/UI/*. However, it is possible that a file may not be related to testing even though its path contains one of these test keywords. For example, in Eclipse 2.0, the source code file *.../core/tests/harness/PerformanceTimer.java*

is a utility class that is used for helping other test cases, but itself is not used for testing. To increase the confidence of identifying test files, we make sure the name of the source code files also contains at least one test keyword. Therefore, a file is identified as a test file if and only if both its name and path contains at least one test keyword. Table 4.1 shows the number of test files that are identified using this heuristic.

4.2.2 Maximal Information Coefficient

Maximal information coefficient (MIC) is an approach that was recently developed to discover different kinds of relationships, such as linear and functional, between pairs of variables (Reshef et al., 2011). Recently, MIC has been shown to be useful in finding relationships in software engineering data (Posnett et al., 2012). Since the relationship between topic testedness and defects may not be linear, we use MIC to find any possible non-linear relationships in the data that cannot be discovered by the commonly used Pearson or Spearman correlation coefficients.

MIC provides *generality*, i.e. finding many different kinds of relationships, and *equitability*, i.e. gives a similar score to different relationships with the same amount of noise. In this way, MIC can be viewed as the coefficient of determination (R^2) in linear regression analysis, except MIC is not limited to a simple linear relationship. We use the following maximal information-based nonparametric exploration (MINE) measures that are computed from MIC using the tool Reshef et al. provide (Reshef et al., 2012):

- *MIC - p^2*: a measure of the linearity of the relationship, where p is the Pearson correlation coefficient. Since MIC value is close to p^2 when the relationship is

linear, if MIC - p^2 is close to zero, then the relationship is linear; if MIC - p^2 is close to MIC, then the relationship is non-linear.

- *MAS*: a measure of the departure from monotonicity, i.e., a function that preserves a given order. MAS is always $\leq$ MIC. If MAS is very close to MIC, then the relationship is not monotonic; otherwise, the relationship is monotonic.

- *MEV*: a measure of the non-functionality, i.e., whether the relationship can pass a vertical line test (Stewart, 2009). If the relationship is functional between two variables, then when one variable changes, the other variable will also change according to some mathematical functions. MEV is also always $\leq$ MIC, and when MEV is close to zero, the relationship is non-functional; otherwise, the relationship is functional.

4.3 Results of Case Studies

4.3.1 How shared are the topics between source code and test files?

In this question, we investigate how topics are shared between source code and test files. To do so, we check whether topics are evenly distributed between these two types of files. If topics are only present in either one of the file types (source code or test), then we cannot examine the effect of testing on code quality using topics, since there is no overlap between the topics in them. By determining which topics are shared, we will also determine which topics are not shared: those that are not found in test files (*source-only* topics), and those that are not found in source code

files (*test-only* topics). Excluding not-shared topics will allow us to remove possible outlier topics (i.e., topics about `assert` will be high tested but low defect prone, because they are keywords in test files; however these topics are not present in source code files) in our further analysis in RQ2 and RQ3.

Approach

We calculate the test ***weight*** of a topic z_i, which measures the total lines of testing code in the topic as

$$\text{W}_{test}(z_i) = 1/LOC_{test} \sum_{j=1}^{n} \theta_{ij} * \text{LOC}(f_j), \tag{4.1}$$

where *LOC(f_j)* is the lines of code of file f_j, and LOC_{test} is the total LOC of all test files.

Similarly, we define source W_{source}, the weight metric for source code files, as

$$\text{W}_{source}(z_i) = 1/LOC_{source} \sum_{j=1}^{n} \theta_{ij} * \text{LOC}(f_j), \tag{4.2}$$

where LOC_{source} is the total LOC of all source code files.

We normalize the weight metrics above by LOC because LOC is different between test code and source code files. This normalization helps eliminate possible influences by the size difference of the two file types. To find shared topics between source code and test files, we define the ***W ratio*** metric of topic z_i as

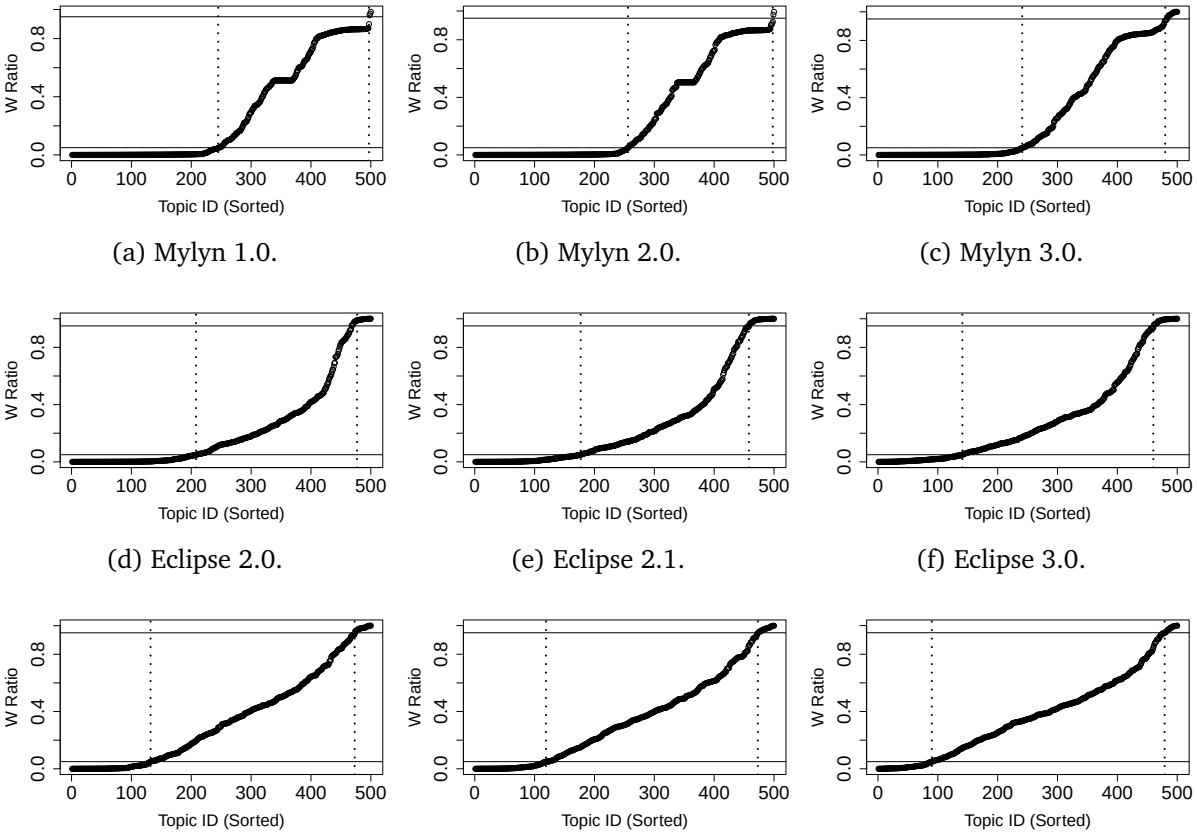
$$\text{W ratio}(z_i) = \frac{W_{test}(z_i)}{W_{test}(z_i) + W_{source}(z_i)}. \tag{4.3}$$

Table 4.2: Summary of topics that belong to source code, test, and shared topics. Percentage of shared topics is calculated as the number of shared topics over the total number of topics (500 topics).

	No. of source-only topics	No. of test-only topics	No. of shared topics	% shared topics
Mylyn 1.0	245	3	252	50%
Mylyn 2.0	256	2	242	48%
Mylyn 3.0	241	17	239	48%
Eclipse 2.0	208	33	269	54%
Eclipse 2.1	177	42	281	56%
Eclipse 3.0	141	40	319	64%
NetBeans 4.0	132	27	341	68%
NetBeans 5.0	119	27	354	71%
NetBeans 5.5.1	90	21	389	78%

The W ratio metric allows us to determine the proportion of a topic's weight found in test files and, conversely, source code files.

There are two possible methods to determine source-only topics (e.g., topics about mutators and accessors, or printing) and test-only topics (e.g., topics about `assert`): (i) by manual inspection of the topics; or (ii) by removing topics according to some predefined thresholds. To avoid any potential bias that could stem from manual inspection, we use W ratio values of 0.05 and 0.95 as thresholds to remove topics that are more prevalent in source code or test files (see Section 4.4 and 4.5 for a discussion of threshold values and excluding topics). Topics that have a W ratio less than 0.05 are more prevalent in source code files; topics with a weighted ratio larger than 0.95 are more prevalent in test files. The topics that have a W ratio between 0.05 and 0.95 are the *shared topics* that we seek.



(a) Mylyn 1.0. (b) Mylyn 2.0. (c) Mylyn 3.0.

(d) Eclipse 2.0. (e) Eclipse 2.1. (f) Eclipse 3.0.

(g) NetBeans 4.0. (h) NetBeans 5.0. (i) NetBeans 5.5.1.

Figure 4.2: The W ratio of test topics in source code files. The dashed lines indicate the cut-off thresholds of 0.05 and 0.95. There are three different categories of topics shown in the figures: source-only topics (left), test-only topics (right), and shared topics (middle).

Results

Figure 4.2 shows the weighted topic ratio of each version of each subject system. The figure illustrates that topics belong to three different categories: source-only topics (left), test-only topics (right), and shared topics (middle). In all the subject systems of our case study, we find that 48–78% of the topics are being shared between source code files and test files, using our chosen threshold value (Table 4.2). Since a majority of topics are shared, we can study code coverage using topic models.

Discussion

To understand which topics are identified as *not-shared topics* (source-only or test-only topics) and what these topics represent, we look at the top words of some representative test and source code topics (Table 4.3 – 4.4).

Mylyn Since Mylyn is a task management system, it requires many GUI and control related features. Thus, the source-only topics in Mylyn are related to sorting, adding, removing, and monitoring tasks or operations (topics 20, 32, 55). Topic 41 is related to synchronizing tasks with the user interface, and topics 190 and 282 are related to accessing source code repositories such as Bugzilla.

Topics 171 and 331 are related to test keywords in Mylyn, which do not appear often in source code files. We look at test files related to topic 198, and find that they are about launching JUnit plugin tests; for example, launching Mylyn Plug-in Development Environment UI configuration tests. Other test topics are about testing hypertext structure bridging (topic 496), testing shared task directory (topic

Table 4.3: Topic label and top words of selected test/source-only topics in our subject systems. The number on the left-hand side of each column represents the topic number.

Test-Only Topics		Source-Only Topics	
Label	Top Words	Label	Top Words
Mylyn 1.0			
171 eclips debug	configur, launch, test, eclips, junit, launch_configur	20 sort	sort, order, sort_order, action, view, categori
198 program arg	arg, program, configur, program_arg, plugin, add	55 progress monitor	work, progress, progress_monitor total_work, total, tick
Mylyn 2.0			
331 assert enable	enabl, assert_enabl, test_assert, accompani, enabl_test, distribut	41 set synchronize	synchron, task_task, update, manag, data_manag, set
496 structur bridge	web, bridg, recourc, structur_bridg, structur, web_resourc	190 select index	version, repositori, combo, valid, server, repositori_version
Mylyn 3.0			
60 sandbox share data	share, share_data, bob, data, sandbox, folder	32 add task	add, add_task, command, servic, id, task_viewer
444 assert wizard	wizard, histori, assert, size, histori_context, page	282 mylyn task core	core, repositori, repositori_repositori, connector, throw, repositori_attach

Test-Only Topics		Source-Only Topics	
Label	Top Words	Label	Top Words
Eclipse 2.0			
101 return qualify	code, test, qualifi, creat, creat_test, code_creat	8 search scope	scope, search, pattern, search_scope, limit, privat
291 assert target exit	file, project, exist, assert, workspac, exit_assert	48 submission handler	servic, submiss, shell, command, bind, activ
Eclipse 2.1			
59 nl	unit, nl, nl_nl, source, type, assert	277 content offset	content, offset, local, attribut, local_content, content_offset
445 assert equal	assert, equal, test, assert_equal, public_test, length	243 gdk color	gdk, gtk, window, gdk_color, style, set
Eclipse 3.0			
424 test suit	test, suit,test_suit, add, test_test, add_test	12 print	print, packet, id, command, stream, println
468 item assert equal	select,test,equal, assert, assert_equal, number	124 handl gtk	gtk, handl, widget, handl_gtk, gtk_widget, signal

Continued in Table 4.4.

Table 4.4: Continued from Table 4.3. Topic label and top words of selected test/source-only topics in our subject systems. The number on the left-hand side of each column represents the topic number.

Test-Only Topics			Source-Only Topics		
	Label	Top Words		Label	Top Words
NetBeans 4.0					
0	test suit	test, suit, test_suit, junit, nb, nb_test	18	content pane	pane, set, layout, add, pane_add, border
295	property test	test, exclud, testbag, config, includ, set	470	mutator method	method, pattern, properti, setter, set, getter
NetBeans 5.0					
38	cvs test	cvsroot, test, set, cv, cvss, crso	33	text area	area, text, text_area, jtext_area, area_set, set
390	test editor	test, action, editor, node, test_editor, netbean_test	182	paint component	paint, compon, paint_compon, draw, item, substitut
NetBeans 5.5.1					
10	jframe test	test, frame, assert, jframe, set, visibl	106	slide bar	slide, tab, bar, bound, slide_bar, compon
79	perform test	test, perform, perform_test, pass, method, pass_test	125	mdb	password, mdb, factori, connect_factori, connect, set

60), and testing task import wizard (topic 444).

Eclipse Source code topics in Eclipse are related to searching (topic 8) and GUI (topics 124 and 243). Topics that are about converting packets to human readable form (topic 12), key binding service (topic 48), and servlet to interface client (topic 277) are more often found in source code files. On the other hand, test topics in Eclipse all contain some test keyword that do not occur in source code files.

NetBeans NetBeans has source code topics related to GUIs (topics 18, 33, 182, and 106), mutator and accessor (topic 470), and message driven beans initialization (topic 125). Test topics in NetBeans are also related to different kinds of software components testing, such as editor (topic 390) and CVS (topic 38).

In general, we would like to see only keywords that are specific in either source-only or test-only topics, such as `assert`, in the not-shared topics. However, based on the results above, our source-only topics also contain some topics, which naturally should have appeared in test files, and are excluded. We discuss about possible threats to the validity of such exclusions of the not-shared topics in Section 4.5.

About 48%–78% of the topics are shared between source code and test files, which implies that we can study topic testedness by comparing the prevalence of a topic in these two types of files. In addition, we find that the not-shared topics are mostly about keyword topics that only exist in source code or test files.

4.3.2 Are more tested topics less defect-prone?

In this question, we want to understand the relationships between how tested a topic is and its defect density. We hypothesize that topics that are more tested will be less defect prone, and topics that are less tested will be more defect prone. If the hypothesis holds, then practitioners can focus on testing source code files related defect-prone topics to improve code quality.

Approach

Researchers have proposed different metrics to capture the code coverage of test files. In this chapter, we want to study code coverage at the abstraction level of topics, and examine the effect of *topic testedness* (i.e., how well is a topic tested) on the post-release defect density of each topic. We define the ***testedness*** of topic z_i as

$$\text{Testedness}(z_i) = \frac{\text{W}_{test}(z_i)}{\text{W}_{source}(z_i)} \tag{4.4}$$

where $W_{test}(z_i)$ and $W_{source}(z_i)$ are the computed weight metrics (Equation 4.1 and 4.2) for test and source code files, respectively. We normalize the weight metrics according to the total lines of code in test files and source code files to take the size difference between both file types into the account. The intuition behind this metric is that if a topic is more prevalent in the test files, then the topic is better tested. We use the Equation 3.2 to quantify the average *post-release defect* density in each topic.

We draw scatter plots of topic post-release defect density against topic testedness to visually see the relationships between them (Figure 4.3). Note that we also exclude all the not-shared topics as identified in RQ1.

We classify topics into four different classes as shown in Figure 4.3:

- Class LTHD: low testedness and high defect density
- Class LTLD: low testedness and low defect density
- Class HTLD: high testedness and low defect density
- Class HTHD: high testedness and high defect density

We classify topics that have a testedness and defect density value smaller than the third quartile of all topics to be class LTLD. Topics that have a defect density larger than or equal to the third quartile and have a testedness value smaller than the third quartile are classified as class LTHD. Class HTLD includes the points that are highly tested (larger than the third quartile) but have lower defect-proneness (smaller than the third quartile). Finally, there is a few topics that are classified as high tested and high defect prone by our approach (HTHD).

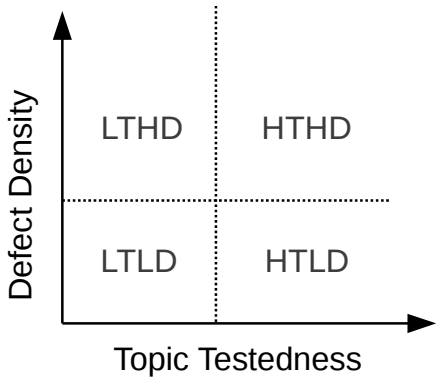


Figure 4.3: Position of each class on the scatter plot. The bottom left corner has low tested and low defect prone (LTLD) topics. The upper left corner has low tested and high defect prone topics. The rest of topics are high tested and low defect prone (HTLD). There is only a few topics that are high tested and high defect prone (HTHD).

We use the MIC score and MINE measures (see Section 4.2.2) to describe and verify the relationships that we observe in the scatter plots.

Results

Table 4.5 summarizes the number of topics in each class. Class LTHD has the least number of topics among all classes, except HTHD, and class LTLD has significantly more topics than other classes. This implies that there exist less defect-prone topics that are not well-tested, and it would be desirable to focus testing on these topics. Figure 4.4–4.6 shows the scatter plots of the relationship between topic testedness and defect density for each version of each system. Due to the approach we use to automatically classify the topics, topics in HTHD are usually at the boundary between LTHD and HTLD, and they do not necessarily have very high testedness nor defect density. The topics in other classes are distributed along the X and Y axes.

Table 4.5: Number of topics in each class.

	LTHD	LTLD	HTLD	HTHD
Mylyn 1.0	55	134	55	8
Mylyn 2.0	56	125	56	5
Mylyn 3.0	56	125	56	5
Eclipse 2.0	52	142	52	13
Eclipse 2.1	60	150	60	11
Eclipse 3.0	72	167	72	8
NetBeans 4.0	72	183	73	13
NetBeans 5.0	60	205	60	29
NetBeans 5.5.1	72	219	73	25

The relationship between topic testedness and defect-proneness. Table 4.6 shows the MINE and MIC scores, indicating that there is a *non-coexistence* relationship between testedness and post-release defect density (Reshef et al., 2011). In our case, this relationship indicates that when a topic is well-tested, then it is less defect prone.

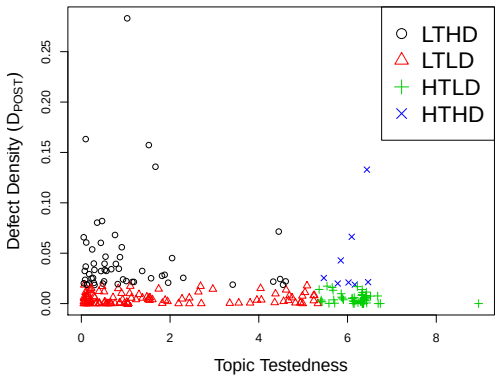
A non-coexistence relationship happens when one variable is more dominant, the other variable is less dominant (cannot both have high values). Topics in class HTLD have a relatively low defect density compared to the other two classes. In addition, topics in class LTHD always have the highest defect density amongst all topics, and their testedness are also low. Although there are some outliers (HTHD), where a few relatively well-tested topics have a higher defect density value, most topics follow the non-coexistence pattern: *when a topic is defect-prone, it is usually not well-tested; when a topic is well-tested, its defect-density is usually low.* Topics in class LTLD are usually related to configuration tasks or trivial operations, which are not usually tested. However, topics in LTLD also have low defect density, so we are

not interested in testing these topics.

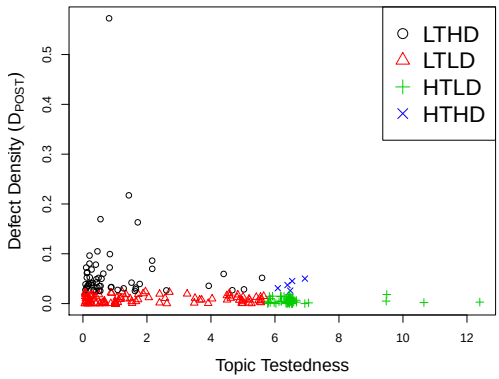
In order to support our argument that the discovered relationships are non-coexistence and not random, we check the linearity, monotonicity, and functionality of the relationship between two variables (Table 4.6). In all subject systems, the p values are all negative, and the differences between MIC and MIC$-p^2$ are very small. This indicates that the relationship between topic testedness and post-release defect density is non-linear (Reshef et al., 2011). For example, a not well-tested topic can be either in LTHD or LTLD, since testedness and defect density are not directly proportional to each other. In addition, the relatively small MAS values imply that the relationship is monotonic, which provides more evidence the relationship is non-coexistence; i.e., when one variable increases, the other variable will also increase/decrease accordingly. For example, when testedness increases, topic defect density decreases. The values for MEV are all very close to MIC, which means there is a functional relationship. For example, post-release defect density can be computed as some function of testedness. Although the relationship is non-linear, negative p values indicate that when the value of testedness increases, post-release defect density decreases (negative correlation). Taken in aggregate, the results in Table 4.6 support our hypothesis that there is a non-coexistence relationship between topic testedness and topic post-release defect density (well-tested topics are usually less defect prone), and this relationship is not random nor non-functional.

Discussion

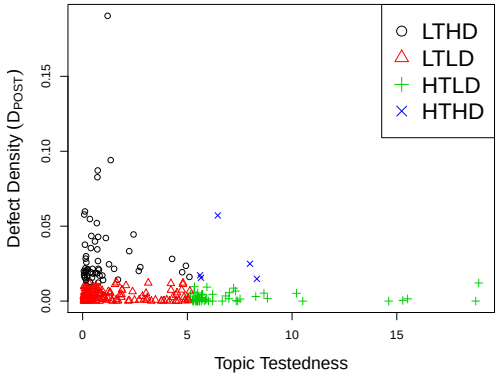
We list one concrete example of a topic in each class from the subject systems we used in our case study in Table 4.7. (We note that there are many other examples in



(a) Mylyn 1.0.

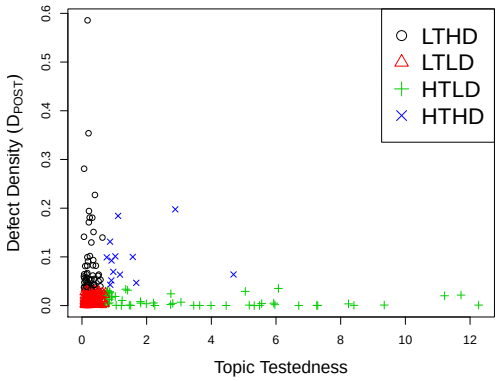


(b) Mylyn 2.0.

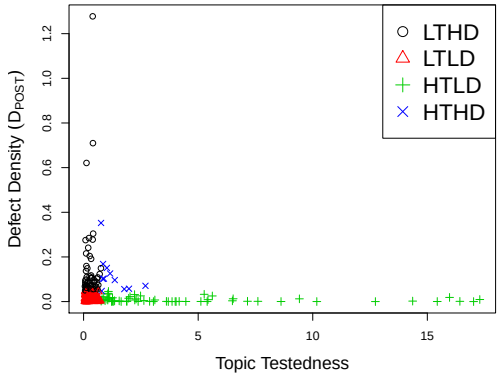


(c) Mylyn 3.0.

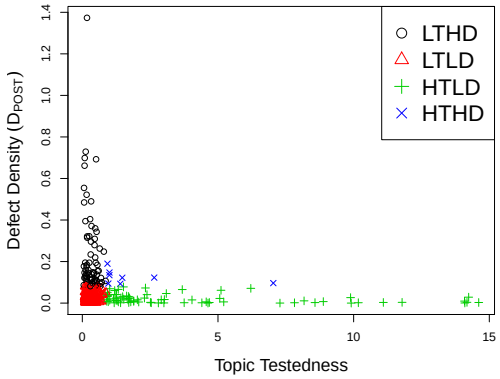
Figure 4.4: Scatter plots of topic post-release defect density against topic testedness for all versions of Mylyn. Each point represents a topic. Black points (class LTHD) are low-tested and defect-prone topics; red points (class LTLD) are low-tested and less defect-prone topics; green points (class HTLD) are highly-tested and less defect-prone topics; and blue points (class HTHD) are highly-tested and high defect-prone topics. Continued in Figure 4.5



(a) Eclipse 2.0.

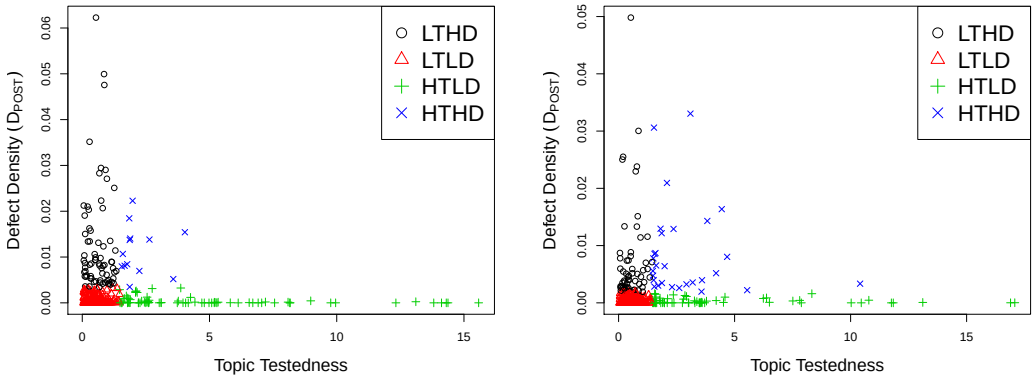


(b) Eclipse 2.1.



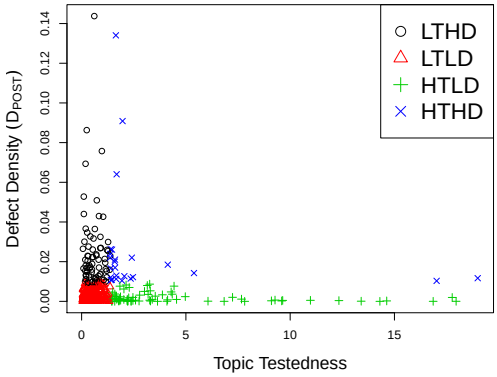
(c) Eclipse 3.0.

Figure 4.5: Scatter plots of topic post-release defect density against topic testedness for all versions of Eclipse. Each point represents a topic. Black points (class LTHD) are low-tested and defect-prone topics; red points (class LTLD) are low-tested and less defect-prone topics; green points (class HTLD) are highly-tested and less defect-prone topics; and blue points (class HTHD) are highly-tested and high defect-prone topics. Continued in Figure 4.6.



(a) NetBeans 4.0.

(b) NetBeans 5.0.



(c) NetBeans 5.5.1.

Figure 4.6: Scatter plots of topic post-release defect density against topic testedness for all versions of NetBeans. Each point represents a topic. Black points (class LTHD) are low-tested and defect-prone topics; red points (class LTLD) are low-tested and less defect-prone topics; green points (class HTLD) are highly-tested and less defect-prone topics; and blue points (class HTHD) are highly-tested and high defect-prone topics.

Table 4.6: Scores of MINE metrics computed between topic testedness and topic post-release defect density.

	MIC	MIC-p^2	MAS	MEV	p
Mylyn 1.0	0.38	0.36	0.12	0.38	-0.12
Mylyn 2.0	0.40	0.37	0.10	0.39	-0.18
Mylyn 3.0	0.26	0.22	0.04	0.25	-0.20
Eclipse 2.0	0.20	0.18	0.02	0.19	-0.12
Eclipse 2.1	0.21	0.20	0.02	0.21	-0.11
Eclipse 3.0	0.24	0.21	0.08	0.24	-0.16
NetBeans 4.0	0.24	0.22	0.07	0.24	-0.13
NetBeans 5.0	0.20	0.20	0.02	0.20	-0.02
NetBeans 5.5.1	0.19	0.19	0.03	0.19	-0.08

Table 4.7: Top words, testedness, and defect-density of selected topics from each class of topics.

	Top words	Testedness	Median of Testedness	Defect Density	Median of Defect Density
class LTHD: Mylyn 1.0					
417	task, mylar, eclips, eclips_mylar, mylar_task	1.03	1.116	0.283	0.004
class LTLD: Eclipse 2.1					
494	sheet, cheat, cheat_sheet, resourc, properti_sheet	0.22	0.343	$<$0.001	0.008
class HTLD: NetBeans 5.5.1					
206	frame, jintern, jintern_frame, intern, intern_frame, pane	2.40	0.644	$<$0.001	0.002
class HTHD: Eclipse 3.0					
206	breakpoint, debug, ijava, thread, core, suspend	2.65	0.387	0.12	0.002

each case study system for each class.) The topic on task-related actions and Eclipse integration is more defect-prone in Mylyn 1.0. As we can see from Table 4.7, the testedness of this topic is also very low, and it is classified as class LTHD. This topic corresponds to tasks management in Mylyn.

Conversely, the stack map frame topic in NetBeans 5.5.1 is a topic that is highly tested and has a very low defect density. It is therefore classified as class HTLD. This topic corresponds to one of the basic operations in NetBeans for controlling the stack, on which many higher-level functions rely. The topic on property sheets is used for displaying program properties to developers in Eclipse IDE. This is not a core functionality in Eclipse. Hence it neither requires much testing nor does it have many defects. Debugging related functionality in Eclipse are tested more, but it also has a relatively high defect density.

We define topic testedness and defect density, and classify topics into four classes according to these two metrics. We find when a topic is well-tested, then it is usually less defect prone; and when a topic is defect prone, then it is usually well-tested. We verify that this relationship exists and is not random using the MIC score and MINE measures.

4.3.3 Can we identify defect-prone topics that need more tests in future releases?

In RQ2, we see evidence that highly tested topics are usually less defect-prone. Therefore, it will be beneficial to know which topics require more testing in future releases, since we can link topics to source code files or identify the defect-prone concerns using the top words in the topics (Section 2.1.1). In this question, we

want to predict the defect-prone topics that may require more testing. Topics that are less tested can be classified into two categories: high defect-prone and less defect-prone. If we could automatically identify those topics that are less tested and more defect-prone, then practitioners could put more testing resources on the source code files or concerns related to these topics, reducing time and cost in the testing phase before releases. Even though topics in class LTLD are low tested, we are not interested in them, since they have low defect-density. By avoiding further testing on topics in class LTLD, we can avoid the unnecessary allocation of testing resources.

Approach

We build a classifier to predict the class (i.e., class LTHD, LTLD, or HTLD) to which a topic belongs. We exclude the topics in HTHD in our analysis, because the main focus of this research question is to predict the topics in LTHD. In addition, HTHD has much smaller number of topics, which may affect the overall quality of the classifier (Provost, 2000). We use the following topic-based metrics as the independent variables (i.e., features) in the classifier: topic weight, support, scatter, and churn density. These topic-based metrics are defined below.

The ***weight*** of a topic measures the total lines of code in the topic as

$$\mathrm{W}(z_i) = \sum_{j=1}^{n} \theta_{ij} * \mathrm{LOC}(f_j), \tag{4.5}$$

where *LOC(f_j)* is the lines of code of file f_j. Since size is a well-known predictor for software defects, we include this metric in the classifier.

The ***support*** of a topic measures how many files contain the topic, and is defined

as

$$\text{Support}(z_i) = \sum_{j=1}^{n} I(\theta_{ij} \geq \delta), \tag{4.6}$$

where I is the indicator function that returns 1 if its argument is true, and 0 otherwise. We set the membership threshold δ in Equation 4.6 to 1% to remove insignificant topics in files when computing the support metric for each topic. A more detailed analysis on this threshold value is presented in Section 4.4.

The ***scatter*** of a topic measures how spread out the topic is across all source code files, based on the information entropy of the topic memberships values. We define the scatter of a topic z_i as

$$\text{Scatter}(z_i) = -\sum_{j=1}^{n} \log(\theta_{ij}) * \theta_{ij}, \tag{4.7}$$

Scatter is a measure of the level of coupling of a topic. If a topic is highly scattered, then the topic is implemented across many different source code files, increasing maintenance difficulty.

Topic churn density measures the number of changes per lines of code that is made to the part of files related to the topic. We define topic churn density as

$$D_{\text{CHURN}}(z_i) = \sum_{j=1}^{n} \theta_{ij} * \left(\frac{\text{CHURN}(f_j)}{\text{LOC}(f_j)} \right), \tag{4.8}$$

where *CHURN(f_j)* is the total number of prior changes to source code file f_j before release. Previous research has shown that if the code in a topic is changed more often, then this topic is more likely to be defect-prone (Nagappan and Ball, 2005). The above-mentioned metrics measure the structure and history of a topic, which

may affect the topic testing and maintenance practice, and its defect-density.

Previous studies have shown that a naive Bayes classifier is an effective algorithm for defect prediction (Menzies et al., 2007; Turhan and Bener, 2009), thus we use naive Bayes in this chapter. We train the classifier on older releases, and predict on newer releases. For example, we train our classifier using Eclipse 2.0, and predict the class in which the topics in Eclipse 2.1 belong. To avoid the problem of having highly correlated independent variables in the classifier, we use principal component analysis (PCA) to transform these variables into a new set of uncorrelated variables (Turhan and Bener, 2009). We select the principal components (PCs) until either 90% of the variances are explained, or when the increase in variance explained by adding a new PC is less than the mean variance explained of all PCs (Jolliffe, 2002).

Since we are interested in finding the defect-prone topics that require more testing, we only report the precision and recall for classifying topics in this class (i.e., class LTHD). Our goal is not predicting defects, but rather, helping practitioners allocate testing resources more effectively.

Results

We show the classification results in Table 4.8. In all the studied system, we obtain, on average, a precision of 0.77 and a recall of 0.75, which means that most of the low-tested and defect-prone topics are classified correctly. Mylyn has the best classification result among three systems, possibly because Mylyn has a larger proportion of test files. In Table 4.1, we can see that although Mylyn is the smallest system among the three, it has relatively more test files. NetBeans, on the other

Table 4.8: Precision, recall, and F-measure for classifying low-testedness and defect-prone topics (class LTHD).

Train on	Test on	Precision	Recall	F-measure
Mylyn 1.0	Mylyn 2.0	0.88	0.80	0.84
Mylyn 2.0	Mylyn 3.0	0.82	0.71	0.76
Eclipse 2.0	Eclipse 2.1	0.79	0.63	0.70
Eclipse 2.1	Eclipse 3.0	0.84	0.79	0.81
NetBeans 4.0	NetBeans 5.0	0.65	0.60	0.62
netbeans 5.0	NetBeans 5.5.1	0.64	0.86	0.73

hand, has the least proportion of test files, which also yields the lowest F-measure. Therefore, if a system has more test files, we can classify the topics more accurately. Since we obtain high precision and recall values for all subject systems, we conclude that practitioners can use our approach to reliably identify those defect-prone topics that require more testing and allocate more testing resources on these specific topics, improving the overall code quality.

Discussion

Testers usually write test cases to test features (or concerns) in software systems (Weyuker, 1998), and topics can be viewed as an approximation of features (Baldi et al., 2008). The results above show that we can predict which topics are low tested and defect prone, which has the potential to help testers allocate testing resources. Further, to show that our approach can be used to complement current prediction-based resource allocation methods for finding defects, we perform an additional experiment that compares our approach with existing approaches.

Namely, we use our prediction model to identify LTHD topics, resulting in a list of possibly-defect-prone topics. We link these topics to their corresponding source

code files using a topic membership value of 0.5, so if a source code file has a membership value larger than 0.5 in any of the LTHD topics, then this file belongs to LTHD. We choose the relatively high value of 0.5 because it helps us find the source code files that truly belong to the topic (a file can only belong to one topic which has a membership value $>$ 0.5), and helps to limit the number of linked files that must be examined by testers.

To compare our approach with prediction-based resource allocation approaches, we build a linear regression model (LM) as a baseline for comparison. We use code churn and LOC as the independent variables in the model and predict the number of defects (Bird et al., 2011). We select the top n files predicted by our approach and the top n files selected by LM, and examine their similarities and differences. (n is determined by the number of files that belong to LTHD, which is different for each system.)

Table 4.9 shows the median LOC, defect density, and percentage of overlap between the source code files predicted by our approach and the LM model. In all studied systems, our approach identifies files with fewer LOC and higher overall defect densities. Previous studies have used prediction models to predict defect density to help allocate software quality assurance efforts (Kamei et al., 2010; Mende and Koschke, 2010). In our study, our approach outperforms the traditional approach (LM) in terms of saving efforts in code inspection and testing.

We note that the actual number of defects in the files identified by LM is higher, because the sheer size of each file is much larger. However, our approach can serve to complement LM, since the overlap between the files identified by our approach and LM is small (0–6.2%), which is because our approach identifies files of much

fewer LOC. In addition, developers are already aware that larger files are usually more defect prone and thus require more testing, meaning that our approach can help developers to also find the smaller files that require more testing. By using the two approaches in concert, developers and testers can locate additional defects.

We can predict defect prone and less tested topics with an average precision and recall of 0.77 and 0.75. In addition, we find that our approach outperforms traditional prediction-based resource allocation approaches in terms of saving testing and code inspection efforts. Our approach is able to identify parts of the systems that are defect prone and not well-tested, and help practitioners allocate testing resources more effectively.

4.4 Sensitivity Analysis for the Parameters of Our Approach

Our topic modeling approach (LDA) involves the choice of several input parameters, each of which may influence the results of our case study. In particular, LDA takes four parameters as input: number of topics (K), number of iterations (II), and two Dirichlet priors used for smoothing (α and β). In addition, we define three other parameters: W ratio (used to determine shared topics), δ (used in Equation 4.6), and PCA cut-off (used to determine number of PCs in the naive Bayes classifier).

We perform a sensitivity analysis to determine how sensitive our results are to our particular parameter value choices. We define a *baseline* set of parameter values, which are those values used in the aforementioned three research questions. In

Table 4.9: Median LOC, defect density, and percentage overlap of the source code files that are predicted to be in class LTHD and the most defect-prone files predicted by the linear regression model. The numbers in the parentheses indicate the percentage improvement of our approach over LM in terms of the defect density of the returned files. *% files* is the percentage of the source code files in a system that is used for comparing the two approaches.

Studied System	Prediction Approach	% files (n)	Median LOC	Defect Density (Defect/KLOC)	% overlapping files
Mylyn 2.0	Topic-based	2.6 %	56.5	9.83 (+45%)	0 %
	LM		652.0	6.76	
Mylyn 3.0	Topic-based	1.3 %	52.0	4.94 (+82%)	0 %
	LM		947.0	2.72	
Eclipse 2.1	Topic-based	4.2 %	23.0	4.08 (+274%)	1.8 %
	LM		450.0	1.09	
Eclipse 3.0	Topic-based	5.9 %	22.0	5.76 (+210%)	2.1 %
	LM		633.0	1.86	
NetBeans 5.0	Topic-based	5.2 %	78.0	0.39 (+56%)	2.0 %
	LM		447.0	0.25	
NetBeans 5.5.1	Topic-based	12.5%	64.0	0.48 (+4%)	6.2 %
	LM		376.0	0.46	

particular, the baseline values are: K=500, II=10,000, W cut-off=0.05, δ=0.01, and PCA cut-off=90%. (We use MALLET optimized α and β (McCallum, 2002).) We increase and decrease each parameter value independently to study the sensitivity of each parameter. For K, we consider $K \pm 100$, which are 400 and 600. For II, we consider $II \pm 1,000$, which are 9,000 and 11,000. For the W cut-off, we consider half and double of the baseline value, which are 0.025 and 0.1. For δ, we consider $\delta/2$ and $\delta * 2$, which are 0.005 and 0.02. Finally, we consider two different PCA cut-off parameters, which are 80% and 95%.

For each parameter, we repeat the experiments in RQ2 and RQ3, and report the MIC score from RQ2 (we study what is the relationship between topic defect density and topic testedness, and we found a non-coexistence relationship), and the precision and recall from RQ3 (we study if we can find not well-tested topics and defect-prone topics). Table 4.10 – 4.11 shows only the parameters that may influence the MIC score, since the topics will be slightly different when K, II, and W cut-off change (the other two parameters cannot change the MIC score, since they do not change the topics). Table 4.12 – 4.13 shows the precision and recall of all the parameters that may influence the classification result. Note that, since we are doing cross-release prediction, the precision and recall of the first version of each subject system are not possible.

In this chapter, we follow the guidelines of previous work (Chen et al., 2012; Lukins et al., 2010) for choosing K=500 for Eclipse and Mylyn. We also used K=500 for NetBeans, since the size of NetBeans is similar to that of Eclipse (Table 4.1). We see from Table 4.10 – 4.11 that in most cases, when K, W cut-off change, and II change, the MIC scores remain stable. Changing K varies the MIC

score by an average of 13% from the baseline value; changing II varies the MIC score by an average of 8%; and changing W cut-off varies the MIC score by an average of 8%. These three parameters determine the number of data points (topics) and topic accuracy in the dataset, which has a direct effect on the overall relationship between topic testedness and defect density. One thing to note is that the MINE measures of all the systems follow the same pattern as described in RQ2, which supports our hypothesis that the relationship is non-coexistence.

4.5 Threats to Validity

4.5.1 Parameter Sensitivity Analysis

Our results may be affected by changing the parameters (i.e., K, II, W cut-off, δ, and PCA cut-off). Although we performed a parameter sensitivity analysis in Section 4.4, we only change one variable at a time. Further research is required to understand the effects of these parameters on the results (e.g., change all the parameters at the same time).

Since it is possible the source-only topics that are excluded in RQ1 belong to the class LTHD, we study how many such topics are there and how defect prone they are (Table 4.14). We find that most of the excluded topics do not belong to LTHD (81%–94%), and these topics have a low overall defect density. Therefore, these topics do not significantly affect our results.

We use the third quantile of topic testedness and defect density to classify topics into three classes. However, this threshold can be changed, and more advanced clustering algorithms may be used. This clustering decision may affect the final

Table 4.10: Results of the parameter sensitivity analysis of the parameters that may influence *the MIC score*. The baseline parameters and their values are shown in the table. For each system, we show the MIC score when the parameter changes. The values in the parentheses indicate the increase/decrease from the baseline MIC score.

Baseline Values: K=500, II=10,000, W cut-off=0.05			
Lower		*Higher*	
New Value	MIC	New Value	MIC
Mylyn 1.0 (Baseline MIC = 0.38)			
K=400	0.29 (-0.09)	K=600	0.52 (+0.14)
II=9K	0.44 (+0.06)	II=11K	0.37 (-0.01)
W cut-off=0.025	0.35 (-0.03)	W cut-off=0.1	0.39 (+0.01)
Mylyn 2.0 (Baseline MIC = 0.40)			
K=400	0.39 (-0.01)	K=600	0.55 (+0.15)
II=9K	0.41 (+0.01)	II=11K	0.41 (+0.01)
W cut-off=0.025	0.38 (-0.02)	W cut-off=0.1	0.41 (+0.01)
Mylyn 3.0 (Baseline MIC = 0.26)			
K=400	0.29 (+0.03)	K=600	0.32 (+0.06)
II=9K	0.27 (+0.01)	II=11K	0.27 (+0.01)
W cut-off=0.025	0.36 (+0.10)	W cut-off=0.1	0.25 (-0.01)
Eclipse 2.0 (Baseline MIC = 0.20)			
K=400	0.25 (+0.05)	K=600	0.21 (+0.01)
II=9K	0.22 (+0.02)	II=11K	0.19 (-0.01)
W cut-off=0.025	0.21 (+0.01)	W cut-off=0.1	0.21 (+0.01)
Eclipse 2.1 (Baseline MIC = 0.21)			
K=400	0.18 (-0.03)	K=600	0.20 (-0.01)
II=9K	0.22 (+0.01)	II=11K	0.20 (-0.01)
W cut-off=0.025	0.25 (+0.04)	W cut-off=0.1	0.22 (+0.01)
Eclipse 3.0 (Baseline MIC = 0.24)			
K=400	0.25 (+0.01)	K=600	0.23 (-0.01)
II=9K	0.21 (-0.03)	II=11K	0.20 (-0.04)
W cut-off=0.025	0.25 (+0.01)	W cut-off=0.1	0.22 (-0.02)
Continued in Table 4.11.			

Table 4.11: Continued from Table 4.10. Results of the parameter sensitivity analysis of the parameters that may influence *the MIC score*. The baseline parameters and their values are shown in the table. For each system, we show the MIC score when the parameter changes. The values in the parentheses indicate the increase/decrease from the baseline MIC score.

Baseline Values: K=500, II=10,000, W cut-off=0.05			
Lower		*Higher*	
New Value	MIC	New Value	MIC
NetBeans 4.0 (Baseline MIC = 0.24)			
K=400	0.19 (-0.05)	K=600	0.25 (+0.01)
II=9K	0.23 (-0.01)	II=11K	0.29 (+0.05)
W cut-off=0.025	0.27 (+0.03)	W cut-off=0.1	0.26 (+0.02)
NetBeans 5.0 (Baseline MIC = 0.20)			
K=400	0.21 (+0.01)	K=600	0.19 (-0.01)
II=9K	0.23 (+0.02)	II=11K	0.21 (+0.01)
W cut-off=0.025	0.21 (+0.01)	W cut-off=0.1	0.19 (-0.01)
NetBeans 5.5.1 (Baseline MIC = 0.19)			
K=400	0.18 (-0.01)	K=600	0.18 (-0.01)
II=9K	0.18 (-0.01)	II=11K	0.20 (+0.01)
W cut-off=0.025	0.20 (+0.01)	W cut-off=0.1	0.19 (—)

Table 4.12: Results of the parameter sensitivity analysis of the parameters that may influence the *prediction result*. The baseline parameters and their values are shown in the table. For each system, we show the precision and recall when the parameter changes. The values in the parentheses indicate the increase/decrease from the baseline precision and recall.

Baseline Values: K=500, II=10,000, W cut-off=0.05, δ=0.01, PCA cut-off=90%			
Lower		*Higher*	
New Value	Precision Recall	New Value	Precision Recall
Mylyn 2.0 (Baseline Precision = 0.88, Base Recall = 0.80)			
K=400	0.85 (-0.03) 0.80 (—)	K=600	0.80 (-0.08) 0.81 (+0.01)
II=9K	0.84 (-0.04) 0.82 (+0.02)	II=11K	0.90 (+0.02) 0.80 (—)
W cut-off=0.025	0.89 (+0.01) 0.84 (+0.04)	W cut-off=0.1	0.84 (-0.04) 0.84 (+0.04)
δ=0.005	0.88 (—) 0.82 (+0.02)	δ=0.02	0.86 (-0.02) 0.79 (-0.01)
PCA cut-off=80%	0.88 (—) 0.80 (—)	PCA cut-off=95%	0.88 (—) 0.80 (—)
Mylyn 3.0 (Baseline Precision = 0.82, Base Recall = 0.71)			
K=400	0.80 (-0.02) 0.73 (+0.02)	K=600	0.68 (-0.14) 0.72 (+0.01)
II=9K	0.78 (-0.04) 0.74 (+0.03)	II=11K	0.80 (-0.02) 0.71 (—)
W cut-off=0.025	0.78 (-0.04) 0.66 (-0.05)	W cut-off=0.1	0.83 (+0.01) 0.71 (—)
δ=0.005	0.82 (—) 0.73 (+0.02)	δ=0.02	0.83 (+0.01) 0.71 (—)
PCA cut-off=80%	0.82 (—) 0.71 (—)	PCA cut-off=95%	0.82 (—) 0.71 (—)
Eclipse 2.1 (Baseline Precision = 0.79, Base Recall = 0.63)			
K=400	0.82 (+0.03) 0.63 (—)	K=600	0.85 (+0.06) 0.72 (+0.09)
II=9K	0.77 (-0.02) 0.67 (+0.04)	II=11K	0.80 (+0.01) 0.61 (-0.02)
W cut-off=0.025	0.79 (—) 0.62 (-0.01)	W cut-off=0.1	0.78 (-0.01) 0.65 (+0.02)
δ=0.005	0.79 (—) 0.63 (—)	δ=0.02	0.81 (+0.02) 0.63 (—)
PCA cut-off=80%	0.79 (—) 0.63 (—)	PCA cut-off=95%	0.79 (—) 0.63 (—)
Eclipse 3.0 (Baseline Precision = 0.84, Base Recall = 0.79)			
K=400	0.78 (-0.06) 0.84 (+0.05)	K=600	0.77 (-0.07) 0.86 (+0.07)
II=9K	0.80 (-0.04) 0.79 (—)	II=11K	0.84 (—) 0.78 (-0.01)
W cut-off=0.025	0.82 (-0.02) 0.78 (-0.01)	W cut-off=0.1	0.84 (—) 0.81 (+0.02)
δ=0.005	0.84 (—) 0.81 (+0.02)	δ=0.02	0.81 (-0.03) 0.79 (—)
PCA cut-off=80%	0.84 (—) 0.79 (—)	PCA cut-off=95%	0.84 (—) 0.79 (—)

Continued in Table 4.13.

Table 4.13: Continued from Table 4.12. Results of the parameter sensitivity analysis of the parameters that may influence the *prediction result*. The baseline parameters and their values are shown in the table. For each system, we show the precision and recall when the parameter changes. The values in the parentheses indicate the increase/decrease from the baseline precision and recall.

Baseline Values: K=500, II=10,000, W cut-off=0.05, δ=0.01, PCA cut-off=90%			
Lower		*Higher*	
New Value	Precision Recall	New Value	Precision Recall
NetBeans 5.0 (Baseline Precision = 0.65, Base Recall = 0.60)			
K=400	0.60 (-0.05) 0.63 (+0.03)	K=600	0.67 (+0.02) 0.58 (-0.02)
II=9K	0.63 (-0.02) 0.59 (-0.01)	II=11K	0.64 (-0.01) 0.59 (-0.01)
W cut-off=0.025	0.65 (—) 0.57 (-0.03)	W cut-off=0.1	0.65 (—) 0.63 (+0.03)
δ=0.005	0.65 (—) 0.60 (—)	δ=0.02	0.63 (-0.02) 0.60 (—)
PCA cut-off=80%	0.65 (—) 0.60 (—)	PCA cut-off=95%	0.65 (—) 0.60 (—)
NetBeans 5.5.1 (Baseline Precision = 0.64, Base Recall = 0.86)			
K=400	0.67 (+0.03) 0.94 (+0.08)	K=600	0.65 (+0.01) 0.79 (-0.07)
II=9K	0.68 (+0.04) 0.88 (+0.02)	II=11K	0.65 (+0.01) 0.85 (-0.01)
W cut-off=0.025	0.66 (+0.02) 0.86 (—)	W cut-off=0.1	0.62 (-0.02) 0.85 (-0.01)
δ=0.005	0.65 (+0.01) 0.86 (—)	δ=0.02	0.61 (-0.03) 0.85 (-0.01)
PCA cut-off=80%	0.64 (—) 0.86 (—)	PCA cut-off=95%	0.64 (—) 0.86 (—)

Table 4.14: Summary of the excluded source-only topics that belong to the class LTHD, and the median defect density of these topics.

System	% Source-only Topics that are in LTHD	Median Defect Density
Mylyn 1.0	10 %	$3.9 * 10^{-3}$
Mylyn 2.0	16 %	$6.3 * 10^{-3}$
Mylyn 3.0	7 %	$1.8 * 10^{-3}$
Eclipse 2.0	19 %	$6.9 * 10^{-3}$
Eclipse 2.1	8 %	$4.9 * 10^{-3}$
Eclipse 3.0	11 %	$8.4 * 10^{-3}$
NetBeans 4.0	6 %	$5.1 * 10^{-6}$
NetBeans 5.0	8 %	$3.5 * 10^{-6}$
NetBeans 5.5.1	8 %	$2.9 * 10^{-4}$

result. Further analysis is needed to examine the effect of topic clustering on the results of our study.

4.5.2 Classifier Choice

The goal of this chapter is to provide initial evidence that it is possible to analyze code coverage using topic models, and to identify topics of the source code files that require more testing. We choose a naive Bayes classifier, which is shown to have a good performance in classifying defective files (Menzies et al., 2007; Turhan and Bener, 2009). However, other models may also be used, and different models may have slightly different classification performance.

4.5.3 Subject Systems

In our case study, we considered in detail three versions of Mylyn, Eclipse, and NetBeans. However, these three systems may not necessarily be representative of all possible software systems. Further studies are needed to determine the generalizability of our results.

4.6 Guideline for Implementing Our Approach

In this section, we provide a step-by-step guideline to help practitioners implement our approach and use it on their software systems.

1. Obtain both the source code and test files of the software system (use the heuristic in Section 4.2.1 to identify test files).

2. Obtain the defect history (i.e., how many pre-release and post-release defects a file has) for the source code files of the previous version of the software system.

3. Preprocess the source and test code. Take only the identifier names and comments from the source and test code files, remove common English stopwords, and finally stem the words (Section 2.2).

4. Use the MALLET tool to apply LDA on the preprocessed source and test code (Section 4.2).

5. Compute the topic metrics proposed in Section 4.3.1. Determine which classes the topics are in, and train a classifier using the topic testedness and post-release defect information from the previous version. The goal is to identify which topics are low-tested and high defect-prone.

6. Using the classifier built using the data from previous version, and predict which topics are more defect-prone and not well-tested. Allocate more testing resources on these topics.

4.7 Chapter Summary

In this chapter, we have studied the effect of topic testedness, i.e., the extent to which a topic is tested, on code quality. We proposed new topic-based metrics to study this relationship. We performed a detailed case study on three large, real-world systems: Mylyn, Eclipse, and NetBeans. The summary and highlights of our findings include:

- Test files and source code files share a significant number of topics.
- Defect density and testedness have a non-coexistence relationship, i.e., cannot both have high values. Highly tested topics are unlikely to have high defect density; high defect density topics have low testedness.
- We are able to predict, with high recall and precision, whether an under-tested topic is at risk of containing defects, which can help practitioners allocate testing sources more effectively.
- Our approach outperforms traditional prediction-based resource allocation methods in terms of allocating testing and code inspection resources.
- Our proposed method is not sensitive to the particular thresholds that we used.

Chapter 5

Summary and Conclusions

We conclude the thesis in this chapter, and summarize our findings and results to our research questions. We also discuss limitations of our approaches and possible future research directions.

5.1 Summary

Software quality plays an important role in software engineering, since it directly affects the quality of the product that the customers receive. Researchers have tried to uncover possible reasons (different software development activities) which may possibly lead to higher defects in the software. However, despite their successes, previous approaches do not consider the concerns in the software when studying software quality. For example, lines of code (LOC) is shown to be one of the best predictors for defects. However, a large file that is responsible for simple tasks, such as storing constant or pre-defined values, may have very low or no defects.

In this thesis, we approximate concerns in software systems as *topics* using topic

models, and study software quality using topics. Our goal is to examine how we can use topics to help explain software defects and assist software quality assurance. To do so, we perform case studies on four large real-world software systems in Chapter 3 and three software systems in Chapter 4, and we find that:

Topics can help explain software defects and help allocate quality assurance efforts.

We study software quality using topics along two dimensions: code quality and code testedness. We summarize our findings in the following:

Using Topic Models to Study Code Quality

In Chapter 3, we perform a preliminary study, and examine whether there are some relationships between topics and defects. We use number of defects in a file to represent the code quality. We propose an approach to measure the defect-proneness of a topic (i.e., topic defect density) using the defect history of topics. We find that only a few topics in a system is defect prone, and these defect prone topics seem to remain defect-prone over time. We look at the words that are related to these defect-prone topics, and find that topics related to new features and the core functionality are likely to be more defect-prone.

We propose various topic-based metrics, which we use to study code quality. For static topic-based metrics, we have number of topics (NT) and topic memberships (TM), and for historical topic-based metrics, we have number of defect-prone topics (NDT) and defect-prone topic memberships (DTM). We show that our metrics can give improvements (4–314%) to the baseline models (LOC, or PRE and churn) in explaining defects. We compare NT with other state-of-the-art topic-based cohesion

and coupling metrics, and examine their defect explanatory power. Our metric, NT, gives the greatest improvement over the LOC (2.9 97%), and we show that NT is not highly correlated with LOC. In addition, NT is much simpler to implement than other topic-based cohesion and coupling metrics.

Using Topic Models to Study Code Testedness

In Chapter 4, we study code testedness (how much a piece of code is tested) using topic models. Since testers usually test the systems at the level of concerns (Weyuker, 1998), identifying which concerns may require testing help prioritize testing efforts. We show that topics that are well tested are unlikely to be defect prone, and topics that are defect prone are usually less tested (we describe this relationship as non-coexistence). Through empirical studies, we show that we are able to obtain a precision and recall of 0.77 and 0.75 when predicting defect prone and less tested topics in *future releases of the system*. Finally, we show our approach is able to save code inspection efforts, and it finds a different set of defect prone files than the ones that are found by traditional approaches (only 0–6.2% overlap).

5.2 Limitations and Future Work

The work presented in this thesis has a number of limitations. More detailed limitations and threats to validity of our specific works are described in previous chapters (Section 3.5 and 4.5). In this section, we discuss the overall limitations and possible future research directions.

- Even though we have experimented our approaches with four large real-world systems, our results may still not be generalizable to all software systems. For example, in Eclipse, our NT metric and other topic-based cohesion and coupling metrics do not give any improvement in deviance explained (less than 1%) to the baseline model. Further studies are required to inspect possible reasons these metrics do not work as well in some systems.

- We make the assumption that the defect information that we obtain from the commit logs are correct. For example, when a developer fixes a file, he/she always includes certain related messages (i.e., *bug fixed*) when committing changes to the file. However, the data may contain some noises, which may affect our results.

- Our approaches depend on the quality of the topics that are generated using identifier names and comments in software systems. If the quality of the topics is low, then the overall performance of our approaches may be affected.

- We use LOC, or PRE and churn as base metrics for comparison. Although these metrics are found to be the most effective metrics for explaining software quality, there may be some other confounding variables that may affect the results of our metrics.

- Our approaches involve choosing parameter values. However, through sensitivity analysis in Chapter 3 and 4, we find out that our approaches are not particular sensitivity to the parameter values that we choose.

- In this thesis, we capture concerns from source code and test files, but other information, such as email and bug reports, is also available. It would be

interesting to find out the relationships between, for example, email discussions and software defects. In addition, by adding more linguistic information to source code files (i.e., commit messages of the files), would we achieve a higher defect explanatory power?

Bibliography

(n.d.).

Allen, E. B. and Khoshgoftaar, T. M. (1999), Measuring coupling and cohesion: An information-theory approach, *in* 'Proceedings of the 6th International Symposium on Software Metrics', pp. 119–127.

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

Baldi, P. F., Lopes, C. V., Linstead, E. J. and Bajracharya, S. K. (2008), A theory of aspects as latent topics, *in* 'Proceedings of the 23rd ACM SIGPLAN Conference on Object-oriented Programming Systems Languages and Applications', pp. 543–562.

Bieman, J. M. and Kang, B.-K. (1998), "Measuring design-level cohesion", *IEEE Transactions on Software Engineering* , Vol. 24, pp. 111–124.

Bird, C., Nagappan, N., Murphy, B., Gall, H. and Devanbu, P. (2011), Don't touch my code!: Examining the effects of ownership on software quality, *in* 'Proceedings

of the 19th Symposium on the Foundations of Software Engineering and the 13rd European Software Engineering Conference', pp. 4–14.

Biyani, S. and Santhanam, P. (1998), Exploring defect data from development and customer usage on software modules over multiple releases, *in* 'Proceedings of the 9th International Symposium on Software Reliability Engineering', pp. 316–320.

Blei, D. M., Ng, A. Y. and Jordan, M. I. (2003), "Latent Dirichlet allocation", *Journal of Machine Learning Research* , Vol. 3, pp. 993–1022.

Briand, L. C., Daly, J. W. and Wüst, J. (1998), "A unified framework for cohesion measurement in object-oriented systems", *Empirical Software Engineering* , Vol. 3, pp. 65–117.

Brown, P. F., deSouza, P. V., Mercer, R. L., Pietra, V. J. D. and Lai, J. C. (1992), "Class-based n-gram models of natural language", *Computational Linguistics* , Vol. 18, pp. 467–479.

Burnham Kenneth P., A. D. R. (2004), "Multimodel inference: Understanding AIC and BIC in model selection", *Sociological Methods Research* , Vol. 33, pp. 467–479.

Chae, H. S., Kwon, Y. R. and Bae, D.-H. (2000), "A cohesion measure for object-oriented classes", *Software Practice & Experience* , Vol. 30, pp. 1405–1431.

Chang, J. and Blei, D. (2009), Relational topic models for document networks, *in* 'Proceedings of the 12th International Conference on Articial Intelligence and Statistics'.

Chen, T.-H., Thomas, S. W., Nagappan, M. and Hassan, A. (2012), Explaining software defects using topic models, *in* 'Proceedings of the 9th Working Conference on Mining Software Repositories'.

Cleary, B., Exton, C., Buckley, J. and English, M. (2008), "An empirical analysis of information retrieval based concept location techniques in software comprehension", *Empirical Software Engineering* , Vol. 14, pp. 93–130.

Crawford, S. G., McIntosh, A. A. and Pregibon, D. (1985), "An analysis of static metrics and faults in c software", *Journal of Systems and Software* , Vol. 5, pp. 37–48.

Cureton, E. and D'Agostino, R. (1993), *Factor Analysis: An Applied Approach*, Lawrence Erlbaum Associates.

DAmbros, M., Lanza, M. and Robbes, R. (2010), An extensive comparison of bug prediction approaches, *in* 'Proceedings of the 7th Conference on Mining Software Repositories', pp. 31–41.

Fenton, N. (1991), *Software metrics: a rigorous approach*, Chapman & Hall.

Foundation, E. (2012), Mylyn. http://www.eclipse.org/mylyn/.

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

Golberg, M. and Cho, H. (2004), *Introduction to regression analysis*.

Gyimothy, T., Ferenc, R. and Siket, I. (2005), "Empirical validation of object-oriented metrics on open source software for fault prediction", *IEEE Transactions on Software Engineering* , Vol. 31, pp. 897–910.

Haan, C. (1977), *Statistical methods in hydrology*, Iowa State University Press.

Hall, T., Beecham, S., Bowes, D., Gray, D. and Counsell, S. (2011), "A systematic review of fault prediction performance in software engineering", *IEEE Transactions on Software Engineering* , Vol. PP.

Huang, J. C. (1975), "An approach to program testing", *ACM Comput. Surv.* , Vol. 7, pp. 113–128.

Ihara, S. (1993), *Information Theory for Continuous Systems*, World Scientific.

Jolliffe, I. (2002), *Principal component analysis*, Springer-Verlag.

Kamei, Y., Matsumoto, S., Monden, A., Matsumoto, K.-i., Adams, B. and Hassan, A. E. (2010), Revisiting common bug prediction findings using effort-aware models, *in* 'Proceedings of the 2010 IEEE International Conference on Software Maintenance', pp. 1–10.

Kan, S. H. (2002), *Metrics and Models in Software Quality Engineering*, 2nd edn, Addison-Wesley Longman Publishing Co., Inc.

Kuhn, A., Ducasse, S. and Gírba, T. (2007), "Semantic clustering: Identifying topics in source code", *Information and Software Technology* , Vol. 49, pp. 230–243.

Kutner, M., Nachtsheim, C. and Neter, J. (2004), *Applied linear regression models*.

Linstead, E., Lopes, C. and Baldi, P. (2008), An application of latent Dirichlet allocation to analyzing software evolution, *in* 'Proceedings of Seventh International Conference on Machine Learning and Applications', pp. 813–818.

Liu, Y., Poshyvanyk, D., Ferenc, R., Gyimothy, T. and Chrisochoides, N. (2009), Modeling class cohesion as mixtures of latent topics, *in* 'Proceedings of the 25th International Conference on Software Maintenance', pp. 233 –242.

Lukins, S. K., Kraft, N. A. and Etzkorn, L. H. (2010), "Bug localization using latent dirichlet allocation", *Information and Software Technology* , Vol. 52, pp. 972–990.

Macro, A. and Buxton, J. (1987), *The craft of software engineering*, Addison-Wesley.

Marcus, A. and Poshyvanyk, D. (2005), The conceptual cohesion of classes, *in* 'Proceedings of the 21st IEEE International Conference on Software Maintenance', pp. 133–142.

Marcus, A., Poshyvanyk, D. and Ferenc, R. (2008), "Using the conceptual cohesion of classes for fault prediction in object-oriented systems", *IEEE Transactions on Software Engineering* , Vol. 34, pp. 287–300.

Maskeri, G., Sarkar, S. and Heafield, K. (2008), Mining business topics in source code using latent Dirichlet allocation, *in* 'Proceedings of the 1st India Software Engineering Conference', pp. 113–120.

McCallum, A. K. (2002), Mallet: A machine learning for language toolkit. http://mallet.cs.umass.edu.

Mende, T. and Koschke, R. (2010), Effort-aware defect prediction models, *in* 'Proceedings of the 2010 14th European Conference on Software Maintenance and Reengineering', pp. 107–116.

Menzies, T., Greenwald, J. and Frank, A. (2007), "Data mining static code attributes to learn defect predictors", *IEEE Transactions on Software Engineering* , Vol. 33, pp. 2–13.

Myers, G., Badgett, T., Thomas, T. and Sandler, C. (2004), *The Art of Software Testing*, 2nd edn.

Nagappan, N. and Ball, T. (2005), Use of relative code churn measures to predict system defect density, *in* 'Proceedings of the 27th international conference on Software engineering', ICSE '05, pp. 284–292.

Nagappan, N., Williams, L., Vouk, M. and Osborne, J. (2007), Using in-process testing metrics to estimate post-release field quality, *in* 'Proceedings International Symposium on Software Reliability Engineering', pp. 209–214.

Nguyen, T. T., Nguyen, T. N. and Phuong, T. M. (2011), Topic-based defect prediction, *in* 'Proceedings of the 33rd International Conference on Software Engineering', pp. 932–935.

Ntafos, S. (1988), "A comparison of some structural testing strategies", *IEEE Transactions on Software Engineering* , Vol. 14, pp. 868 –874.

Oram, A. and Wilson, G. (2010), *Making Software: What Really Works, and Why We Believe It*, O'Reilly Series, O'Reilly Media, Incorporated.

Poshyvanyk, D., Gueheneuc, Y., Marcus, A., Antoniol, G. and Rajlich, V. (2007), "Feature location using probabilistic ranking of methods based on execution scenarios and information retrieval", *IEEE Transactions on Software Engineering* , pp. 420–432.

Poshyvanyk, D. and Marcus, A. (2006), The conceptual coupling metrics for object-oriented systems, *in* 'Proceedings of the 22nd IEEE International Conference on Software Maintenance', pp. 469–478.

Posnett, D., Devanbu, P. and Filkov, V. (2012), Mic check: A correlation tactic for ESE data, *in* 'Proceedings of the 9th Working Conference on Mining Software Repositories'.

Provost, F. (2000), Machine learning from imbalanced data sets 101 (extended abstract), *in* 'Proceedings of the AAAI2000 Workshop on Imbalanced Data Sets'.

Raftery, A. (1995), "Bayesian model selection in social research (with discussion)", *Sociological Methodology* , Vol. 25, pp. 111–163.

Rao, S. and Kak, A. (2011), 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', pp. 43–52.

Reshef, D. N., Reshef, Y. A., Finucane, H. K., Grossman, S. R., McVean, G., Turnbaugh, P. J., Lander, E. S., Mitzenmacher, M. and Sabeti, P. C. (2011), Detecting novel associations in large data sets, Vol. 334, pp. 1518–1524.

Reshef, D., Reshef, Y., Sabeti, P. and Mitzenmacher, M. (2012), Mine: maximal information-based nonparametric exploration. http://www.exploredata.net/.

Revelle, M., Gethers, M. and Poshyvanyk, D. (2011), "Using structural and textual information to capture feature coupling in object-oriented software", *Empirical Software Engineering* , Vol. 16, pp. 773–811.

Robillard, M. P. and Murphy, G. C. (2007), "Representing concerns in source code", *ACM Transactions on Software Engineering and Methodology* , Vol. 16, p. 3.

Rosenberg, J. (1997), Some misconceptions about lines of code, *in* 'Proceedings of the 4th International Symposium on Software Metrics', pp. 137–142.

Slaughter, S. A., Harter, D. E. and Krishnan, M. S. (1998), "Evaluating the cost of software quality", *Communications of the ACM* , Vol. 41, pp. 67–73.

Stapleton, J. (2008), *Models for probability and statistical inference: theory and applications*.

Stewart, J. (2009), *Calculus: Concepts and Contexts*, Stewart's Calculus Series, Cengage Learning.

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

Thomas, S. W., Adams, B., Hassan, A. E. and Blostein, D. (2011), Modeling the evolution of topics in source code histories, *in* 'Proceedings of the 8th Working Conference on Mining Software Repositories', pp. 173–182.

Tian, K., Revelle, M. and Poshyvanyk, D. (2009), Using latent Dirichlet allocation for automatic categorization of software, *in* 'Proceedings of the 6th International Working Conference on Mining Software Repositories', pp. 163–166.

Titov, I. and McDonald, R. (2008), Modeling online reviews with multi-grain topic models, *in* 'Proceedings of the 17th international conference on World Wide Web', pp. 111–120.

Turhan, B. and Bener, A. (2009), "Analysis of naive bayes assumptions on software fault data: An empirical study", *Data and Knowledge Engineering* , Vol. 68, pp. 278–290.

Ujhazi, B., Ferenc, R., Poshyvanyk, D. and Gyimothy, T. (2010), New conceptual coupling and cohesion metrics for object-oriented systems, *in* 'Proceedings of the 2010 10th IEEE Working Conference on Source Code Analysis and Manipulation', pp. 33–42.

Wallach, H., Mimno, D. and McCallum, A. (2009), "Rethinking LDA: Why priors matter", *Proceedings of Neural Information Processing Systems, Vancouver, BC* .

Weyuker, E. (1998), "Testing component-based software: a cautionary tale", *IEEE Software* , Vol. 15, pp. 54 –59.

Witten, I., Frank, E. and Hall, M. (2011), *Data Mining: Practical Machine Learning Tools and Techniques*, Elsevier Science.

Zaidman, A., Rompaey, B. V., Demeyer, S. and Deursen, A. v. (2008), Mining software repositories to study co-evolution of production & test code, *in* 'Proceedings of the 1st International Conference on Software Testing, Verification, and Validation', pp. 220–229.