MINING DEVELOPMENT KNOWLEDGE TO UNDERSTAND AND SUPPORT SOFTWARE LOGGING PRACTICES

by

HENG LI

A thesis submitted to the Graduate Program in Computing

in conformity with the requirements for the

Degree of Doctor of Philosophy

Queen's University

Kingston, Ontario, Canada

December 2018

Abstract

DEVELOPERS insert logging statements in their source code to trace the runtime behaviors of software systems. Logging statements print runtime log messages, which play a critical role in monitoring system status, diagnosing field failures, and bookkeeping user activities. However, developers typically insert logging statements in an *ad hoc* manner, which usually results in fragile logging code, i.e., insufficient logging in some code snippets and excessive logging in other code snippets. Insufficient logging can significantly increase the difficulty of diagnosing field failures, while excessive logging can cause performance overhead and hide truly important information. The goal of this thesis is to help developers improve their logging practices and the quality of their logging code. We believe that development knowledge (i.e., source code, code change history, and issue reports) contains valuable information that explains developers' rationale of logging, which can help us understand existing logging practices and provide helpful tooling support for such logging practices.

Therefore, this thesis proposes to mine different aspects of development knowledge to understand and support software logging practices. We mine issue reports to understand developers' logging concerns, i.e., the benefits and costs of logging from developers' perspective. Our findings shed lights on future research opportunities for helping developers leverage the benefits of logging while minimizing logging costs. We mine source code to learn how developers distribute logging statements in their source code, and propose an approach to provide automated suggestions about *where to log*. We find that the semantic topics of a code snippet provide another dimension to explain the likelihood of logging a code snippet. We mine code change history to understand how developers develop and maintain their logging code, and propose an automated approach that can provide developers with log change suggestions when they change their code. We also mine code change history to understand how developers choose log levels for their logging statements, and propose an automated approach that can help developers determine the appropriate log level when they add a new logging statement. This thesis highlights the need for standard logging guidelines and automated tooling support for logging.

Acknowledgments

I would like to thank my supervisor Dr. Ahmed E. Hassan for all his guidance and support throughout these four years. His academic advising has fostered my enthusiasm in scientific research and guided me to grow as a researcher. In addition, his valuable support for my personal life has helped me go through many difficulties in my life. I feel very grateful and lucky that I had the chance to work under his supervision. I would like to express my sincere gratitude to Dr. Ahmed E. Hassan, my advisor and my friend.

A sincere appreciation to my supervisory and examination committee members, Dr. Hossam S. Hassanein, Dr. Mohammad Zulkernine, and Dr. Joshua Dunfield for their continued critique, support and guidance. Many thanks to my examiners for their insightful feedback and valuable advice on my work.

I am very lucky to work with some of the brightest researchers during my Ph.D. research. I would like to thank all of my colleagues and collaborators, including Dr. Weiyi Shang, Dr. Tse-Hsun (Peter) Chen, Dr. Ying (Jenny) Zou, Dr. Bram Adams, Safwat

Hassan, Filipe Cogo, Dr. Shane McIntosh, Dr. Cor-Paul Bezemer, Dr. Shaowei Wang, Dr. Mohamed Sami Rakha, Dayi Lin, Dr. Gustavo Oliva, Dr. Muhammad Asaduzzam, and Suhas Kabinna for all the fruitful discussions and collaborations.

I would like to thank the BlackBerry company and the members of the BlackBerry Performance Engineering team. In particular, I would like to thank Mohamed Nasser and Parminder Flora for their continuous support during my internship with BlackBerry. The industrial environment has encouraged and will continue to encourage my research that deals with practical challenges in the field.

During these years I have been receiving so much love from many people, including Dehui Kong, Xubo Miao, Chuansheng Dong, Jiaxing Len, Zhen Zeng, Min Xie, Zhiying Zhang, Panpan Zhang, Safwat Hassan, Stephen Hung, Hannah Zhang, Nancy Smith, and Shaowei Wang, and their families. Thank you all for making my Ph.D. life enjoyable and wonderful.

A special thanks to my family. My gratitude for all your support and sacrifices is beyond any words. I am so proud of being one of your family member. I dedicate this thesis to my family.

Co-authorship

The work presented in the thesis is published as listed below:

- Heng Li, Weiyi Shang, Ying Zou, and Ahmed E. Hassan. 2017. Towards just-in-time suggestions for log changes. Empirical Software Engineering: An International Journal (EMSE). Volume 22, Issue 4, pages 1831–1865. This work is described in Chapter 5.

- Heng Li, Weiyi Shang, and Ahmed E. Hassan. 2017. Which log level should developers choose for a new logging statement? Empirical Software Engineering: An International Journal (EMSE). Volume 22, Issue 4, pages 1684–1716. This work is described in Chapter 6.

- Heng Li, Tse-Hsun (Peter) Chen, Weiyi Shang, and Ahmed E. Hassan. 2018. Studying software logging using topic models. Empirical Software Engineering: An International Journal (EMSE). In Press. This work is described in Chapter 4.

- Heng Li, Weiyi Shang, Bram Adams, and Ahmed E. Hassan. A qualitative study of developers' logging concerns. IEEE Transactions on Software Engineering (TSE). Under review. This work is described in Chapter 3.

For each of the work, my contributions include proposing the initial research idea, investigating background knowledge and related work, proposing research methods, conducting experiments and collecting the data, analyzing the data and verifying the hypotheses, and writing and polishing the manuscript. My co-contributors supported me in refining the initial ideas, providing suggestions for potential research methods, providing feedback on experimental results and earlier manuscript drafts, and provide advice for polishing the writing.

Table of Contents

List of Tables

List of Figures

CHAPTER 1

Introduction

SOFTWARE developers insert logging statements in their source code to record valuable runtime information. A logging statement, when executed at runtime, prints a time-stamped log message into a pre-specified log file. Log messages help software practitioners (i.e., developers, testers, and operators) better understand system behaviors at runtime and assist in quality assurance efforts. For example, software developers rely on log messages for debugging field failures (Glerum et al., 2009; Yuan et al., 2010). Software operators leverage the rich information in log messages to guide capacity planning efforts (Sharma et al., 2011; Kavulya et al., 2010), to monitor system health (Bitincka et al., 2010), and to identify abnormal behaviors (Fu et al., 2009; Xu et al., 2009b; Jiang et al., 2008).

However, developers usually insert logging statements in an *ad hoc* manner (Yuan et al., 2012b). As a result, software practitioners usually miss important logging statements in a system, which often results in difficulties when debugging a field issue (Yuan et al., 2010). Nevertheless, adding logging statements excessively is not a good solution, since adding unnecessary logging statements can negatively impact system performance (Zeng et al., 2015) and mask the truly important information (Fu et al., 2014). In practice, providing appropriate logging statements (i.e., maximizing the value of the logged information while minimizing logging overhead) remains a challenge for software developers (Fu et al., 2014; Yuan et al., 2012b).

Prior studies proposed approaches to improve logging through *proactive logging* (Yuan et al., 2012a, 2011) and *learning to log* (Zhu et al., 2015; Jia et al., 2018). *Proactive logging* approaches use static analysis to conservatively add more logged information to the existing code (e.g., in exception catch blocks), in order to improve software failure diagnosis. However, these approaches do not consider developers' expertise and significantly increase the amount of logged information (i.e., excessive logging). *Learning to log* approaches, on the other hand, learn statistical models from existing logging code and further leverage the models to provide suggestions for new logging code. These approaches have four main drawbacks: 1) They focus on specific types of code snippets (e.g., exception catch blocks or function calls) which together only cover 41% of the logging instances (Fu et al., 2014); 2) They provide logging suggestions as a post-development process instead of providing logging suggestions during the development process, i.e., when developers are changing their code; 3) They do not consider logging patterns (e.g., log levels, or stack trace logging) which also play important roles in determining the overall amount of log information; and 4) They do not consider developers' code change history and issue reports that explain the rationale behind

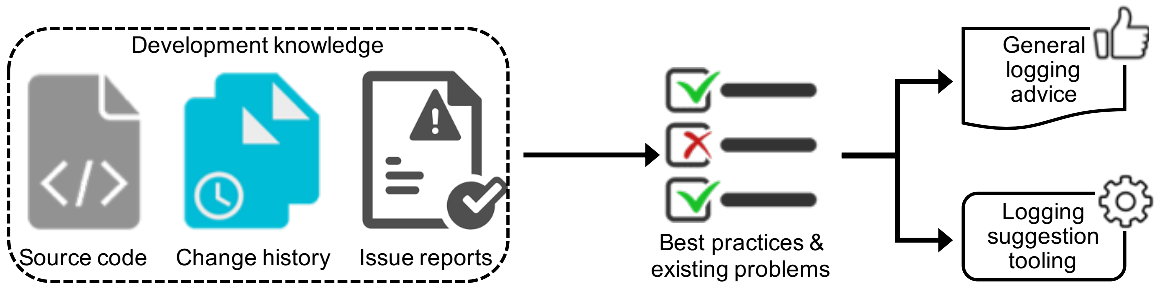


Figure 1.1: An overview of mining development knowledge to understand and support software logging practices.

developers' logging practices.

1.1 Research Hypothesis

This thesis aims to mine development knowledge to understand and support developers' logging practices. Development knowledge includes source code, change history (code changes and commit messages), as well as issue reports (issue descriptions, comments and patches). Development knowledge records developers' development activities and their intention behind these activities. This thesis proposes the following research hypothesis:

Research hypothesis: Development knowledge (e.g., source code, change history and issue reports) contains valuable information that can explain developers' rationale of logging, which can help us understand current logging practices and develop helpful tools to support such logging practices.

Figure 1.1 shows an overview of this thesis work. Through mining development knowledge, we find the best logging practices and the existing problems (e.g., excessive logging) within current logging practices. Based on these findings, we derive general logging advice and built automated tools to support current logging practices (i.e., to help developers solve existing logging problems). Specifically, this thesis mines three types of repositories: 1) mining issue reports (Chapter 3), 2) mining source code (Chapter 4 and Chapter 7), and 3) mining change history (Chapter 5 and Chapter 6)).

1.2 Thesis Overview

We now give a brief overview of the work presented in this thesis.

1.2.1 Chapter 2: Literature Review

For our literature review, we focus on prior studies that attempt to understand or improve software logging practices. We characterize and compare the surveyed literature along four categories:

- **Mining logging code.** Prior work empirically studies how developers insert logging statements in their source code and the evolution of their logging code.
- **Mining log messages.** Prior studies mine the rich source of log messages that are generated at run time, in order to support various software engineering purposes (e.g., anomaly detection).
- **Automatic log insertion.** Based on static code analysis and heuristics, prior studies automatically add log information in the source code, in order to support failure diagnosis.
- **Learning to log.** Prior studies learn statistical models or heuristics from existing logging code, in order to provide logging suggestions such as *where to log*.

From the literature review, we observe that software logging is a pervasive software engineering process. However, developers usually have difficulties when making their logging decisions, and they spend much effort maintaining their logging code. Prior studies propose automated approaches to help developers improve their logging code. Yet, these studies rarely take developers' expertise and concerns for logging into consideration. These studies provide logging suggestions as a post-development process instead of providing logging suggestions during the development process, i.e., when developers are changing their code.

1.2.2 Chapter 3: Understanding Developers' Logging Concerns

Modern software development processes use issue reports to manage development tasks (e.g., new features, feature enhancement, or bug fixes). Issue reports record the description, rationale, developers' comments, and related code changes of a development task. Mining logging-related issue reports helps us better understand developers' logging concerns, i.e., the benefits (e.g., assisting in debugging) and costs (e.g., performance overhead) of logging.

Therefore, we perform a qualitative study on the logging-related issue reports from three large and successful open source projects. We manually investigate these issue reports and derive high-level concepts about developers' logging concerns and how they address their logging concerns. Along with our qualitative study, we also summarize best logging practices and general logging advice that can help developers (and logging library providers) improve their logging code (and logging libraries).

1.2.3 Chapter 4: Understanding Software Logging Using Topic Models

Source code represents the resulting fact of developers' code change activities. Prior studies learn statistical models from existing source code and provide logging suggestions about *where to log*. These studies use the structural information of the source code (e.g., exception types or method calls) to explain the likelihood that a code snippet needs a logging statement. We believe that the semantic topics of a code snippet (e.g., "network connection") also help explain the likelihood of having a logging statement in the code snippet.

Therefore, we study the relationship between the topics of a code snippet and the likelihood of a code snippet being logged (i.e., to contain a logging statement). Through a case study on six open source systems, we find that there exists a small number of "log-intensive" topics that are more likely to be logged than other topics, and that our topic-based metrics help explain the likelihood of a code snippet being logged. As a result, leveraging both structural metrics and topic-based metrics can provide better suggestions about "where to log". Our findings highlight that topics contain valuable information that can help guide and drive developers' logging decisions.

1.2.4 Chapter 5: Automated Suggestions for Log Changes

Code change history records developers' code change activities that have been committed and the associated commit messages. Developers' logging activities (i.e., adding, deleting, or updating logging statements) are also recorded in the code change history. Such logging activities can help us better understand current software logging practices and the rationale behind these activities.

Therefore, we first empirically study why developers make log changes through a manual investigation of developers' logging activities. Based on our findings, we propose an automated approach to provide developers with log change suggestions as soon as they commit a code change. Through a case study on four open source projects, we find that the reasons for log changes can be grouped along four categories: block change, log improvement, dependence-driven change, and logging issue. We also find that our automated approach can effectively suggest whether a log change is needed for a code change.

1.2.5 Chapter 6: Automated Suggestions for Choosing Log Levels

Software practitioners use log levels to disable some verbose log messages while allowing the printing of other important ones. However, prior research observes that developers often have difficulties when determining the appropriate level for their logging statements. We analyze the development history of four open source projects to study how developers assign log levels to their logging statements. We also propose an automated approach to help developers determine the most appropriate log level when they add a new logging statement. Our automated approach can accurately suggest the levels of logging statements with an AUC (Area Under the Curve) of 0.75 to 0.81. We find that the characteristics of the containing block of a newly-added logging statement, the existing logging statements in the containing source code file, and the content of the newly-added logging statement play important roles in determining the appropriate log level for that logging statement.

1.2.6 Chapter 7: Automated Suggestions for Logging Stack Traces

Software developers log the stack traces of program exceptions for debugging purposes. However, stack traces can also pollute log files very fast because a stack trace is usually much longer than a regular log message. We observe that developers have difficulties to decide whether to log the stack trace of an exception. Therefore, we propose an automated approach to help developers make informed decisions about whether to print the stack trace of an exception in a logging statement. Our experimental results on four open source projects show that our automated approach can accurately suggest whether to print the stack trace of an exception in a logging statement, with an AUC of 0.85 to 0.94. Our findings also provide developers and researcher insights into the important factors that drive developers's decisions of logging exception stack traces.

1.3 Thesis Contributions

This thesis empirically studies developers' logging practices and proposes automated approaches to help developers make informed logging decisions. The results of the thesis highlight the importance of considering developers' expertise and concerns when providing automated approaches for logging improvement. The thesis mainly makes the following contributions:

- The discussion of developers' logging concerns (in Chapter 3) sheds light on future research opportunities for logging improvement (i.e., helping developers leverage the benefits of logging while minimizing logging costs).

- We show that the semantic topics of a code snippet can provide another dimension to explain the likelihood of logging a code snippet (in Chapter 4).
- We propose an automated approach to provide developers with log change suggestions as soon as they commit a code change (in Chapter 5).
- We propose an automated approach to help developers determine the most appropriate log levels when they add new logging statements in the source code (in Chapter 6).
- We propose an automated approach to help developers make informed decisions about whether to print an exception stack trace in a logging statement (in Chapter 7).

CHAPTER 2

Literature Review

My thesis aims to understand and support software logging practices through mining development knowledge. Understanding current logging practices is the first step towards helping developers improve their logging practices. Two categories of prior studies help software practitioners and researchers understand the current logging practices in industry and in the open source community: *mining logging code* and *mining log messages*. The former category studies how logging code are added into software products, while the latter category studies how runtime log messages are leveraged to support software engineering processes.

- **Mining logging code.** Prior work empirically studies how developers insert logging statements in their source code and the evolution of their logging code.

- **Mining log messages.** Prior studies mine the rich source of log messages that are generated at run time, in order to support various software engineering purposes, for example, anomaly detection or failure diagnosis.

Prior studies also aim to improve software logging through *automatic log insertion* or *learning to log*. Both categories of studies propose automatic approaches to identify the code snippets that need to be logged.

- **Automatic log insertion.** Based on static code analysis and heuristics, prior studies automatically add (or improve) logging statements in the source code, in order to support failure diagnosis.
- **Learning to log.** Prior studies learn statistical models or heuristics from existing logging code, in order to provide logging suggestions such as *where to log*.

This chapter first explains the literature selection process, then discuss the existing studies along the aforementioned four categories.

2.1 Literature selection

There has been a large number of prior studies that focused on log analysis, i.e., leveraging runtime logs for various domain-specific purposes (e.g., monitoring system performance). However, this literature review is focused on the papers that aim to understand and support software logging practices (i.e., the software engineering process that adds logging code to the source code). Therefore, this literature review starts from papers that are published on major software engineering journals and conferences.

This literature review starts from the venues that are listed in Table 2.1. This literature review considers the papers that were published in the past 10 years (i.e., from

Table 2.1: Names of conferences and journals as starting venues of the literature review.

Venue Type	Venue Name	Abbreviation
Journal	IEEE Transactions on Software Engineering	TSE
Journal	ACM Transactions on Software Engineering and Methodology	TOSEM
Journal	Empirical Software Engineering	EMSE
Journal	Automated Software Engineering	ASE
Journal	Journals of Systems and Software	JSS
Conference	ACM SIGSOFT Symposium on the Foundation of Software Engineering/ European Software Engineering Conference	FSE/ESEC
Conference	International Conference on Software Engineering	ICSE
Conference	International Conference on Automated Software Engineering	ASE
Conference	International Conference on Software Maintenance and Evolution	ICSME
Conference	International Conference on Software Analysis, Evolution, and Reengineering	SANER
Conference	International Conference on Mining Software Repositories	MSR
Conference	International Conference on Architectural Support for Programming Languages and Operating Systems	ASPLOS

2008 to 2018). To improve the coverage of this literature review, we also check the citations of each reviewed paper. Initially, all the reviewed paper fall into the categories of *mining logging code, automatic log insertion* and *learning to log*. We found that many papers about *mining log messages* are cited by the other three categories of papers, thus we included the *mining log messages* category in this literature review. We detail each category of papers below.

2.2 Mining logging code

Characterizing logging practices. Prior work performs empirical studies to characterize current logging practices. Yuan et al. (2012b) make the first attempt to provide a characteristic study of the current logging practices within four C/C++-based open-source projects. They quantitatively study the logging code and the change history of the logging code. They find that software logging is a pervasive practice in software development, and that developers spend much effort maintaining their logging code (e.g., modifying log levels).

Chen and Jiang (2017b) replicate the work of Yuan et al. (2012b) on 21 Java-based open source projects. They also find that logging is a pervasive software logging practice and that developers spend much effort on logging code maintenance. However, these two studies conflict with each other in some findings. For example, the former study (Yuan et al., 2012b) finds that developers spent shorter time fixing reported failures when log messages are presented in failure reports, while the latter study finds the opposite (Chen and Jiang, 2017b).

Chen and Jiang (2017a) also characterize the auti-patterns of logging code in open source projects by learning from how developers fix the defects in their logging code. They find six different anti-patterns in the logging code, such as wrong log level and logging nullable objects.

Shang et al. (2015) explore the relationship between logging characteristics and code quality. Surprisingly, their results show that logging characteristics provide a strong indicator of post-release defects. They explain that it might be the case that developers often relay their concerns about a piece of code through logging statement, thus source code files with high log density are more defect-prone.

Logging practices in industry. Software logging is a widely adopted practice in industry. Fu et al. (2014) conduct source code analysis on two industrial software projects at Microsoft, in order to find out what categories of code snippets are logged and what factors are considered for logging. They find five categories of logged code snippets (including return-value-check snippets and exception-catch snippets). In addition, they discuss the characteristics of the logged exception-catch snippets versus unlogged exception-catch snippets.

Pecchia et al. (2015) study the industrial logging practices at Selex ES. They find that logging is a widely adopted practice in a critical industrial domain. They also observe that the logging behavior is strongly developer dependent and development-team dependent. They highlight the need to establish standard company-wide logging policies.

Evolution of logging code. Prior research studies the evolution of logging code in software projects. They find that logging code changes over time at a high rate. Shang et al. (2011, 2014a) perform a case study on two open source and one industrial software projects, in order to explore the evolution of logging code in these projects. They find that the logging code changes at a high rate across versions, which might break the functionality of log processing applications. They also suggest that the majority of the logging code changes could be avoided.

Kabinna et al. (2016a,b, 2018) study the evolution of logging code in Apache Software Foundation (ASF) projects. They find that many ASF projects have undergone logging library migrations (Kabinna et al., 2016a). However, performance is rarely improved after a migration. They also find that a large amount of logging statements change throughout their lifetime, and they discuss the factors that impact the stability of a logging statement (Kabinna et al., 2016b, 2018).

Prior studies find that software logging is a pervasive practice in software development, and that developers spend much effort maintaining their logging code. Most empirical studies of software logging practices analyze the logging code and its evolution. They focus on developers' logging behaviors without exploring the rationale behind developers' logging decisions.

2.3 Mining log messages

Prior studies have proposed various approaches to mine log messages for different purposes.

Understanding system runtime behaviors. System logs are widely used for system operators to understand system behaviors. With the increasing scale and complexity of software systems, it has become challenging for system operators to manually analyze system logs. Fu et al. (2013) propose an approach to help operators understand system behaviors by mining execution patterns from system logs. An execution pattern is reflected by a sequence of system logs. Based on the mined execution patterns, their approach further learns essential contextual factors that cause a specific code path to be executed. Their approach helps system operators understand system runtime behaviors in various tasks (e.g., system problem diagnosis).

An operation profile captures the common usage scenarios (e.g., sending email) of a particular system (e.g., an email client) and their occurring rate. Hassan et al. (2008) propose an approach to customize operational profiles for large deployments. Their approach can uncover the most repetitive usage scenarios which are usually critical to system performance. They leverage a textual compression algorithm to compress different segments of log files. A high compression ratio indicates highly repetitive sequences of log messages and thus representing a common usage scenario.

Shang et al. (2013) mine log messages of big data analytics applications to reduce developers' effort to verify the deployment of such applications on Hadoop clouds. Their approach uncovers the differences between a pseudo-cloud deployment and a large-scale cloud deployment, and direct developers' attention to examining such differences, thereby reducing the deployment verification effort.

Anomaly detection. Log messages are widely used to monitor the health of software systems and identify abnormal conditions. Traditionally, software practitioners can search keywords such as "error" and "failure" to spot the failures of a system execution. Prior studies also propose more sophisticated approaches to detect more implicit failures. Xu et al. (2009b,a, 2008) propose a general methodology to mine the rich information in logs to detect system runtime problems. Based on the assumption that a problem manifests as a abnormality in the relationships among different types of log messages, their approaches extract features that capture various correlations among log messages (e.g., relative frequencies). Then, they use a Principal Component Analysis based anomaly detection method with the extracted features to identify runtime problems.

Fu et al. (2009), and Mariani and Pastore (2008) learn a Finite State Automaton (FSA) from training log sequences to represent the normal work flow for each system component. Then, the FSA can automatically detect anomalies in newly input log files.

Jiang et al. (2008) mine load testing logs to learn dominant behavior (i.e., execution sequences) and flag deviations (i.e., anomalies) from the dominant behavior, in order to detect problems in load testing tasks.

Failure diagnosis. Log messages are usually the most important clues for failure diagnosis and the only resource for diagnosing field failures. Yuan et al. (2010) propose *SherLog*, which leverages runtime log information and source code analysis to infer the execution paths (i.e., what must or may have happened) during a failed production run, in order to assist developers in failure diagnosis. They find that the information inferred by *SherLog* is very useful for developers to diagnose real world software failures.

Syer et al. (2013) leverage both performance counters and execution logs to diagnose memory-related performance issues. They combine the performance counters and execution events (i.e., abstracted execution logs) by discretizing them into time-slices. Then, they use statistical techniques to identify the set of execution events that are associated to a performance issue.

Nagaraj et al. (2012) propose *DISTALYZER*, which leverages the vast log data available from large scale systems to support developers in diagnosing performance problems. *DISTALYZER* uses machine learning techniques to compare logs with good performance and logs with bad performance, and automatically infer the strongest associations between system components and performance.

Quality log messages are critical for understanding system runtime behaviors, anomaly detection and failure diagnosis. The importance of logging quality motivates our study to understand current logging practices and assist software developers in making better logging code.

2.4 Automatic log insertion

Proactive logging. As the run-time log information is frequently insufficient for detailed failure diagnosis, prior studies propose approaches to automatically insert additional log information to the source code. *LogEnhancer* (Yuan et al., 2011) automatically adds variables into existing logging statements, in order to aid in the diagnosis of future failures. *LogEnhancer* conduct static analysis on the source code and automatically identify *causally-related* variables that if logged, can minimize the uncertainty of the execution paths during failure diagnosis.

Errlog (Yuan et al., 2012a) proactively adds appropriate logging statements into source code. *Errlog* analyzes real-world failures and derives common error sites (e.g., system call return errors, exceptions), then automatically inserts missing logging statements into such error sites. Both *LogEnhancer* and *Errlog* are reported to significantly reduce failure diagnosis time.

Zhao et al. (2017b,a) propose *Log20*, which automatically place logging statements under a specified threshold of performance overhead. *Log20* measures how effective each logging statement is in disambiguating code paths, and automatically place logging statements so that they can minimize code path ambiguity while satisfying the given threshold of performance overhead.

Interactive logging. Prior research also explores interactive logging, i.e., inserting logging code when it is needed. *AutoLog* (Zhang et al., 2011) generates additional informative logs to help developers discover the root causes when there is a failure. When developers need more clues to diagnose a system failure, *AutoLog* performs program slicing to find the execution paths that might lead to the failure, and incrementally add logging statements along the execution paths, with the goal to approach the root cause quickly with fewer logs, and then execute the program again to generate more logs. The process ends when developers have enough clues to find the root cause of the failure.

Automatic log insertion tools are helpful when developers need to collect more clues for failure diagnosis. However, these tools do not consider developers' expertise and concerns for logging. Therefore, the logging statements that are generated by these tools are difficult to be integrated into production software.

2.5 Learning to Log

Learning statistical models. Prior studies learn statistical models from common logging practices and leverage the models to provide logging suggestions. Zhu et al. (2015) propose *LogAdvisor*, which automatically learn *where to log* from existing logging code and provide informative logging guidance to developers. *LogAdvisor* extracts contextual features of a code snippet (*exception-catch* snippet or *return-value-check* snippet), then learns statistical models to suggest whether a logging statement should be added to such a code snippet. *LogAdvisor* is the first step towards "learning to log". Similarly, *LogOpt* (Lal and Sureka, 2016) extract contextual features from the source code and build statistical models to predict whether a logging statement is needed in an *exception-catch* block.

Jia et al. (2018) propose an intention-aware log automation tool called *SmartLog*, which uses an Intention Description Model (IDM) to explore the intension of existing logs and mine log rules from such intentions. *SmartLog* only focuses on logging placement of *function call* code snippets.

Learning logging heuristics. Prior research also learns logging heuristics from experiences. King et al. (2015) propose empirical heuristics to help developers identify mandatory log events (i.e., defined as actions that must be logged to hold the software user accountable for performing the actions). They extract 3,513 verb-object pairs from natural language software artifacts, and manually classify each verb-object pair as either a mandatory log event or not. Then, then use grounded theory analysis to derive 12 heuristics that can help determine whether a verb-object pair describes a mandatory log event or not.

King et al. (2017) also perform a controlled experiment to evaluate whether the derived heuristics can effectively help developers identify mandatory log events. However, the heuristics do not help developers more correctly identify mandatory log events at a statistically significant level.

Prior studies learn common logging practices from the source code; they never consider the code change history nor the issue reports that provide additional dimensions to explain developers' logging practices. Prior studies provide logging suggestions as a post-development process instead of providing logging suggestions during the development process, i.e., when developers are changing their code.

CHAPTER 3

Understanding Developers' Logging Concerns

As we observed in Chapter 2, prior studies aimed to improve logging either by proactively inserting logging statements in certain code snippets or by learning where to log from existing logging code. However, there exists no work that studies developers' logging concerns, i.e., the benefits and costs of logging from developers' perspective. Without understanding developers' logging concerns, automated approaches for logging improvement are based primarily on researchers' intuition and unconvincing to developers. In order to fill this gap, we performed a qualitative study on 533 logging-related issue reports from three large and successful open source projects. We manually investigated these issue reports and derived high-level concepts about developers' logging concerns and how they address their logging concerns. Along with our qualitative analysis, we also summarized best logging practices and general logging advice that can help developers (and logging library providers) improve their logging code (and logging libraries). Our empirical findings also shed lights on future research opportunities for helping developers leverage the benefits of logging while minimizing logging costs.

An earlier version of this chapter is under review at the IEEE Transactions on Software Engineering Journal (TSE).

3.1 Introduction

SOFTWARE developers leverage logging statements in the source code to generate runtime log messages which are crucial for understanding system runtime behaviors and diagnosing runtime issues. Missing an important piece of logging information can increase the difficulty of diagnosing a field failure (Yuan et al., 2011, 2012a). For example, issue report HADOOP-13458[1] complains that it was hard to comprehend the meaning of an exception message without logging the stack trace. On the other hand, adding logging statements excessively is not an optimal solution, since adding too much logging can significantly increase system overhead (Zeng et al., 2015; Fu et al., 2014). For example, issue report HADOOP-12903 complains that too much logging slowed down the speed of servers.

Prior studies have proposed automated approaches to improve logging through *proactive logging* (Yuan et al., 2011, 2012a) and *learning to log* (Zhu et al., 2015; Lal and Sureka, 2016; Jia et al., 2018). *Proactive logging* approaches use static analysis to automatically add more logged information to the existing code, in order to improve software failure diagnosis. *Learning to log* approaches, on the other hand, learn statistical models from existing logging practices and further leverage the models to suggest *where to log*. All these existing studies aim to ease or improve developers' logging practices. Their goal is to help developers address their logging concerns, i.e., by leveraging the benefits of logging while minimizing its costs.

However, there exists no work that studies developers' logging concerns, i.e., the

[1]All the issue reports mentioned in this paper can be accessed though the URL https://issues.apache.org/jira/browse/<ISSUE-ID>.

benefits and costs of logging, from the perspectives of developers. Without a clear understanding of developers' logging concerns, automated approaches for logging improvement are not convincing to developers. On the other hand, developers are also not fully aware about the benefits and costs of logging , and in some cases raise conflicting concerns regarding some logging issues (see RQ1 - Discussion). Therefore, this chapter aims to understand developers' logging concerns, and how they address their logging concerns.

Developers communicate their logging concerns in their logging-related issue reports. For example, issue report HADOOP-13693 raises a concern that logging an error message for a successful operation is confusing and misleading. Therefore, we performed a qualitative study on 533 logging-related issue reports from three large and successful open source projects. We used a manual coding approach to derive high-level concepts from these issue reports, in order to understand developers' logging concerns and how they address their logging concerns. Along with our qualitative analysis, we also derived best logging practices and general logging advice which can be leveraged by software practitioners in their logging practices. In particular, we address the following two research questions (RQs).

RQ1: What are developers' logging concerns?

Our goal is not to find the direct causes of logging issues (e.g., incorrect log levels), but rather to go deeper and understand why developers raise such logging issues, i.e., what are the benefits of logging that they want to leverage, and what are the costs of logging that they want to avoid.

RQ2: How do developers address their logging concerns?

> Developers address their logging concerns by fixing logging-related issue reports. By examining developers' issue-fixing processes, we want to understand how they balance the benefits and costs of logging (e.g., by adjusting log level).

By learning from developers' logging concerns, we can provide developers with insight on how to leverage the benefits of logging while avoiding too much negative impact. We can also provide suggestions for logging library providers to improve their logging libraries (e.g., to support different log levels for different parts of a logging statement). Finally, our empirical findings inspire new research opportunities for logging improvement (e.g., developing methods and tools to leverage logging benefits and minimize logging costs).

Chapter organization. The remainder of the chapter is organized as follows. Section 3.2 describes our case study setup, covering our subject projects, data preparation and data analysis approaches. Section 3.3 presents the experimental results for answering our research questions. Section 3.4 discusses threats to the validity of our findings. Finally, Section 3.5 draws conclusions and outlines future research opportunities that are inspired by our study.

3.2 Case Study Setup

This section describes our case study subjects, the process that we used to prepare the data for our case study, and the qualitative analysis approaches that we used to study developers' logging concerns.

Table 3.1: Overview of the studied projects.

Project	**Studied History**	**SLOC**[*]	**Primary Language (SLOC)**	**# Logging Statements** [*]
Hadoop Common	2012.06 - 2017.06	313 K	Java (226 K)	638
Hive	2012.06 - 2017.06	1,445 K	Java (1,081 K)	5,503
Kafka	2012.06 - 2017.06	238 K	Java (149 K)	853

[*] The SLOC and the number of logging statements are calculated at the end of the studied period, i.e., June 30th, 2017. The number of logging statements for each project is calculated for the primary language.

3.2.1 Subject Projects

In order to study developers' logging concerns, we manually investigated the logging-related issue reports from three large and successful open source software projects, namely *Hadoop Common*[2], *Hive*[3], and *Kafka*[4]. *Hadoop Common* implements the common utilities for Hadoop, a distributed computing platform. *Hive* is a data warehouse that supports accessing big data sets residing in distributed storage using SQL. *Kafka* is a streaming platform for messaging, storing and processing real-time records. All of these projects are widely used by today's tech giants, such as Google, Amazon, Facebook, etc. We select these three subject projects because their logging code is well maintained, for example, they have many logging-related issue reports that are dedicated for maintaining logging code. As the log messages that are generated by these projects are exposed to the aforementioned tech giants as well as a much wider audience, the quality of their logging code is critical to their success.

Table 3.1 shows the overall information of the studied projects. *Hadoop Common* has 313 K source lines of code (SLOC), and it is primarily implemented in Java. *Hive* has

[2]`http://hadoop.apache.org`
[3]`https://hive.apache.org`
[4]`https://kafka.apache.org`

```
project in ("Project Name") AND summary ~ "(log || logger || print) NOT
    (\"log in\" || \"log out\" || \"blue print\" || \"print command\"~10)"
    ORDER BY created DESC
```

Figure 3.1: The JQL query that we used to search for the logging-related issue reports.

an SLOC of 1,445 K, and its dominant programming language is also Java. *Kafka* has an SLOC value of 238 K, and it is mostly implemented in Java. *Hive* has the largest number (i.e., 5,503) of logging statements, while *Hadoop Common* has the least number (i.e., 638) of logging statements. We study the logging-related issue reports that were created from June 2012 to June 2017. We extracted the logging issue data in December 2017 (at least six months after the creation of any studied issue report), to ensure that the status of the studied logging issues are relatively stable after a long time since their creation.

3.2.2 Data Preparation

We extract our logging issues from the Apache JIRA issue tracking system[5]. Figure 3.2 demonstrates our data extraction process. First, we use the JIRA Query Language (JQL) to automatically search for the JIRA issues reports that are related to logging (i.e., issue reports with logging-related keywords in their summaries). We use the JQL query in Figure 3.1 to search for the logging issue reports of each of the studied projects. The "Project Name" is replaced by "Hadoop Common", "Hive", and "Kafka" for our respective projects. This JQL query searches for all the issue reports of the specified project that have "log", "logger" , "print" , or their variations (e.g., "logging"), but don't have "log in", "log out", "blue print", or "print" and "command" together, in its summary, sorted by their creation time using a reverse-chronological order.

[5]https://issues.apache.org/jira

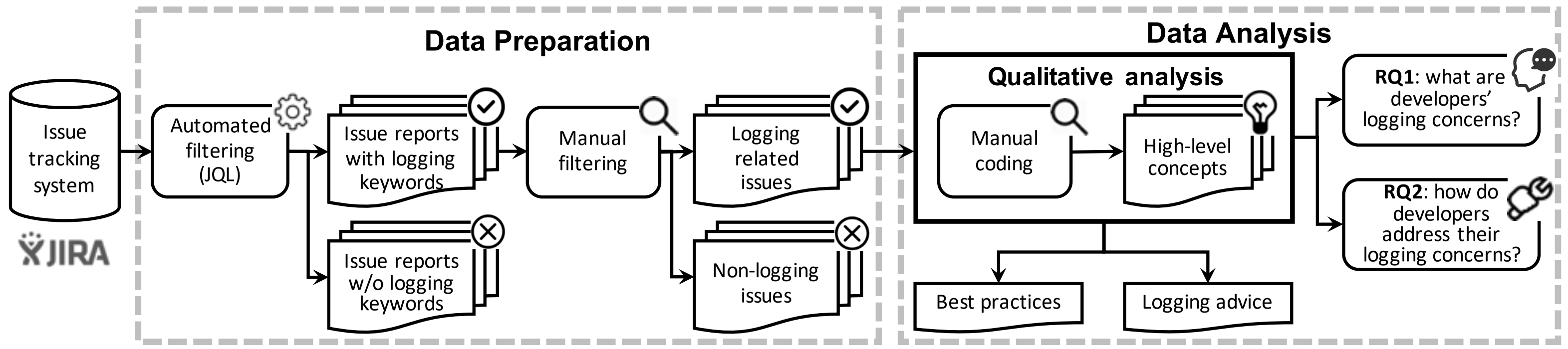

Data Preparation
Issue tracking system
JIRA
Automated filtering (JQL)
Issue reports with logging keywords
Issue reports w/o logging keywords
Manual filtering
Logging related issues
Non-logging issues
Data Analysis
Qualitative analysis
Manual coding
High-level concepts
Best practices
Logging advice
RQ1: what are developers' logging concerns?
RQ2: how do developers address their logging concerns?

Figure 3.2: The process of preparing and analyzing logging issue data.

The resulting issue reports from the automated filtering process may falsely include some non-logging issue reports. For example, issue report HADOOP-14060 has "log" in its summary but it is about the access control for the "logs" folder instead of a logging issue. In order to remove these non-logging issue reports, we manually examined all the resulting issue reports from the automated filtering process. For each issue report, we first checked its summary to determine if it is a logging issue. If we could not decide it from the summary, we further checked the description of the issue report. We only kept those issue reports that dealt with logging issues. We also removed duplicated logging issue reports and kept only one issue report for each duplication. We ended up with 533 logging-related issue reports.

Table 3.2 shows the number of issue reports that are resulted from the automated filtering process and the number of remaining issue reports after the manual filtering process (i.e., the number of logging issue reports that are studied in the rest of the chapter). Using our query criterion (i.e., Figure 3.1), we get 193, 395 and 314 issue reports of *Hadoop Common*, *Hive* and *Kafka*, respectively. 74% and 68% of the JQL-queried issue reports are concerned with logging for the *Hadoop Common* and *Hive* projects, respectively. However, only 39% of the JQL-queried issue reports are concerned with logging for the *Kafka* project. As the *Kafka* project deals with messaging, storing and processing of log messages, it has a large number of issue reports with the keyword "log" (or its variations) in their summaries but actually they are not necessarily concerned with the logging aspect of the project.

Table 3.2: Number of studied logging issue reports per project.

Project	**# JQL-queried issues**	**# Logging issues**
Hadoop Common	193	143 (74%)
Hive	395	268 (68%)
Kafka	314	122 (39%)
Total	902	533 (59%)

3.2.3 Data Analysis

Figure 3.2 also shows our data analysis process. In order to understand developers' logging concerns (RQ1) and how they address their logging concerns (RQ2), we performed a qualitative analysis on the 533 logging issue reports. We use a manual coding approach (see details below) to extract high-level concepts (e.g., developers' logging concerns) from the detailed information of these issue reports (e.g., summaries, descriptions, and comments). Inspired by our understanding of developers' logging concerns, we further derived some best logging practices and general logging advice that can help developers improve their logging code.

Our manual coding approach. Our manual coding approach follows an open card sorting approach (Spencer, 2009; Rugg and McGeorge, 2005; Zimmermann, 2016), except that we did not print our content (i.e., issue reports) on physical cards. The reason that we did not print our content on physical cards is that, for each issue report, we need to investigate the summary, description, comments, patches, code review comments, and commit messages, which cannot fit in a small card. Two researchers including the author of this thesis and a collaborator jointly performed our manual coding process. We first examined 50 issue reports together and jointly assign codes to these issue reports. For each examined issue report, we compared the new issue report with existing codes; if we could not find an appropriate existing code for the new issue report, we created a new code and assign the issue report with the new code. Then, we divided up the remaining issue reports and individually assigned codes to these issue reports. We kept communicating during the entire coding process. We discussed whenever someone is not certain about which code an issue report can fit, and we informed the other coder when a new code is created. We group lower-level codes into higher-level codes when it is appropriate. We also constantly made changes to our existing coding results whenever appropriate. Another two collaborators reviewed our coding results and suggested appropriate changes.

3.3 Case Study Results

In this section, we present the results for answering our research questions. For each research question, we discuss our motivation, our approaches, and the detailed experimental results.

3.3.1 RQ1: What are developers' logging concerns?

Motivation

Prior studies proposed automated approaches (e.g., statistical models) to help developers improve their logging practices. Without a clear understanding of developers' logging concerns, automated approaches for logging improvement may not meet developers' real needs. In order to fill the gap, we qualitatively examined 533 issue reports from three large and successful open source projects, to better understand the logging concerns from the perspective of developers. We are not trying to understand the direct causes of these issue reports (e.g., incorrect log levels), but rather to go deeper and understand why developers raise such logging issues, i.e., what are the benefits of logging that they want to leverage, and what are the costs of logging that they want to avoid. Software practitioners can learn from our findings to better understand the benefits and costs of logging and improve their logging code. Our findings also shed light on future research opportunities for logging improvement (e.g., helping developers minimize the costs of logging).

Approach

In order to analyze developers' logging concerns, we qualitatively examine the issue summaries, issue descriptions, issue comments, patches, commit messages and code review comments that are associated with and directly accessible from the studied logging issue reports. We use the manual coding approach discussed in Section 3.2.3 to derive high-level concepts about developers' logging concerns from the detailed information of these issue reports. Along with our qualitative analysis of developers' logging concerns, we also derived best practices and general advice for logging, based on the fact that developers raise similar logging issues across different issue reports and different projects.

A few (17) logging issue reports combine several logging issues together in one issue report (e.g., HIVE-12713, "Miscellaneous improvements in driver compile and execute logging"). We treat each of such issue reports as multiple logging issues and analyze each logging issue separately. We end up with 560 logging issues that are raised in 533 logging issue reports.

Some issue reports show different (and even opposite) concerns regarding a logging issue. For example, issue report HADOOP-11180 raises a concern that printing out too many "warn" messages for a successful execution can mislead end users, so the developer proposes to downgrade the "warn" messages to "debug" messages. However, another developer raises another concern that downgrading the "warn" messages can hide fundamental software problems. In such cases, we only consider a primary logging concern for each logging issue. For example, the primary logging concern for the example issue report (HADOOP-11180) is "misleading end users". In some cases, we could not understand developers' logging concerns from the issue reports. For example, issue report HADOOP-14296 proposes to migrate logging APIs to SLF4J[6] but never explains the rationale for doing so. We assign an "unknown" label for such cases.

Results

Half of the logging issues are concerned with the benefits of logging, while the other half are concerned with the costs of logging. Figure 3.3 summarizes the high-level categories of logging concerns that are derived from our qualitative analysis. Developers' logging concerns are grouped into ten high-level categories, among which, five of them are concerned with logging benefits (i.e., *assisting in debugging, exposing runtime problems, bookkeeping, showing execution progress,* and *providing runtime performance*), and the other five are concerned with logging costs (i.e., *excessive log information, misleading end users, performance overhead, exposing sensitive information,* and *exposing unnecessary details*). In total, we investigated 560 logging issues from the three studied projects (some issue reports discuss multiple logging issues). We could

[6]https://www.slf4j.org

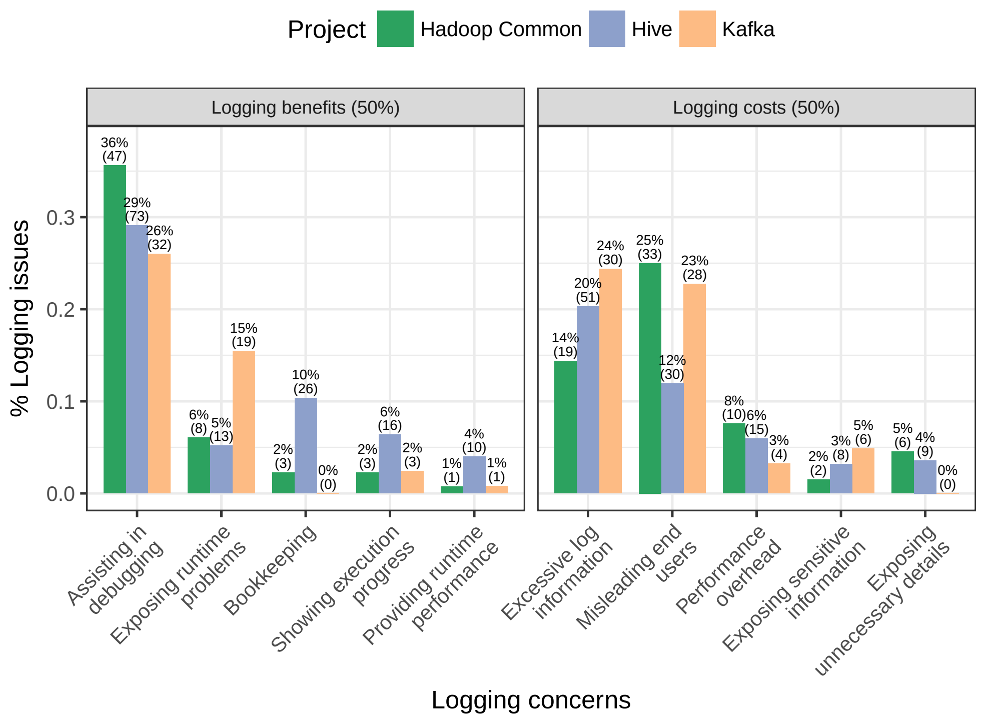


Figure 3.3: Developers' logging concerns and the percentage of logging issues in each project that raise each logging concern.

not conceptualize the logging concerns of 54 (~10%) logging issues (i.e., marked as "unknown"), because these issue reports never explain the rationale for changing the logging code. Among the logging issues for which we are able to conceptualize logging concerns, 255 (~50%) of them are concerned with the benefits of logging, and 251 (~50%) of them are concerned with the costs of logging. Detailed discussions about each of these logging concerns are as follows.

BENEFIT 1: Assisting in debugging. The most commonly concerned benefit of logging is *assisting in debugging*. In the studied projects, 26% to 36% of the studied logging issues raise the logging benefit of *assisting in debugging* of errors that have been identified by developers or end users. Log messages help developers narrow down the

```
try {
  Thread.sleep(retryInfo.delay);
} catch (InterruptedException e) {
  /* Logging the stack trace by specifying the exception (e) as the last
      parameter of a logging statement. */
  LOG.debug("Interrupted while waiting to retry", e);
}
```

Figure 3.4: Example of logging for assisting in debugging.

execution paths of a process and find the root cause of an execution failure (Yuan et al., 2010). For example, issue report HADOOP-14497 requests to log the lifecycle (i.e., creation, renewal, cancellation, and expiration) of delegation tokens, in order to identify the root causes of authentication failures related to delegation tokens. In particular, logging specific information instead of general information (e.g., HIVE-11163), logging the causes of an error in addition to the error itself (e.g., KAFKA-4164), and logging the stack trace of an unexpected exception (e.g., HADOOP-13682) in addition to the exception message can effectively help developers narrow down the root causes. Modern logging libraries (e.g., Log4j[7] and SLF4J[6]) usually support convenient ways to log the stack trace of an exception, e.g., Figure 3.4 shows a code example of logging the stack trace of an exception for assisting in debugging. Developers also recommend to log context information (e.g., thread id, session id, query id, user id, etc.) in a multi-task program (e.g., HIVE-13517, HIVE-11488, KAFKA-3816, HIVE-15631, HIVE-6876).

PRACTICE 1: Log specific information (e.g., detailed error spots) instead of general information and the context information (e.g., thread ids) of an event to support better failure diagnosis.

BENEFIT 2: Exposing runtime problems. The second mostly concerned benefit of

[7]https://logging.apache.org/log4j/2.x/

```
catch (IOException ex) {
  LOG.warn("Failed to connect to {}:{}", url.getHost(), url.getPort());
}
```

Figure 3.5: Example of logging for exposing runtime problems.

logging is *exposing runtime problems*, which can be used to triage, understand and prioritize runtime problems. In the *Kafka* project, in particular, 15% of the studied logging issues raise the logging benefit of *exposing runtime problems*. Log messages can help developers and users *identify* the problems or anomalies in a system execution. For example, issue report HADOOP-12901 requests to log a "warn" message when a client fails to connect to a server, so that users can easily identify the connection problem and fix it. Figure 3.5 shows a code example of logging for exposing runtime problems. In particular, developers need to log unhandled exceptions, otherwise there is nothing to indicate such exceptions (e.g., HADOOP-12749). Anomaly detection tools (Jiang et al., 2008; Fu et al., 2009; Xu et al., 2009b) automatically analyze large amounts of log messages (that are hard for humanbeings to investigate manually) and alert anomalies that are indicated in the log messages. Missing such alerting log messages can make it hard to identify runtime problems at an early stage and bring difficulties for locating the problems (e.g., HADOOP-11328). These alerting logs are usually logged at the "warn" or "error" levels instead of lower log levels. For example, issue reports HADOOP-12887 requests to change the log level of a logging statement from "info" to "warn" so that one can easily identify a configuration error (e.g., by searching the keyword "warn"). However, logging normal events or properly handled problems at the "warn" or "error" levels can spam the log files with "warn" or "error" messages and make it difficult to identify real problems (e.g., HIVE-8382).

PRACTICE 2: Log real problems (e.g., unhandled exceptions) at the "warn" or "error" levels and properly handled problems at lower levels to help in uncovering runtime problems.

BENEFIT 3: Bookkeeping. Developers can use log messages to record (i.e., bookkeep) important transactions or operations in a system execution, such as user login/logout, database operations, remote queries and requests. Such bookkeeping log information can be later processed for various analysis activities, such as security analysis (Oliner et al., 2012), performance analysis (Syer et al., 2013), and capacity planning (Kavulya et al., 2010). In the *Hive* project, 10% of the studied logging issues raise the logging benefit of bookkeeping (e.g., bookkeeping database queries). The Sarbanes-Oxley Act of 2002 (Sarbanes, 2002) requires all telecommunication and financial applications to log some mandatory log events, such as user activities, network activities and database activities[8]. The logging of the traceable information of a transaction or operation, such as the hostname (e.g., HIVE-12235) and client IP (e.g., HIVE-3512) of a query, and the identities of the operated objects (e.g., the ids of the cancelled queries, HIVE-16286; the names of the created/deleted directories, HIVE-13058; the ids of the opened/closed sessions, HIVE-14209), is usually required. Figure 3.6 shows a code example of logging for bookkeeping. Developers also suggest that bookkeeping logs need to be symmetric. For example, when the creation a certain object (e.g., a table) is logged, the deletion of the object should also be logged. Otherwise one could not confirm if the object still exists (e.g., HIVE-13058).

PRACTICE 3: Log traceable information (e.g., object identities or client IPs) when bookkeeping transactions or operations.

[8]https://sarbanes-oxley-101.com/sarbanes-oxley-audits.htm

```
hdfsSessionPath.getFileSystem(conf).delete( hdfsSessionPath, true);
LOG.info("Deleted HDFS directory: " + hdfsSessionPath);
```

Figure 3.6: Example of logging for bookkeeping.

```
LOG.debug("Waiting to acquire compile lock: " + command);
compileLock.lock();
LOG.debug("Acquired the compile lock");
```

Figure 3.7: Example of logging for showing execution progress.

BENEFIT 4: Showing execution progress. Log messages help in tracking the status or progress of an execution, such as the start or end of an event (e.g., HIVE-11314), a status change (e.g., flip over a flag, HADOOP-10046), an ongoing action (e.g., retrying, HADOOP-10657), or the status of waiting for some resources (e.g., waiting for a lock, HIVE-14263). Figure 3.7 shows a code example of logging for showing execution progress. While bookkeeping logging supports post-execution analysis of important transactions and operations, progress logging can help in determining whether a system is progressing as expected or something is going wrong. In particular, printing the progress information for a process that takes a long time is important for figuring out what's going on in the process. For example, issue report KAFKA-5000 requests regular progress information to be logged for a long process so that one can know whether the process is progressing or stuck. Concerns are often raised about the symmetry of progress logging. For example, "waiting for lock" should be followed by "lock acquired" (HIVE-14263), while "start of process" should be followed by "end of process" (HIVE-12787).

```
perfLogger.PerfLogBegin(CLASS_NAME, method.getName());
doSomething();
perfLogger.PerfLogEnd(CLASS_NAME, method.getName());
```

Figure 3.8: Example of logging for providing runtime performance.

PRACTICE 4: Logging needs to be symmetric, e.g., logging both the creation and deletion of an object, or logging both the start and end of a process. Future research is needed to help developers make symmetric logging.

BENEFIT 5: Providing runtime performance. Logging statements are used to record the performance information of a system at runtime (*a.k.a.*, performance logging). Such performance information is usually related to the execution time or memory usage of a process. For example, issue report HIVE-14922 requests the logging of the time spent in several performance-critical tasks, and issue report KAFKA-4044 requests the logging of the actually used buffer size. Figure 3.8 shows a code example of logging for providing runtime performance. Such performance information helps in understanding system health (e.g., HIVE-8210), in tuning system performance (e.g., HADOOP-13301), and in adjusting resource allocation (e.g., KAFKA-4044). It is suggested to log such performance information using standardized perf loggers to separate performance logging from event logging, for better performance analysis (e.g., HIVE-11891).

PRACTICE 5: Logging the performance information of critical tasks can help developers understand system health, tune system performance, and allocate system resources. Standardized perf loggers are preferred over using event logging libraries.

COST 1: Excessive log information. The most frequently raised logging cost is *excessive log information*. In the studied projects, 14% to 24% of the studied logging issues raise the concern of *excessive log information*. Excessive log information is usually caused by repeated log messages for a single event type (i.e., log messages produced by a single logging statement), such as logging database operations on every row of a table (e.g., HIVE-8153), logging every entry (e.g., KAFKA-3792), logging every request (e.g., KAFKA-3737), or logging every user (e.g., HADOOP-12450). Such excessive log information can mask other important information and lead to expensive storage costs (Fu et al., 2014). In particular, repetitive logging of stack traces usually grows the log files very fast and frustrates the end users. For example, issue report HADOOP-11868 raises a major concern about the excessive logging of stack traces for invalid user logins. It is advisable to aggregate such highly repetitive log lines, for example, by logging aggregated information at a higher log level and detailed information at a lower log level (e.g., HIVE-10214, KAFKA-4829).

Many issue reports suggest, for a single event, to log the normal log text at a higher level (e.g., "error") and the stack trace at a lower level (e.g., "debug"), such that the stack traces are hidden in normal cases and are only printed out when needed (e.g., HADOOP-13669). It is also suggested to log the important information of an event at a higher level, while logging the detailed information of the same event at a lower level (e.g., KAFKA-1199). Therefore, there is a strong need for supporting logging different parts (in particular, stack traces) of a logging statement at different log levels, which is not supported by modern logging libraries. As a workaround, developers usually need to insert two separate logging statements at different log levels for a single event (e.g., HADOOP-11868). Developers also need to be cautious when throwing and logging an exception at the same time[9]. Because it may lead to duplicated logging as the handler of the exception may log the exception again (e.g., KAFKA-1591).

ADVICE 1: Repetitive log messages for a single event type can cause excessive log information, which is suggested to be aggregated.

ADVICE 2: Logging libraries should consider supporting different log levels for different parts (e.g., the error message vs the stack trace) of a logging statement.

ADVICE 3: Developers need to be cautious when logging and throwing an exception at the same time, since the handler of the exception may also log the exception.

COST 2: Misleading end users. The second most frequently raised logging cost is *misleading end users*. In the studied projects, 12% to 25% of the studied logging issues are concerned with *misleading end users*. As log messages are directly exposed to end users, inappropriate log information can be confusing and misleading. In particular, logging "warn" or "error" messages for successful operations is the most frequent cause for this concern. For example, HADOOP-13693 complains that a warning in a successful operation confuses end users. Even worse, sometimes inappropriate log messages can annoy or frustrate end users. For example, HADOOP-13552 complains that there are too many "scary looking" stack traces being printed out in the log files, but in fact those exceptions can be handled automatically. Such large amount of repetitive stack traces can frustrate (e.g., HIVE-11062) or annoy (e.g., HIVE-7737) end users.

ADVICE 4: Developers should avoid logging successful operations at the "warn" or "error" levels. In particular, logging the stack traces of properly handled exceptions can unnecessarily alarm end users.

[9]https://www.loggly.com/blog/logging-exceptions-in-java/

COST 3: Performance overhead. Performance overhead is considered as a major cost of logging (Fu et al., 2014; Zhu et al., 2015), as printing a log message into a log file involves expensive IO operations, string concatenations, and possible method invocations for producing the log strings. 3% to 8% of the studied logging issues in the studied projects raise performance concerns. One cause of performance overhead is overwhelmingly repetitive printing of similar log messages. For example, issue report HIVE-12312 complains that the compilation of a complex query is significantly slowed down as a code snippet with a logging statement is called for many thousands of times. The execution process was speeded up by 20% after disabling the logging statement. Another cause of performance overhead is the invocation of expensive methods in logging statements. For example, issue report HADOOP-14369 complains that including some method calls in logging statements is "pretty expensive". Surprisingly, even disabled lower level logging can cause serious performance overhead, because the parameters of a logging statement are evaluated before the check for the log level. For example, issue KAFKA-2992 reports that "trace" logs in tight loops cause significant performance issues even when "trace" logs are not enabled.

ADVICE 5: Logging overhead might exist even when logging is disabled. Developers should minimize logging in tight loops and avoid expensive method invocations in logging. Future research is needed to detect performance-critical logging.

COST 4: Exposing sensitive information. Sensitive information (e.g., usernames and passwords) should not be printed in log files. Once such sensitive information is logged, it might be archived for years and cannot be tampered with due to legal regulations. However, sometimes such sensitive information might end up logged by mistake. For example, issue reports HIVE-14098 complains that users' passwords are logged in clear text, which is undesirable. In particular, developers have difficulties to avoid logging sensitive information that is contained in an URL (e.g., URL containing usernames and passwords, HIVE-13091) or a user configuration field (e.g., cloud storage keys, HADOOP-13494). Developers may log the content of a URL or a configuration field without noticing the contained sensitive information. In an even more difficult situation, users may put their sensitive information in an unknown configuration field (e.g., caused by typo). In such a case, developers are likely to log the unknown configuration field (i.e., to alert the unknown configuration) and unintentionally expose users' sensitive information (e.g., KAFKA-4056).

ADVICE 6: Developers should not log users' sensitive information. In particular, they need to pay extra attention when logging URLs, configuration fields, or other user inputs. Future research and tooling support is needed to help in preventing logging sensitive information.

COST 5: Exposing unnecessary details. While *excessive log information* is concerned with the overall amount of log messages, and *exposing sensitive information* is concerned with divulging users' sensitive data, *exposing unnecessary details* is concerned with the exposure of the inner structures (e.g., library dependencies) and processes (e.g., algorithms) of a software system. Such inner structures and processes may be used by developers for their debugging purposes. However, sometimes developers log such inner information at a higher (more user-facing) log level (e.g., "info") and unnecessarily expose the inner information to end users. For example, issue report HADOOP-13550 complains that the internal information about disabling threads when thread count is zero should not be logged at a high level such as "warn". Another issue report, HIVE-7737, argues that printing the whole stack trace for "table not found" is unnecessary and misleading because "table not found" is usually caused by user errors. Developers should avoid exposing such unnecessary and misleading details of their software systems to end users.

ADVICE 7: Printing detailed structure or algorithm information to end users is unnecessary and misleading. Such detailed information should be avoided or logged at lower levels.

Discussion

Conflicting concerns about log levels. Developers usually have a hard time deciding the appropriate log levels for their logging statements (Yuan et al., 2012b; Oliner et al., 2012). In many cases they have conflicting concerns about a log level. For example, in issue report HADOOP-11180, a developer proposes to change a logging statement from "warn" to "debug", as "there are too many such warnings" (i.e., *excessive log information*) and it can *mislead end users*. However, another developer raises a conflicting concern and argues that "downgrading the logs is going to hide fundamental problems" (i.e., not able to *expose runtime problems*). Developers are mostly confused about the "debug" and "info" levels (Li et al., 2017a). In some cases, developers tend to print detailed debugging information at the "info" level (e.g., HIVE-16629). In other cases, "critical piece of information" was logged at the "debug" level (e.g., HADOOP-12789). Therefore, we developed an automated approach to help developers choose the most appropriate log level when they add a logging statement to their source code (Li et al., 2017a; Chapter 6).

Logging the stack trace of an exception? Logging exceptions is considered a good logging practice (Yuan et al., 2012a; Li et al., 2017b). In particular, logging the stack traces of exceptions is very helpful for debugging the exceptions (i.e., *assisting in debugging*).

However, as a stack trace is usually much longer than a log message, logging the stack traces can usually cause *excessive log information* and *performance overhead*. Logging unnecessary stack traces to the end users can also *expose unnecessary details* of a software system and *mislead end users* (e.g., "scary looking" stack traces for successful processes, as reported in HADOOP-13552). Developers usually have difficulties to balance the benefits and costs of logging stack traces. In fact, they raise conflicting concerns about adding stack traces to exception logging. For example, issue report HADOOP-10571 proposes to add stack traces to many exception logging statements across modules. However, other developers raise concerns that stack traces should be avoided for some of these exception logging statements. As a result, it takes significant efforts (e.g., as much as 10 patches) to resolve the conflicting concerns. Future research and tooling support is needed to help developers make informed decisions about whether to log the stack trace of an exception.

Developers leverage five categories of logging benefits, including *assisting in debugging, exposing runtime problems, bookkeeping, showing execution progress*, and *providing runtime performance*. However, developers are also concerned about five categories of logging costs, including *excessive log information* (size), *misleading end users* (accuracy), *performance overhead* (performance), *exposing sensitive information* (safety), and *exposing unnecessary details* (exposure).

3.3.2 RQ2: How do developers address their logging concerns?

Motivation

In the previous research question, we discuss developers' logging concerns (i.e., logging benefits and costs) that are communicated in their logging-related issue reports.

In this research question, we want to understand how developers address their logging concerns, i.e., how they balance the benefits and costs of logging. Our results can help software developers understand the maintenance effort of their logging code, which can help them make better allocation of their limited logging resources. In addition, developers can learn from others' logging experiences (e.g., about whether to remove a costly logging statement or reduce its log level) and apply them in their own logging practices. Our results also shed light on future research opportunities for addressing logging concerns automatically.

Approach

Developers usually track their activities of fixing an issue in an issue report (e.g., a JIRA issue report). A JIRA issue report uses a "Status" field to indicate the status of an issue in its lifecycle. The "Status" of an issue report can be "Open", "Patch Available", "Resolved", and "Closed", etc. A JIRA issue report also uses a "Resolution" field to indicate how the issue was resolved (e.g., "Fixed", "Duplicate", "Won't Fix", etc.), or otherwise "Unresolved". As mentioned in Section 3.2.1, we extracted our logging issue data after a long time (i.e., at least six months) since the creation of the logging issue reports, so the status of these issue reports tend to be stable at the time of our data extraction and follow-up analysis. In this RQ, we categorize the status of an issue report into four types: *open*, *fixed*, *patched*, and *rejected*.

- *Open* issue reports - the issue reports with an "Open", "Reopened", or "In Progress" status. These issues have not reached a solution yet, and a patch is not submitted for fixing these issues.
- *Rejected* issue reports - the issue reports with an "Invalid", "Not A Problem", or

"Won't Fix" resolution. Developers refused to make changes for these issues.

- *Fixed* issue reports - the issue reports with a "Fixed" resolution (in rare cases the resolution can be "Done", "Resolved", or "Pending Closed"). Developers have made code changes to address these issue reports.
- *Patched* issue reports - the issue reports with a "Patch Available" status. Developers have made code changes to address these issue reports (i.e., a patch is submitted), but the code changes are not integrated into the central code repositories (e.g., patch rejected by reviewers).

We continue to use the qualitative approach that we discuss in Section 3.2.3 to analyze how developers address their logging concerns. We focus our analysis on the *fixed* issue reports, as developers have not yet reached a solution for other issue reports. We examine the issue summaries, descriptions, comments, patches, commit messages and code review comments of the studied issue reports, and use the manual coding approach discussed in Section 3.2.3 to derive high-level concepts that conceptualize developers' resolutions for their logging issues.

We also discuss the resolutions for each of the logging concerns that are discussed in RQ1. For example, issue report HADOOP-13552 raises a concern about generating a "scary looking" stack trace as a warning when an exception is automatically handled by the code (i.e., *misleading end users*). The developer fixed the issue by reducing the log level from "warn" to "debug", and thereby addressing the concern of *misleading end users*.

Table 3.3: The status distribution of the studied logging issues.

Status	Open	Rejected	Fixed	Patched
# Logging issues	58 (10%)	31 (6%)	422 (75%)	49 (9%)

Results

Developers *fixed* 75% of the studied logging issues. Table 3.3 shows the distribution of the status of the studied logging issues. Among the 560 studied logging issues (as discussed in RQ1, some issue reports raise multiple logging issues), developers *rejected* to fix 6% of the logging issues, and left 10% and 9% of the remaining logging issues as *open* and *patched*, respectively. Figure 3.9 summarizes developers' resolutions for addressing their logging issues, which are derived from our qualitative analysis. In total, we derived 19 high-level resolutions which fall into four dimensions, namely *adding/removing log content*, *adjusting log level*, *improving logging configuration*, and *improving logging statements*. Table 3.4 explains each of the derived resolutions.

Developers constantly balance their log information through *adding/removing log content* and *adjusting log level*. Developers increase their log information through *adding log content*, *logging stack trace*, *increasing log level*, and *lowering logging threshold*. In comparison, they reduce their log information through *removing log content*, *reducing repetitive information*, *removing stack trace*, *reducing log level*, and *raising logging threshold*. As shown in Figure 3.9, developers *add log content* to address 32% to 37% of the studied logging issues. In contrast, they only *remove log content* to address 3% to 6% of the studied logging issues. Developers *reduce log levels* to address 8% to 20% of the studied logging issues, while only *increase log levels* to address 2% to 6% of the studied logging issues. In our manual analysis of these issue reports, we

Table 3.4: Developers' resolutions for addressing their logging issues.

Dimension	**Resolution**	**Description**	**Example**
Adding/ removing log content	Adding log content	Adding new logging statements or more information to existing logging statements	HADOOP-13854 added a new logging statement that records the error details of an exception, for debugging purposes
	Removing log content	Removing logging statements or partial content of logging statements	HADOOP-10422 removed a redundant logging statement
	Logging stack trace	Adding stack trace information to existing logging statements	HADOOP-13458 added stack trace information to an existing logging statement as it's hard to comprehend the meaning of an exception message without a stack trace
	Reducing repetitive information	Reducing log messages that communicate repetitive information	HADOOP-10756 reduced repetitive log information by consolidating similar log messages
	Removing stack trace	Removing stack trace information from logging statements	HADOOP-13710 removed the stack trace information from an exception logging as the exception is expected in a routine operation
Adjusting log level	Reducing log level	Reducing the log level of a logging statement (e.g., changing from "error" to "warn")	HADOOP-13850 reduced the log level of a logging statement from "info" to "debug", since users don't need to know the logged details
	Increasing log level	Increasing the log level of a logging statement (e.g., changing from "info" to "warn")	HADOOP-12789 increased the log level of a logging statement from "debug" to "info", since the log information (i.e., a class path) is critical for troubleshooting
	Raising logging threshold	Raising the logging threshold (i.e., the enabled log level) of a module, thus allowing less logging statements to print log messages	HIVE-15954 configured the log level of some modules from the default "info" level to "warn" level, in order to disable noise information that are logged at "info" level
	Lowering logging threshold	Lowering the logging threshold (i.e., the enabled log level) of a module, thus allowing more logging statements to print log messages	HIVE-5599 changed the allowed log level of Hive's root logger from "warn" to "info", since "Hive logs a large amount of important information at the info level"
Improving logging configuration	Changing logging configuration	Changing the environment setting of logging, such as logging format and logging destination	HADOOP-8552 added usernames to log file names, in order to avoid naming conflicts in multi-user scenarios
	Improving logging configurability	Improving the configurability of logging (e.g., log level configurability)	HADOOP-13098 added support for case-insensitive log level settings
	Turning on logging	Enabling logging in a module or to a destination	HIVE-3277 enabled Metastore audit logging for insecure connections
	Turning off logging	Disabling logging in a module or to a destination	HIVE-14852 disabled "qtest" logging to the console
	Redirecting log information	Redirecting log messages from one destination to another	KAFKA-2633 redirected log information from stdout to stderr
Improving logging statements	Improving log content	Improving existing logging statements to fix inappropriate or misleading log information	HADOOP-10126 improved the content of a misleading logging statement
	Reducing unneeded computing	Reducing unneeded computing when the log level of a logging statement is disabled, in order to reduce performance overhead	HADOOP-14369 removed expensive "toString()" invocations within logging statements as such invocations are not necessary when "debug" level is disabled
	Masking sensitive information	Removing/masking sensitive information (e.g., usernames and passwords) from log messages	HIVE-14098 removed passwords from the logging of environment variables by masking passwords with certain symbols
	Moving log location	Changing the location of logging statements in the source code	HIVE-10167 moved the location of a logging statement since the log was printed too late compared to the occurring time of the actual logged event
	Migrating logging API	Changing logging APIs from one logging library to another	HIVE-12237 migrated from Apache Commons Logging APIs to SLF4J logging APIs

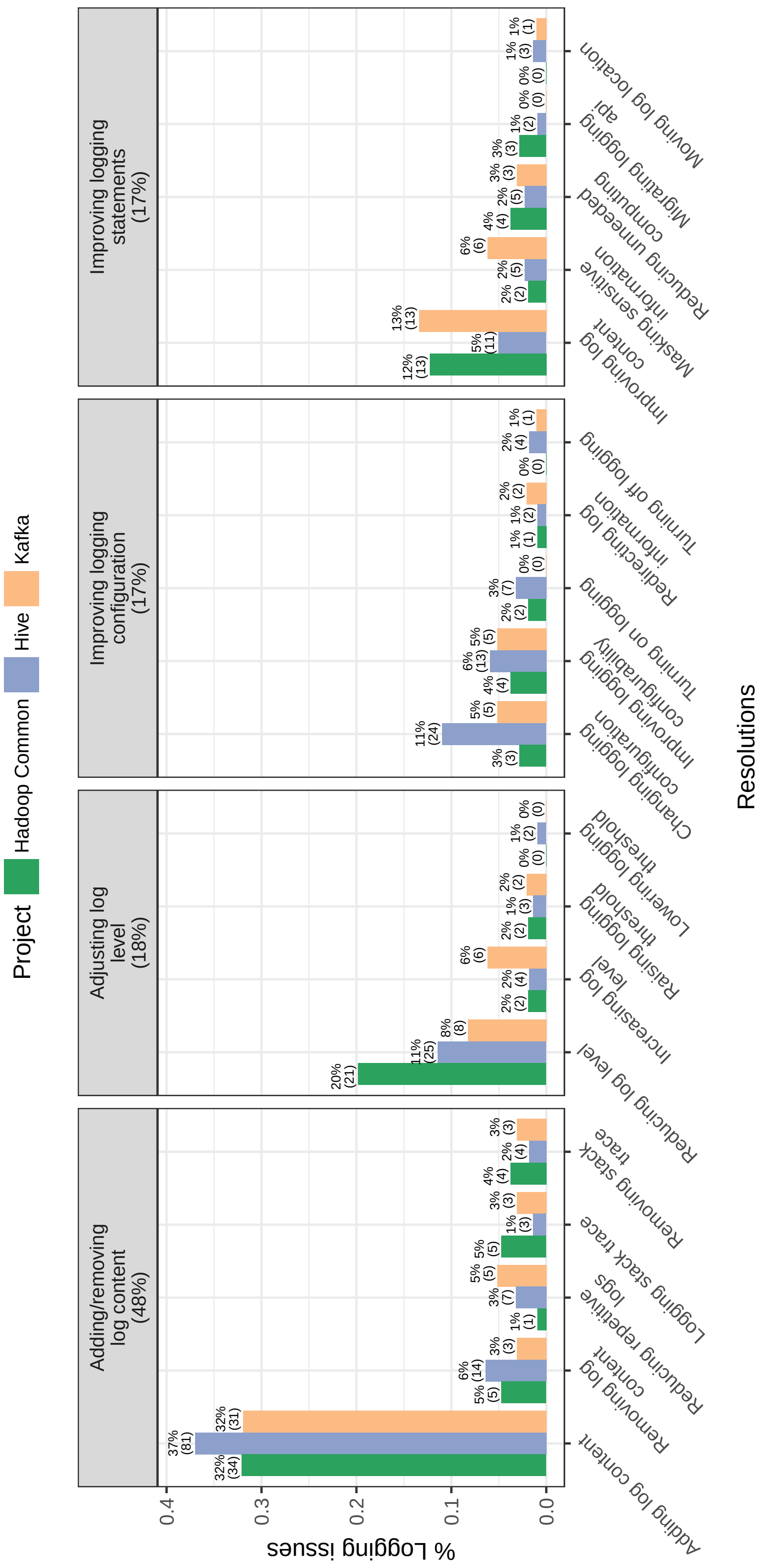


Figure 3.9: Developers' resolutions for addressing their logging issues and the percentage of logging issues in each project that are addressed by each resolution. The resolutions are grouped into four dimensions: *adding/removing log content*, *adjusting log level*, *improving logging configuration*, and *improving logging statements*.

find that developers tend to use a higher log level when they initially add a logging statement and later on reduce the log level. Such a behavior could bring unnecessary logging costs at the first place and increase the effort for maintaining the logging code. Developers need to pay extra attention to examine the log levels of their newly added logging statements (Li et al., 2017a; Chapter 6).

Developers' primary approach of addressing their concerns of logging benefits is *adding log content*, while their primary approach of addressing their concerns of logging costs is *reducing log level*. Figure 3.10 summarizes developers' resolutions for addressing each of their logging concerns. Developers *add log content* to address 67% to 82% of the logging issues that are concerned with each of the logging benefits. In addition, developers *increase log levels* to address 21% of the logging issues that are concerned with the logging benefit of *exposing runtime problems*. As we highlight in **PRACTICE 2**, higher log levels such as "warn" and "error" are needed to expose runtime problems.

The primary resolution for addressing the concerns of *excessive log information* (i.e., in 37% of the cases) and *exposing unnecessary details* (i.e., in 55% of the cases) is *reducing log level*. Developers also *reduce log levels* to address 24% of the logging issues that are concerned with *misleading end users*. In general, developers are more likely to reduce the level of a costly logging statement rather than removing the logging statement. Reducing the log level of a costly logging statement is safer than removing the logging statement, as the logging statement might still be useful in some scenarios (e.g., for debugging). However, developers are more likely to remove logging statements (i.e., in 13% of the cases) than reducing log levels (i.e., in 9% of the cases) to address the logging cost of *performance overhead*. This phenomenon can be explained by the fact that even disabled low-level logging statements can still cause performance overhead (see **ADVICE 5**).

Developers *improve logging configuration* to address 17% of the studied logging issues. Logging is usually highly configurable in a software system. Besides log level configuration (which we include in the *adjusting log level* dimension), developers also configure logging format, logging destinations, log rotation, and logging synchronization, etc. As shown in Figure 3.9, developers frequently *change logging configuration* (i.e., in 3% to 11% of the studied logging issues) or *improve logging configurability* (i.e., in 4% to 6% of the studied logging issues) to meet users' growing needs. Therefore, it is advisable to carefully examine the potential usage scenarios of the logging code when designing the configurability of logging.

Developers *improve logging statements* to address 17% of the studied logging issues. Logging statements produce log messages that are directly exposed to end users. Therefore, logging statements need to be accurate and should not expose inappropriate information (e.g., sensitive information). In practices, however, developers usually insert logging statements in an *ad hoc* manner. As a result, they usually need to improve their logging statements as *after-thoughts* (Yuan et al., 2012b). As shown in Figure 3.10, developers *improve log content* to address the logging cost of *misleading end users*, and *mask sensitive information* to address the logging cost of *exposing sensitive information*. Developers' primary resolutions for addressing the logging cost of *performance overhead* are *reducing unneeded computing* and *migrating logging api*. Future research is needed to help developers improve their logging statements automatically (e.g., by reducing unneeded computing in logging statements).

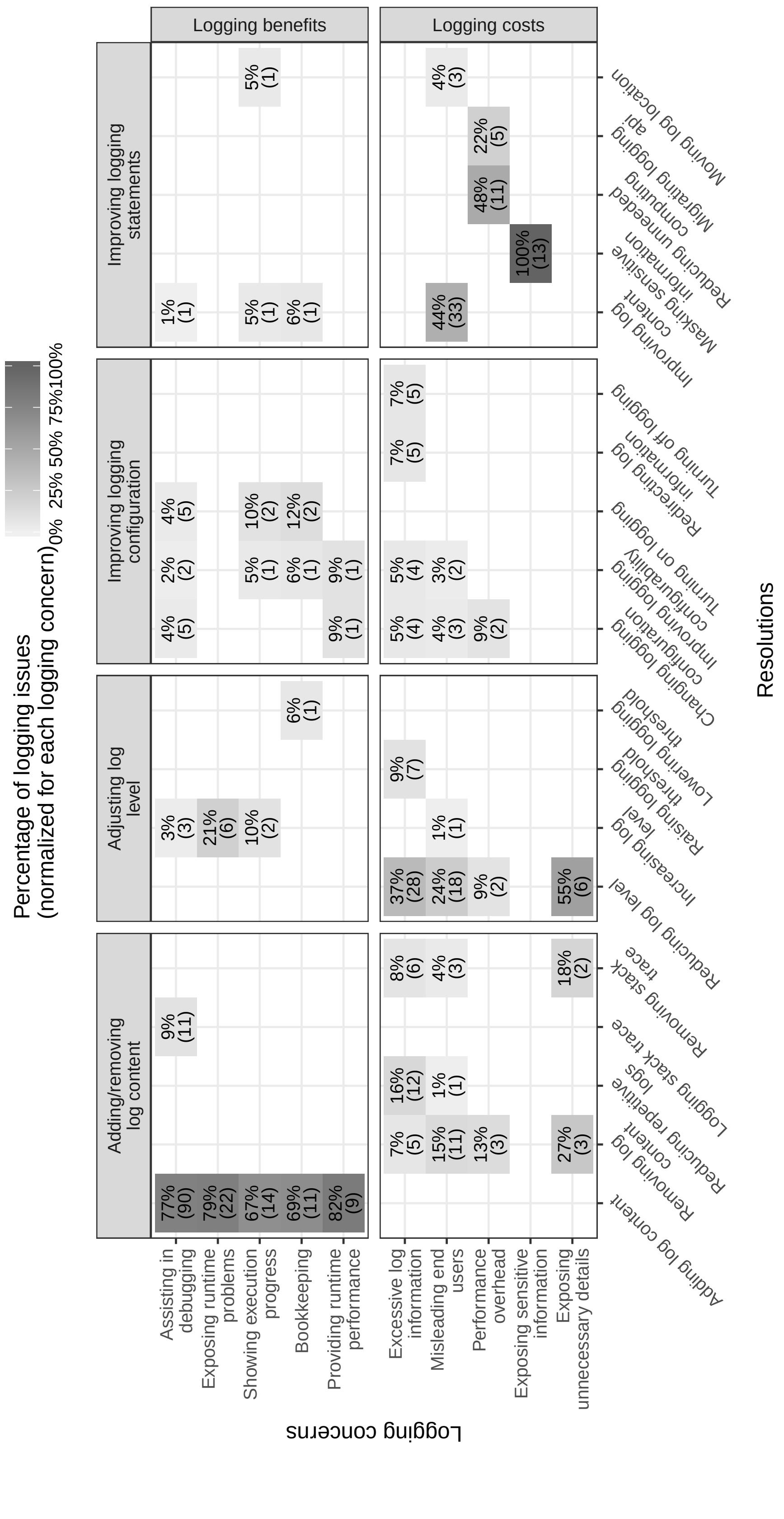


Figure 3.10: Developers' resolutions for addressing each of the logging concerns. Each grid in the heat map indicates the percentage of logging issues that are addressed by each resolution as normalized for each logging concern.

Discussion

Increasing/reducing log level versus lowering/raising logging threshold. As discussed in this RQ, developers usually change their log levels, either by adjusting the log levels of their individual logging statements (i.e., *increasing log level* or *reducing log level*) or by adjusting the default log level setting of a module (i.e., *raising logging threshold* or *lowering logging threshold*). Changing the log level of an individual logging statement would only affect the printing of the individual logging statement. In comparison, changing the default log level setting of a module would affect the printing of all the logging statement in that module.

Developers usually want the system to print the logging statements that are relevant to themselves, while suppressing other logging statements which could introduce noise. In some cases, developers play the "log level" trick to have their own logs highlighted. Taking issue report HIVE-10291 for example, developers use the "info" level for debugging purposes (which is not recommended) so that they could avoid configuring the log level setting to "debug" and seeing too much irrelevant "debug" information: "I thought of dumping at debug level, but it might be quite a pain to sift through all the debug logs to get at this." In issue report HIVE-10700, developers also use "info"/"warn" logs for debugging purposes. Their reason is similar: other irrelevant "debug" logs could cause too much noise. However, if many developers of a system use the "info" or higher log levels for their debugging logging statements, the system logs would be flooded with too much detailed log information. When assigning log levels for their logging statements, developers need to consider other stakeholders' logging needs and the overall logging resources of a system. It is advisable for logging libraries to provide the ability to detect developers' logging intentions and spot inappropriate usage of log levels. Future research is also needed to collect the log messages that are relevant to a developer (or a failure) and filter out the irrelevant ones.

Developers balance the benefits and costs of logging through *adding/removing log content*, *adjusting log level*, *improving logging configuration*, and *improving logging statements*. Developers' primary approach of addressing their concerns of logging benefits is *adding log content*, while their primary approach of addressing their concerns of logging costs is *reducing log level*.

3.4 Threats to Validity

External Validity. This chapter studies the logging concerns from the perspectives of the developers of three open source projects. Developers of other software projects might be concerned about different aspects of logging. We find that our findings are general among the three studied projects. We expect that our findings can also stand for other projects. However, findings from additional case studies on other projects can benefit our study.

Internal Validity. In this chapter, we study developers' logging concerns through a qualitative analysis of logging-related issue reports. However, developers' logging concerns may also be expressed in other forms, such as requirement specifications or mailing-lists. Therefore, our findings may not represent all of developers' logging concerns. In the three studied projects, an issue report is always needed for every code commit. Thus, we expect that our findings are quite representative for developers' logging concerns that involve code changes.

Construct Validity. We use a qualitative analysis to study developers' logging concerns and how developers address their logging concerns. Like other qualitative studies, our results are biased by the individuals who conduct the qualitative analysis. In order to reduce the bias, two researchers including the author of the thesis and a collaborator jointly perform the qualitative analysis to derive high-level concepts from the issue reports. Besides, our goal is not to accurately estimate the distribution of the derived logging concerns. Future quantitative studies are encouraged to validate the statistical distribution of developers' logging concerns.

3.5 Chapter Summary

Logging statements are beneficial for developers to understand system runtime behaviors and debug field failures. However, logging can also bring negative impact (e.g., performance overhead) to a software system. In order to understand developers' logging concerns (i.e., the benefits and costs of logging from developers' perspective) and how they address their logging concerns, we performed a qualitative study on 533 logging-related issue reports from three large and successful open source projects. We conceptualized developers' logging concerns (and how they address their concerns) into easy-to-perceive categories. We also derived best logging practices and general logging advice along with our qualitative analysis, which can help developers improve their logging code and be aware of some logging traps. Besides, logging library providers can learn from our advice to improve their logging libraries, e.g., to support different log levels for different parts of a logging statement. Our empirical findings also shed lights on future research opportunities for helping developers leverage the benefits of logging while minimizing logging costs.

CHAPTER 4

Understanding Software Logging Using Topic Models

In Chapter 3, we study developers' logging concerns and how they address such concerns. Learning to log approaches aim to help developers address their logging concerns by learning from existing logging code. These approaches usually consider the structural information of a code snippet to guide developers' logging decisions. In our qualitative analysis of logging-related issue reports (Chapter 3), we find that many issue reports are concerned with the logging of a few topics (e.g., "connections"). Therefore, this chapter studies the relationship between the semantic topics of a code snippet and the likelihood of a code snippet being logged. Our driving intuition is that certain topics in the source code are more likely to be logged than others. To validate our intuition, we conduct a case study on six open source systems, and we find that i) there exists a small number of "log-intensive" topics that are more likely to be logged than other topics; ii) each pair of the studied systems share 12% to 62% common topics, and the likelihood of logging such common topics has a statistically significant correlation of 0.35 to 0.62 among all the studied systems; and iii) our topic-based metrics help explain the likelihood of a code snippet being logged, providing an improvement of 3% to 13% on AUC and 6% to 16% on balanced accuracy over a set of baseline metrics that capture the structural information of a code snippet. Our findings highlight that topics contain valuable information that can help guide and drive developers' logging decisions.

An earlier version of this chapter is published in the Empirical Software Engineering Journal (EMSE) (Li et al., 2018).

4.1 Introduction

Developers depend heavily on logging statements for collecting valuable runtime information of software systems. Such information can be used for a variety of software quality assurance tasks, such as debugging and understanding system usage in production (Oliner et al., 2012; Xu et al., 2009b; Yuan et al., 2010; Mariani and Pastore, 2008; Syer et al., 2013; Chen et al., 2016a, 2017b). Logging statements are inserted by developers manually in the code to trace the system execution. As there exists no standard guidelines nor unified policies for software logging, developers usually miss including important logging statements in a system, resulting in blind code spots (i.e., cannot recover system execution paths) when debugging (Yuan et al., 2014, 2011).

However, adding logging statements excessively is not an optimal solution, since adding unnecessary logging statements can also bring negative impact, such as excessive log information and performance overhead (Chapter 3). Prior studies proposed approaches to enhance the information that is contained in logging statements through static analysis (Yuan et al., 2012a, 2011) and statistical models (Zhu et al., 2015; Lal and Sureka, 2016; Li et al., 2017b,a). These approaches help developers identify code locations that are in need of additional logging statements, or in need of log enhancement (e.g., requiring the logging of additional variables).

However, the aforementioned approaches do not take into account the functionality of a code snippet when making logging suggestions. We believe that code snippets that implement certain functionalities are more likely to require logging statements

```
public QueueConnection createQueueConnection()
throws JMSException
{
  QpidRASessionFactoryImpl s = new QpidRASessionFactoryImpl(_mcf, _cm,
      QpidRAConnectionFactory.QUEUE_CONNECTION);
  if (_log.isTraceEnabled())
    _log.trace("Created queue connection: "+s);
  return s;
}
```

Figure 4.1: A logged method that is related to the “connection” topic.

```
public String toString( String tabs )
{
  StringBuilder sb = new StringBuilder();
  sb.append( tabs ).append( "LessEqEvaluator : " ).append(
      super.toString() ).append( "\n" );
  return sb.toString();
}
```

Figure 4.2: A method that is related to the “string builder” topic.

than others. For example, Figure 4.1 and Figure 4.2 show two code snippets from the *Qpid-Java*[1] system. These two methods are of similar size and complexity, yet the method shown in Figure 4.1 has a logging statement to track a connection creation event, while the method shown in Figure 4.2 has no logging statements. The different logging decisions in these two code snippets might be explained by the fact that these two code snippets are related to different functionalities: the first code snippet is concerned with “connection”, while the second code snippet is concerned with “string builder”. In addition, in Section 4.2, we show real-life requirements for adding logging statements in the context of “connection”.

Prior research (Liu et al., 2009a; Nguyen et al., 2011; Maskeri et al., 2008; Linstead

[1]https://qpid.apache.org/components/java-broker

et al., 2008) leverage statistical topic models such as latent Dirichlet allocation (Blei et al., 2003) to approximate the functionality of a code snippet. Such topic models create automated topics (using co-occurrences of words in code snippets), and these topics provide high-level representations of the functionality of code snippets (Baldi et al., 2008; Thomas et al., 2010; Chen et al., 2016b).

We conjecture that source code that is related to certain topics is more likely to contain logging statements. We also want to determine if there exist common topics that are similarly logged across software systems. In particular, we performed an empirical study on the relationship between code topics and logging decisions in six open source systems: *Hadoop*, *Directory-Server*, *Qpid-Java*, *CloudStack*, *Camel* and *Airavata*. We focus on the following research questions:

RQ1: Which topics are more likely to be logged?

A small number of topics are more likely to be logged than other topics. Most of these log-intensive topics capture communication between machines or interaction between threads. Furthermore, we observe that the logging information that is captured by topics is not statistically correlated to code complexity.

RQ2: Are common topics logged similarly across different systems?

Each studied system shares a portion (12% to 62%) of its topics with other systems, and the likelihood of logging the common topics has a statistically significant correlation of 0.35 to 0.62 among these studied systems. Therefore, developers of a particular system can consult other systems when making their logging decisions or when developing logging guidelines.

RQ3: Can topics provide additional explanatory power for the likelihood of a code snippet being logged?

> Our topic-based metrics provide additional explanatory power (i.e., an improvement of 3% to 13% on AUC and an improvement of 6% to 16% on balanced accuracy) to a baseline model that is built using a set of metrics that capture the structural information of a code snippet, for explaining the likelihood of a code snippet being logged. Five to seven out of the top ten important metrics for determining the likelihood of a method being logged are our topic-based metrics.

Our chapter is the first work that studies the relationship between topics and logging decisions. Our findings show that source code related to certain topics is more likely to contain logging statements. Future log recommendation tools should consider topic information in order to help researchers and practitioners in deciding where to add logging statements.

Chapter organization. Section 4.2 uses examples to motivate the study of software logging using topic models. Section 4.3 provides a brief background about topic models. Section 4.4 describes our case study setup. Section 4.5 presents the answers to our research questions. Section 4.6 discusses potential threats to the validity of our study. Section 4.7 surveys related work. Finally, Section 4.8 concludes the chapter.

4.2 Motivation Examples

In this section, we use several real-life examples to motivate our study of the relationship between code topics and logging. Table 4.1 lists ten JIRA issue reports of the *Qpid-Java* system that we fetched from the Apache JIRA issue repository[2].

A closer examination of these ten issue reports shows that all these issue reports

[2]`https://issues.apache.org/jira`

are concerned with logging in the context of “connections”. For example, issue report QPID-4038[3] proposes to log certain connection details (e.g., local and remote addresses) after each successful connection, as “it will provide useful information when trying to match client application behaviour with broker behaviour during incident analysis”. The developer fixed this issue by adding the required logging information. Figure 4.3 gives a code snippet that is part of the code fix[4] for this issue. The code snippet shows that it is concerned with the topics that are related to “connections” (i.e., connection setting, connecting, get user ID, etc.). In fact, in RQ1 we found that “connection management” is one of the most log-intensive topics for the *Qpid-Java* system.

From these examples, we observed that software practitioners tend to use logs to record certain functionalities (or topics), for example, “connections”. However, we cannot manually investigate all the topics that need logging. Therefore, in this chapter, we propose to use topic modeling to understand the relationship between software logging and code topics in an automated fashion. Specifically, we want to study whether certain topics are more likely to be logged (RQ1). We also want to study whether there exist common topics that are similarly logged across systems (RQ2). Finally, we want to study whether topics can help explain the likelihood of a code snippet being logged (RQ3).

4.3 Topic Modeling

In this section, we briefly discuss the background of latent Dirichlet allocation (LDA), which is the topic modeling approach that we used in our study.

[3] https://issues.apache.org/jira/browse/QPID-4038

[4] Qpid-Java git commit: d606368b92f3952f57dbabd8553b3b6f426305e1

Table 4.1: Examples of JIRA issues of the *Qpid-Java* system that are concerned with the logging of "connections".

Issue ID[1]	**Issue report summary**
QPID-4038	Log the connection number and associated local and remote address after each successful [re]connection
QPID-7058	Log the current connection state when connection establishment times out
QPID-7079	Add connection state logging on idle timeout to 0-10 connections
QPID-3740	Add the client version string to the connection establishment logging
QPID-7539	Support connection and user level logging
QPID-2835	Implement connections (CON) operational logging on 0-10
QPID-3816	Add the client version to the connection open log messages
QPID-7542	Add connection and user info to log messages
QPID-5266	The client product is not logged in the connection open message
QPID-5265	The client version is only logged for 0-8/9/9-1 connections if a clientid is also set

[1] For more details about each issue, the readers can refer to its web link which is "https://issues.apache.org/jira/browse/" followed by the issue ID. For example, the link for the first issue is "https://issues.apache.org/jira/browse/QPID-4038".

```
  ConnectionSettings conSettings =
     retriveConnectionSettings(brokerDetail);
  _qpidConnection.setConnectionDelegate(new
     ClientConnectionDelegate(conSettings, _conn.getConnectionURL()));
  _qpidConnection.connect(conSettings);
  _conn.setConnected(true);
  _conn.setUsername(_qpidConnection.getUserID());
  _conn.setMaximumChannelCount(_qpidConnection.getChannelMax());
  _conn.getFailoverPolicy().attainedConnection();
+ _conn.logConnected(_qpidConnection.getLocalAddress(),
   _qpidConnection.getRemoteAddress());
```

Figure 4.3: A code snippet that is part of the fix for issue QPID-4038, showing that a logging statement was added to a code snippet within the context of "connections".

Our goal is to extract the functionality of a code snippet; however, such information is not readily available. Thus, we used the *linguistic data* in the source code files (i.e., the identifier names and comments) to extract topics of the code snippet in order to approximate the functionality in an automated and scalable fashion. We leveraged topic modeling approaches to derive topics (i.e., co-occurring words). Topic modeling approaches can automatically discover the underlying relationships among words in a corpus of documents (e.g., classes or methods in source code files), and group similar words together as topics. Unlike using words directly, topic models provide a higher-level overview and interpretable labels of the documents in a corpus (Blei et al., 2003; Steyvers and Griffiths, 2007).

In this chapter, we used latent Dirichlet allocation (LDA) (Blei et al., 2003) to derive topics. LDA is a probabilistic topic model that is widely used in Software Engineering research for modeling topics in software repositories (Chen et al., 2016b). Moreover, LDA generated topics are less likely to overfit and are easier to interpret, in comparison to other topic models such as probabilistic latent semantic analysis (PLSA), and latent semantic analysis (LSA) (Blei et al., 2003).

In LDA, a *topic* is a collection of frequently co-occurring words in the corpus. Given a corpus of n documents $f_1, ..., f_n$, LDA automatically discovers a set Z of topics, $Z = \{z_1, ..., z_K\}$, as well as the mapping θ between topics and documents (see Figure 4.4). 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. In a nutshell, LDA will generate two matrices – a topic-word matrix and a document-topic matrix. The topic-word matrix shows the most probable words in each topic, and the document-topic matrix shows the most probable topics in each document.

Formally, each topic is defined by a probability distribution over all of the unique words in the corpus (e.g., all source code files). Given two Dirichlet priors (used for

	Top words
z_1	*thread, sleep, notify, interrupt*
z_2	*network, bandwidth, timeout*
z_3	*view, html, javascript, css*

(a) Topics (Z).

	z_1	z_2	z_3
f_1	0.2	0.8	0.0
f_2	0.0	0.8	0.2
f_3	0.6	0.0	0.4
f_4	1.0	0.0	0.0

(b) Topic memberships (θ).

Figure 4.4: An example result of topic models, where three topics are discovered from four files. (a) The three discovered topics (z_1, z_2, z_3) are defined by their top (i.e., highest probable) words. (b) The four original source code files (f_1, f_2, f_3, f_4) are represented by the topic membership vectors (e.g., $\{z_1 = 0.2, z_2 = 0.8, z_3 = 0.0\}$ for file f_1).

computing Dirichlet distributions), α and β, LDA will generate a topic distribution, called θ_j, for each file f_j based on α, and generate a word distribution, called ϕ_i, for each topic z_i based on β. We exclude the mathematical details of LDA since they are out of the scope of this chapter. Interested readers may refer to the original paper on LDA (Blei et al., 2003) for the details.

4.4 Case Study Setup

This section describes the studied systems and the process that we followed to prepare the data for our case study[5].

4.4.1 Studied Systems

We performed a case study on six open source Java systems: *Hadoop*, *Directory-Server*, *Qpid-Java*, *CloudStack*, *Camel* and *Airavata* (Table 4.2). The studied systems are large and successful systems across different domains with years of development. *Hadoop* is

[5]Our replication package: `http://sailhome.cs.queensu.ca/replication/LoggingTopicModel`

a distributed computing platform; *Directory-Server* is an embeddable directory server; *Qpid-Java* is a message broker; *CloudStack* is a cloud computing platform; *Camel* is a rule-based routing and mediation framework; and *Airavata* is a framework for executing and managing computational jobs and workflows on distributed computing resources. The Java source code of these systems uses standard logging libraries such as *Log4j*[6], *SLF4J*[7], and *Commons Logging*[8]. We excluded test files from our analysis, since we are interested in the logging practices in the main source code files of these systems, and we expect that logging practices will vary between main and test code.

4.4.2 Data Extraction

Our goal is to study the relationship between logging decisions and the topics of the source code. We use topics to approximate the functionality of a code snippet. Therefore, we applied LDA at the granularity level of a source code *method*, since a method usually implements a relatively independent functionality. We did not apply LDA at the *class* level granularity because a class typically implements a mixture of functionalities. For example, a calculator class may implement input, internal calculation, and output functionalities.

Figure 4.5 presents an overview of our data extraction approach. We fetched the source code files of the studied systems from their Git repositories. We used the Eclipse Java development tools (JDT[9]) to analyze the source code and extract all the methods. Small methods usually implement simple functionalities (e.g., getters and setters, or initialize fields of a class object). Intuitively, such methods are less likely to have logging

[6]http://logging.apache.org/log4j
[7]http://www.slf4j.org
[8]https://commons.apache.org/logging
[9]http://www.eclipse.org/jdt

Table 4.2: Overview of the studied systems.

System	Release	LOC	Number of methods	Number of logged methods	Number of filtered methods	Filtered logged methods	Number of remaining methods	Remaining logged methods
Hadoop	2.5.0	1,194K	42.7K	2.9K (6.7%)	25.6K	156 (0.6%)	17.1K	2.7K (15.9%)
Directory-S.	2.0.0-M20	399K	7.9K	883 (11.2%)	3.3K	46 (1.4%)	4.5K	837 (18.4%)
Qpid-Java	6.0.0	476K	20.0K	1.3K (6.6%)	13.1K	62 (0.5%)	6.9K	1.2K (18.2%)
CloudStack	4.8.0	820K	40.1K	4.4K (10.9%)	28.4K	251 (0.9%)	11.7K	4.1K (35.1%)
Camel	2.17.0	1,342K	41.1K	2.9K (7.0%)	21.4K	126 (0.6%)	19.8K	2.7K (13.8%)
Airavata	0.15	446K	29.4K	1.8K (6.1%)	11.1K	26 (0.2%)	18.4K	1.8K (9.6%)

statements. For example, 95% of the logged methods are among the top 40% ($17.1K$ out of $42.7K$) largest methods, while only 5% of the logged methods in the *Hadoop* system are among the rest 60% ($25.6K$ out of $42.7K$) of the methods. Moreover, topic models are known to perform poorly on short documents. Therefore, for each system, we filtered out the methods that are smaller, in terms of LOC, than a predefined threshold. We defined the threshold for each system as the LOC of the 5% smallest methods that contain a logging statement. The thresholds are 8, 8, 8, 5, 8 and 4 for *Hadoop*, *Directory-Server*, *Qpid-Java*, *Camel*, *CloudStack* and *Airavata*, respectively. Table 4.2 also shows the effect of our filtering process, i.e., the number of methods that are filtered and kept, as well as the portions of them being logged, respectively. Section 4.5.3 discusses the effect of such filtering on our modeling results.

In order to study the relationship between logging decisions and the topics of methods, we removed all the logging statements from the logged methods before we performed the topic modeling. As logging statements contain textual information (e.g., “logger”, “info”, “print”) that is related to logging, extracting topics from the source code without removing logging statements would introduce bias for studying the relationship between logging and code topics. The use of standard logging libraries in these systems brings uniform formats (e.g., *logger.error(message)*) to the logging statements, thus we used a set of regular expressions to identify the logging statements. Finally, we preprocessed the log-removed methods and applied topic modeling on the preprocessed corpus of methods (see Section 4.4.3 “Source Code Preprocessing and LDA”).

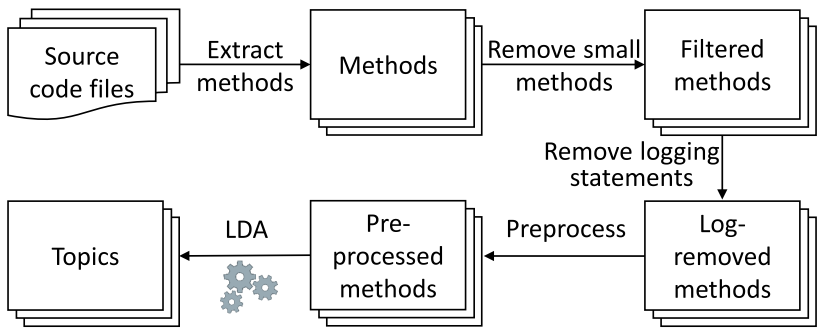


Figure 4.5: An overview of our data extraction approach.

4.4.3 Source Code Preprocessing and LDA

In this subsection, we discuss our source code preprocessing approach, and how we apply LDA on the preprocessed source code.

We extracted the linguistic data (i.e., identifier names, string literals, and comments) from the source code of each method, and tokenized the linguistic data into a set of words, similar to an approach that was proposed by Kuhn et al. (2007) and used in many prior studies (Chen et al., 2016b). With the set of words for each method, we applied common text preprocessing approaches such as removing English stop words (e.g., "a" and "the") and stemming (e.g., from "interruption" to "interrupt"). We also removed programming language keywords (e.g., "catch" and "return") from the set of words for each method. An open source implementation by Thomas (2012) eased our preprocessing of the source code. Finally, we applied LDA on both unigram (i.e., single word) and bigram (i.e., pairs of adjacent words) in each method, since including bigrams helps improve the assignments of words to topics and the creation of more meaningful topics (Brown et al., 1992).

Running LDA requires specifying a number of parameters such as K, α, and β (as explained in Section 4.3), as well as the number of Gibbs sampling iterations (II) for computing the Dirichlet distributions (i.e., per-document topic distributions and per-topic word distributions). These LDA parameters directly affect the quality of the LDA generated topics. However, choosing the optimal parameters values can be a computational expensive task (Panichella et al., 2013), and such optimal values may vary across systems and tasks (Panichella et al., 2013; Wallach et al., 2009; Chang et al., 2009). As a result, we applied hyper-parameter optimization to automatically find the optimal α and β when applying LDA using the MALLET tool (McCallum, 2002). A prior study by Wallach et al. (2009) found that using optimized hyper-parameters can improve the quality of the derived topics. We also set the number of Gibbs sampling iterations II to a relatively large number (10,000) such that LDA can produce more stable topics (Binkley et al., 2014).

We chose our K to be 500 when applying LDA on each studied system. As suggested by prior studies (Wallach et al., 2009; Chen et al., 2016b) using a larger K does not significantly affect the quality of LDA generated topics. The additional topics would have low topic membership values (i.e., noise topics), and can be filtered out. On the other hand, choosing a smaller K can be more problematic, since the topics cannot be separated precisely. We also tried other values of K in our study. However, we did not notice any significant differences in our findings (Section 4.6).

4.5 Case Study Results

In this section, we present the results of our research questions. For each research question, we present the motivation behind the research question, the approach that we used to answer the research question, and our experimental results.

4.5.1 RQ1: Which topics are more likely to be logged?

Motivation

In this research question, we study the relationship between topics in the source code and logging decisions. By studying this relationship, we can verify our intuition that the source code related to certain topics is more likely to contain logging statements. We are also interested in understanding which topics are more likely to contain logging statements. Since topics provide a high-level overview of a system, studying which topics are more likely to contain logging statements may provide insights about the logging practices in general.

Approach

We applied LDA on each of our studied systems separately to derive the topics for individual systems. In order to quantitatively measure how likely a topic is to be logged, we define the **log density** (**LD**) for a topic (z_i) as

$$\mathrm{LD}(z_i) = \frac{\sum_{j=1}^{n} \theta_{ij} * \mathrm{LgN}(m_j)}{\sum_{j=1}^{n} \theta_{ij} * \mathrm{LOC}(m_j)}. \tag{4.1}$$

where LgN(m_j) is the number of logging statements of method m_j, LOC(m_j) is the number of lines of code of method m_j, n is the total number of source code methods, and θ_{ij} is the topic membership of topic z_i in method m_j. A topic with a higher LD value is more likely to be logged.

As the LD metric does not consider the popularity of a topic, i.e., how many times a topic is logged, we also follow the approach of prior studies (Chen et al., 2012, 2017a) and define a **cumulative log density (CumLD)** for a topic (z_i) as

$$\text{CumLD}(z_i) = \sum_{j=1}^{n} \theta_{ij} * \frac{\text{LgN}(m_j)}{\text{LOC}(m_j)}, \tag{4.2}$$

A topic with a higher CumLD value is logged more often than a topic with a lower CumLD value. While the LD metric indicates the likelihood of a method of a particular topic being logged, the CumLD metric captures the overall relationship between a topic and logging. A topic might have a very high LD value, but there might only be a small number of methods that have a membership of such a topic; in contrast, such a topic would have a low CumLD value. Therefore, we consider both LD and CumLD metrics when we determine the top-log-density topics for detailed analysis. We define a topic as a **log-intensive topic** if the topic has both a high LD value and a high CumLD value.

We analyzed the statistical distribution of the log density values for all 500 topics in each system, to verify the assumption that some topics are more likely to be logged than other topics. We also manually studied the topics that have the highest log density values, i.e., the log-intensive topics, to find out which topics are more likely to be logged. For each log-intensive topic, *we not only analyzed the top words in this topic, but also investigated the methods that have the largest composition (i.e., large θ value) of the topic,* as well as the context of the methods, to understand the meaning and context of that particular topic.

Results

A small number of topics are much more likely to be logged. Table 4.3 shows the five number summary and the skewness of the log density (LD) values of the 500 topics for

Table 4.3: The five number summary and the skewness of the LD values of the 500 topics in each of the six studied systems.

System	Min	1st Qu.	Median	3rd Qu.	Max.	Skewness
Hadoop	0.00	0.01	0.01	0.02	0.07	0.98
Directory-S	0.00	0.00	0.01	0.02	0.10	2.10
Qpid-Java	0.00	0.00	0.01	0.01	0.06	1.72
Camel	0.00	0.01	0.01	0.02	0.10	1.61
Cloudstack	0.00	0.02	0.03	0.04	0.14	0.88
Airavata	0.00	0.00	0.01	0.02	0.16	2.32

Table 4.4: The five number summary and the skewness of the CumLD values of the 500 topics in each of the six studied systems.

System	Min	1st Qu.	Median	3rd Qu.	Max.	Skewness
Hadoop	0.00	0.11	0.24	0.44	3.55	2.90
Directory-S	0.00	0.01	0.04	0.10	3.68	9.76
Qpid-Java	0.00	0.01	0.05	0.16	7.58	13.49
Camel	0.00	0.11	0.25	0.57	5.95	3.65
CloudStack	0.00	0.16	0.42	0.82	5.14	2.64
Airavata	0.00	0.01	0.06	0.20	15.69	10.53

each studied system. The LD distribution is always positively skewed in every studied system. Taking the *Hadoop* system as an example, the minimal LD value for a topic is 0.00, the inter-quantile-range (the range from the first quantile to the third quantile) ranges from 0.01 to 0.02, while the maximum LD value for a topic is 0.07. The LD distribution for the *Hadoop* system has a skewness of 0.98 (a skewness of 1 is considered highly skewed (Groeneveld and Meeden, 1984)). Other studied systems have similar or more skewed distributions of the LD values, i.e., skewness ranges from 0.88 to 2.32. The high positive skewness indicates that a small number of topics are much more likely to be logged than other topics. Table 4.4 shows the five number summary and the skewness of the cumulative log density (CumLD) values of the 500 topics for each studied system. The CumLD values also present a highly skewed distribution, i.e., with a skewness of 2.64 to 13.49. The high skewness of the CumLD values implies that a small number of topics are logged more often than other topics.

Most of the log-intensive topics in the studied systems can be generalized to topics that are concerned with communication between machines or interaction between threads. Table 4.5 list the top six log-intensive topics for each system. In order to ensure that the six topics for each system have both the highest LD and CumLD values, we used an iterative approach to get these topics. Initially, we chose the intersection of the six topics with the highest LD values and the six topics with the highest CumLD values. If the number of topics in the intersection set is less than six, we chose the intersection of the seven topics with the highest LD values and the seven topics with the highest CumLD values. We continued expanding our search scope until we got the top six log-intensive topics. By manually studying the log-intensive topics in the studied systems, we labeled the meaning of each of these log-intensive topics in Table 4.5. 61% (22 out of 36) of the top log-intensive topics capture communication between machines, while 14% (5 out of 36) of the top log-intensive topics capture interactions between threads. We use a $*$ symbol in Table 4.5 to mark topics that are concerned with communication between machines, and use a † symbol in Table 4.5 to mark topics that are concerned with interactions between threads. For instance, the first log-intensive topic in the *Directory-Server* system, as well as the third log-intensive topic in the *Qpid-Java* system, are concerned with “connection management”. Developers tend to log the management operations, such as connecting, refreshing, closing, and information syncing, of a connection between two machines. As the communication process between two machines cannot be controlled or determined by a single

Table 4.5: Top six log-intensive topics in each system. The listed topics have the highest LD values and highest CumLD values. A topic label is manually derived from the top words in each topic and its corresponding source code methods. We use underscores to concatenate words into bigrams. A topic label marked with a “$*$” symbol or a “†” symbol indicates that the topic is concerned with communication between machines or interaction between threads, respectively.

System	LD	CumLD	Top words	Topic label
Hadoop	0.07	1.32	attr, file, client, nfsstatu, handl	network file system $*$
	0.05	3.55	thread, interrupt, except, interrupt_except, sleep	thread interruption †
	0.05	1.04	write, respons, verifi, repli, channel	handling write request $*$
	0.04	1.85	deleg, token, deleg_token, number, sequenc	delegation tokens $*$
	0.04	2.31	event, handl, handler, event_handler, handler_handl	event handling †
	0.04	1.07	command, shell, exec, executor, execut	OS command execution †
Directory-S	0.09	0.48	statu, disconnect, connect, replic_statu, replic	connection management $*$
	0.08	0.78	target, target_target, mojo, instal, command	installer target
	0.08	0.84	session, messag, session_session, session_write, write	session management $*$
	0.08	0.41	ldap, permiss, princip, permiss_except, ldap_permiss	LDAP[1]permission $*$
	0.06	2.17	contain, decod_except, except, decod, length	decoder exception
	0.06	3.68	close, debug, inherit, except, close_except	cursor operation
Qpid-Java	0.06	7.58	except, messag, error, except_except, occur	message exception $*$
	0.06	0.73	activ, spec, endpoint, handler, factori	Qpid activation
	0.05	1.15	connect, manag, manag_connect, info, qpid	connection management $*$
	0.05	1.21	resourc, except, resourc_except, resourc_adapt, adapt	JCA[2] $*$
	0.05	0.66	interv, heartbeat, setup_interv, heartbeat_interv, setup	heartbeat[3] $*$
	0.05	0.78	locat, transact_manag, manag, transact, manag_locat	transaction management
Camel	0.10	2.63	level, level_level, info, warn, messag	customized logging
	0.07	2.09	header, event, transact, event_header, presenc_agent	event header $*$
	0.07	2.41	interrupt, sleep, thread, reconnect, except	thread interruption †
	0.06	2.52	file, gener, gener_file, except, fail	remote file operation $*$
	0.06	4.23	channel, close, channel_channel, futur, disconnect	channel operation $*$
	0.05	2.30	send, messag, send_messag, websocket, messag_send	sending message $*$
CloudStack	0.10	1.75	result, router, execut, control, root	router operation $*$
	0.09	2.68	agent, host, attach, disconnect, transfer	agent connection $*$
	0.08	1.84	wait, except, timeout, interrupt, thread	thread interruption †
	0.08	1.92	command, citrix, base, resourc_base, citrix_resourc	citrix connection $*$
	0.07	2.64	context, context_context, overrid_context, overrid, manag	VM context operation
	0.07	3.02	host, hyper, hyper_host, context, vmware	host command request $*$
Airavata	0.16	9.21	object, overrid, object_object, format, format_object	customized logging
	0.13	15.69	type, resourc, except, resourc_type, registri	resource operation
	0.10	2.14	channel, except, queue, connect, exchang	channel operation $*$
	0.09	1.40	except, client, airavata, airavata_client, except_airavata	client connection $*$
	0.09	1.85	server, derbi, start, jdbc, except	server operation exception $*$
	0.08	2.63	server, port, transport, except, server_port	server operation $*$

[1] Lightweight directory access protocol.
[2] Java EE Connector Architecture (JCA) is a solution for connecting application servers and enterprise information systems.
[3] A heartbeat is a periodic signal sent between machines to indicate normal operations.

machine, logging statements provide a way for developers, testers, or users to monitor the communication processes and provide rich information for debugging such processes. Similarly, the interaction between threads cannot be controlled by a single thread, thus developers may also use logging statements more often to track such interactions between threads. As an example, the second log-intensive topic in *Hadoop* is about “thread interruption”.

Most top log-intensive topics only appear in one individual system, but a few topics emerge across systems. As we applied LDA on each studied system separately, it is not surprising that we generate mostly different topics for different systems, likewise for top log-intensive topics. For example, the first log-intensive topic in *Hadoop* is related to “network file system” (NFS). Developers use logging statements to track various operations on a network file system, such as creation, reading, writing and lookup. Although we know that such a topic is concerned with communication, the topic itself is not a general topic for all systems. Systems that do not use network file systems would not consider logging such a topic. Another example is the fourth log-intensive topic “LDAP permission” in *Directory-Server*. If a party is accessing a directory but it does not have the permission to access that particular directory, such a behavior would be logged as an error. Only the systems that use LDAP need to consider logging such a topic. However, a few topics do emerge across systems. For example, the second log-intensive topic in *Hadoop*, the third log-intensive topic in *Camel* and the third log-intensive topic in *CouldStack* are all concerned with “thread interruption”. For another example, the fifth log-intensive topic in *Camel* and the third log-intensive topic in *Airavata* are both related to “channel operation”. **The findings motivate us to study how common topics (i.e., topics shared by multiple systems) are logged across different**

Table 4.6: The five number summary and the skewness of the LD values of the topics in the *Hadoop* system.

Number of topics	Min	1st Qu.	Median	3rd Qu.	Max.	Skewness
100	0.00	0.01	0.01	0.02	0.04	0.71
500	0.00	0.01	0.01	0.02	0.07	0.98
1,000	0.00	0.01	0.01	0.02	0.07	1.29

systems (see RQ2).

Discussion

Impact of choosing a different number of topics. In this RQ, we use LDA to identify 500 topics for each system and study the distribution of log density among these topics. We now explore how the choice of the number of topics impacts our analysis in this RQ. In this sub-section, we consider the *Hadoop* system as an example, and vary the number of topics between 100 and 1,000. Table 4.6 and Table 4.7 summarize the distributions of the LD values and the CumLD values for the *Hadoop* system when varying the number of topics. As we increase the number of topics, the skewness of the LD values and the skewness of the CumLD values both increase. This phenomenon can be explained by the intuition that using a larger number of topics can better distinguish log-intensive topics from other topics. However, both the LD values and the CumLD values still present highly positive-skewed distributions when we vary the number of topics, which supports our observation that a small number of topics are much more likely to be logged.

Table 4.8 lists the top six log-intensive topics in the *Hadoop* system when choosing a different number of topics (i.e., 100, 500, and 1,000). The top log-intensive topics do not remain the same when we vary the number of topics, because using a different

Table 4.7: The five number summary and the skewness of the CumLD values of the topics in the *Hadoop* system.

Number of topics	Min	1st Qu.	Median	3rd Qu.	Max.	Skewness
100	0.30	0.87	1.37	2.35	8.66	1.99
500	0.00	0.11	0.24	0.44	3.55	2.90
1,000	0.00	0.02	0.08	0.23	3.56	4.21

number of topics generates topics at different granularity. However, some topics, such as “thread interruption”, “event handling”, “network file system”, and “OS command execution”, do appear among the top log-intensive topics when varying the number of topics. We highlight these common topics in **bold** font in Table 4.8. Moreover, even when we vary the number of topics, most of the log-intensive topics are still about communication between machines or interaction between threads. We also have similar observations in the other studied systems.

Relationship between topics and structural complexity. In this RQ, we found that a few topics are more likely to be logged than other topics. However, it is possible that these differences are related to the differences of the code structures. In this subsection, we examine the relationship between the topics and the structural complexity of a method.

We use McCabe’s cyclomatic complexity (McCabe, 1976) (**CCN**) to measure the structural complexity of a method. We define two metrics, topic diversity (**TD**) and topic-weighted log density (**TWLD**), to measure the diversity of topics in a method (i.e., cohesion) and the log density of a method which is inferred from its topics, respectively. The topic diversity, which is also called *topic entropy* (Misra et al., 2008; Hall et al., 2008), of a method is defined as $\mathrm{TD}(m_j) = -\sum_{i=0}^{T} \theta_{ij} \log_2 \theta_{ij}$, where θ_{ij} is the membership of topic i in method j and T is the total number of topics. A larger topic

Table 4.8: Top six log-intensive topics in the *Hadoop* system, using different number of topics. A topic label marked with a "$*$" symbol or a "†" symbol indicates that the topic is concerned with communication between machines or interaction between threads, respectively. The bold font highlights the common topics that appear among the top log-intensive topics when varying the number of topics.

Number of topics	Top words	Topic label
100	thread, except, interrupt, interrupt_except, wait	**thread interruption** †
	servic, server, stop, start, handler	server operation $*$
	event, event_event, handl, event_type, handler	**event handling** †
	block, replica, datanod, pool, block_block	**work node operation** $*$
	resourc, request, contain, prioriti, node	resource allocation $*$
	contain, contain_contain, statu, launch, contain_statu	container allocation $*$
500	attr, file, client, nfsstatu, handl	**network file system** $*$
	thread, interrupt, except, interrupt_except, sleep	**thread interruption** †
	write, respons, verifi, repli, channel	handling write request $*$
	deleg, token, deleg_token, number, sequenc	**delegation tokens** $*$
	event, handl, handler, event_handler, handler_handl	**event handling** †
	command, shell, exec, executor, execut	**OS command execution** †
1000	attr, file, client, nfsstatu, handl	**network file system** $*$
	bean, mbean, info, object, info_bean	bean object
	node, path, node_path, data, path_node	**work node operation** $*$
	thread, interrupt, except, interrupt_except, wait	**thread interruption** †
	state, deleg, master, secret_manag, manag	**delegation tokens** $*$
	command, shell, exec, exit, exit_code	**OS command execution** †

diversity means that a method is more heterogeneous, while a smaller topic diversity means that a method is more coherent.

The topic-weighted log density of a method j is defined as $\text{TWLD}(m_j) = \sum_{i=0}^{T} \theta_{ij}\text{LD}_{i,-j}$, where $\text{LD}_{i,-j}$ is the log density of topic i that is calculated from Equation 4.1 considering all the methods except for the method j. When calculating the TWLD value of a method, we excluded that particular method from Equation 4.1 to calculate the log density of topics, in order to avoid bias. A large TWLD value means that a method contains a large proportion of log-intensive topics.

Figure 4.6 shows the pairwise Spearman rank correlation between cyclomatic complexity (CCN), topic diversity (TD), and topic-weighted log density (TWLD) of all the methods in our studied systems. We use the Spearman rank correlation because it is robust to non-normally distributed data (Swinscow et al., 2002). In fact, the Shapiro-Wilk normality test shows that the distributions of these three metrics are all statistically significantly different from a normal distribution (i.e., p-value < 0.05). Topic diversity and cyclomatic complexity have a positive correlation of 0.22 to 0.39 in the studied systems. In other words, more structurally complex methods tend to have more diverse topics, which matches prior findings (Liu et al., 2009b). On the other hand, the topic-weighted log density of a method has a very weak (-0.15 to 0.21) correlation (Swinscow et al., 2002) with the cyclomatic complexity of a method, which means that the log intensity of the topics is unlikely to be correlated with the cyclomatic complexity of the code. Therefore, **even though structurally complex methods tend to have diverse topics, the logging information that is captured by these topics is not correlated with code complexity.**

A small number of topics are more likely to be logged than other topics. Most of these log-intensive topics in the studied systems correspond to communication between machines or interaction between threads. Our findings encourage future work to develop topic-based logging guidelines (i.e., which topics need developers' further attention for logging).

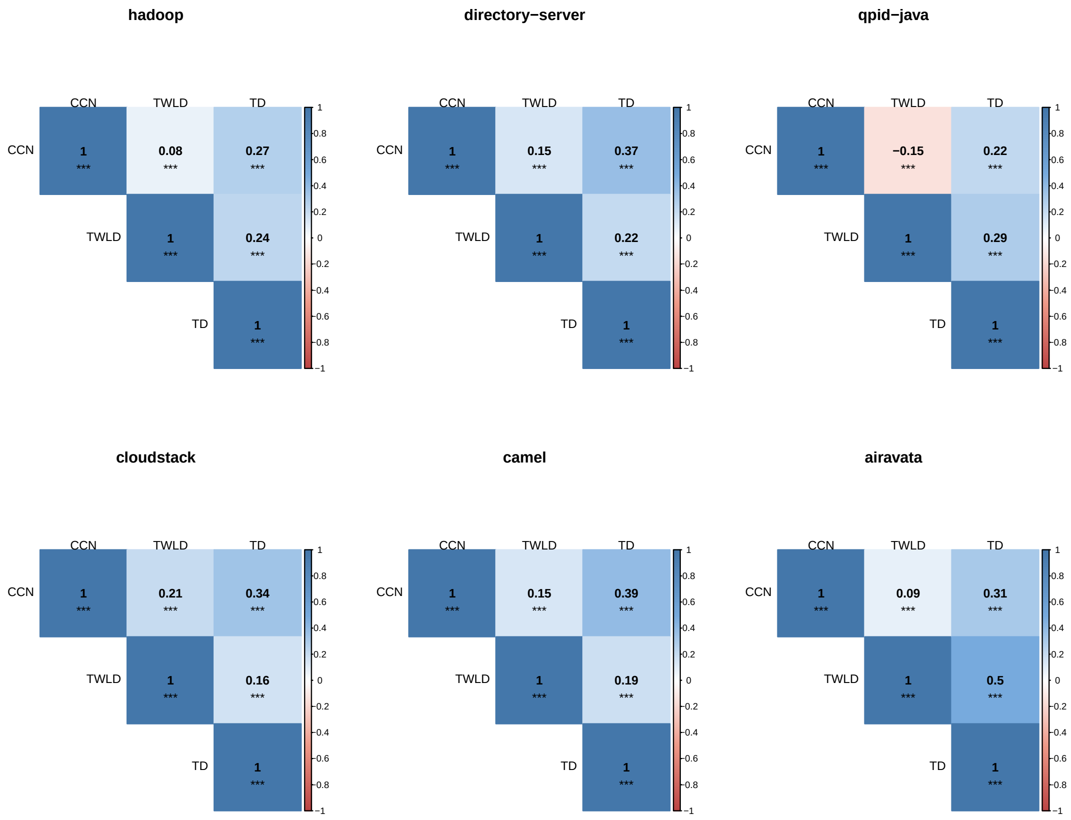


Figure 4.6: Pairwise Spearman correlation between cyclomatic complexity (CCN), topic diversity (TD), and topic-weighted log density (TWLD). The symbols below the correlation values indicate the statistical significance of the respective correlation: o $p \geq 0.05$; * $p < 0.05$; ** $p < 0.01$; *** $p < 0.001$.

4.5.2 RQ2: Are common topics logged similarly across different systems?

Motivation

In RQ1, we applied LDA on each system separately and we got mostly different top log-intensive topics for different systems. However, we did find a few top log-intensive topics that emerge across different systems. Therefore, in this research question, we quantitatively study how common topics are logged across different systems. If common topics are similarly logged across different systems, we might be able to provide general suggestions on what topics should be logged across systems; otherwise, developers should make logging decisions based on the context of their individual system.

Approach

Cross-system topics. In order to precisely study the logged topics across different systems, we combined the methods of the studied systems together into one corpus, and applied LDA using K=3,000. We use 3,000 topics as we hope to identify topics that have the same granularity as the topics that we identified in RQ1 (i.e., 500 topics $*$ 6 systems). We used the same preprocessing and topic modeling approach as we had applied to individual systems in RQ1. We refer to the resulting topics as "cross-system topics". With the cross-system topics, we firstly need to determine whether a topic exists in each studied system. If a topic exists in multiple systems, then this topic is common among multiple systems.

Topic assignment in a system. We use the topic assignment to measure the total presence of a topic in a system. The assignment of a topic in a system is the sum of that topic's memberships in all the methods of that system. A higher topic assignment means that a larger portion of the methods is related to the topic (Thomas et al., 2014; Baldi et al., 2008). The assignment of topic z_i in system s_k is defined as

$$\mathrm{A}(z_i, s_k) = \sum_{j=0}^{N_k} \theta_{ij}, \tag{4.3}$$

where N_k is the number of methods in system s_k, and θ_{ij} is the topic membership of topic z_i in method m_j.

As different systems have different number of methods, it is unfair to compare the assignment of a topic in different systems. Therefore, we instead use a normalized definition of assignment:

$$\mathrm{AN}(z_i, s_k) = \sum_{j=0}^{N_k} \theta_{ij} / N_k, \tag{4.4}$$

The normalized assignment values of all the topics sum up to 1 for each individual system. We refer to normalized assignment as “assignment” hereafter.

Common topics shared across systems. Figure 4.7 shows the cumulative assignments of all the topics in each system when sorting the topics by their assignments. For each system, a small portion of topics (208 to 696 out of 3,000 topics) account for 90% of the total assignment of each system. In other words, only a small portion of topics are significantly assigned in each system. For each system, we define its **important topics** as its most assigned topics that account for 90% of the total assignment of that particular system. For example, 696 out of 3,000 topics are important topics in the *Hadoop* system.

We define a topic to be a **common topic** if the topic is important in multiple systems. For example, if a topic is important in two systems, then this topic is commonly shared between the two systems. If a topic is important in all the studied systems, then this topic is commonly shared across all the studied systems.

Log density correlation. In order to study whether common topics are logged similarly across different systems, we measured the pairwise correlation of the log density of the common topics that are shared among different systems. Specifically, for each pair of systems, we first calculated their respective log density values for their common topics, so we calculate two sets of log density values for the same set of common topics.

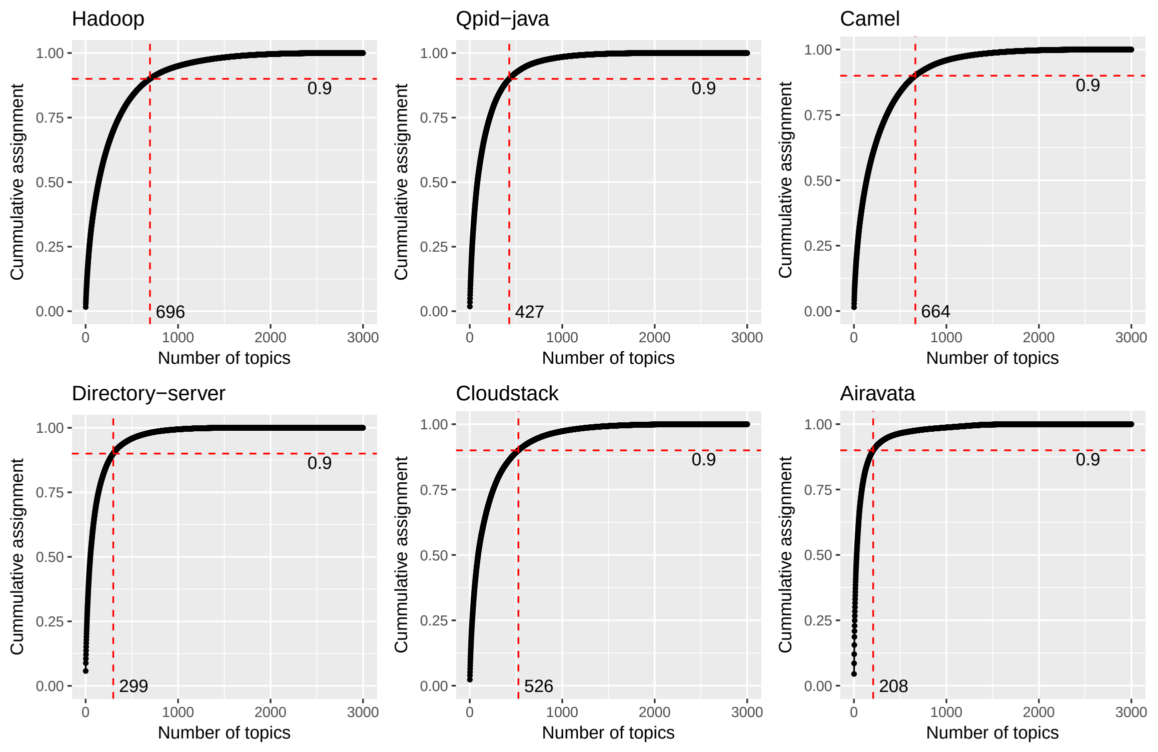


Figure 4.7: The cumulative assignment of all the topics in each studied system. The topics are sorted by their assignments from high to low.

We then calculated the Spearman rank correlation between these two sets of log density values. A large correlation value indicates that the common topics are logged similarly across these two systems. As discussed in RQ1, the log density values of the topics have a skewed distribution. In fact, the Shapiro-Wilk test shows that the distributions of the log density values are statistically significantly different from a normal distribution (i.e., p-value < 0.05). Therefore, we chose the Spearman rank correlation method because it is robust to non-normally distributed data (Swinscow et al., 2002). Prior studies also applied Spearman ranking correlation to measure similarity (e.g. Goshtasby, 2012).

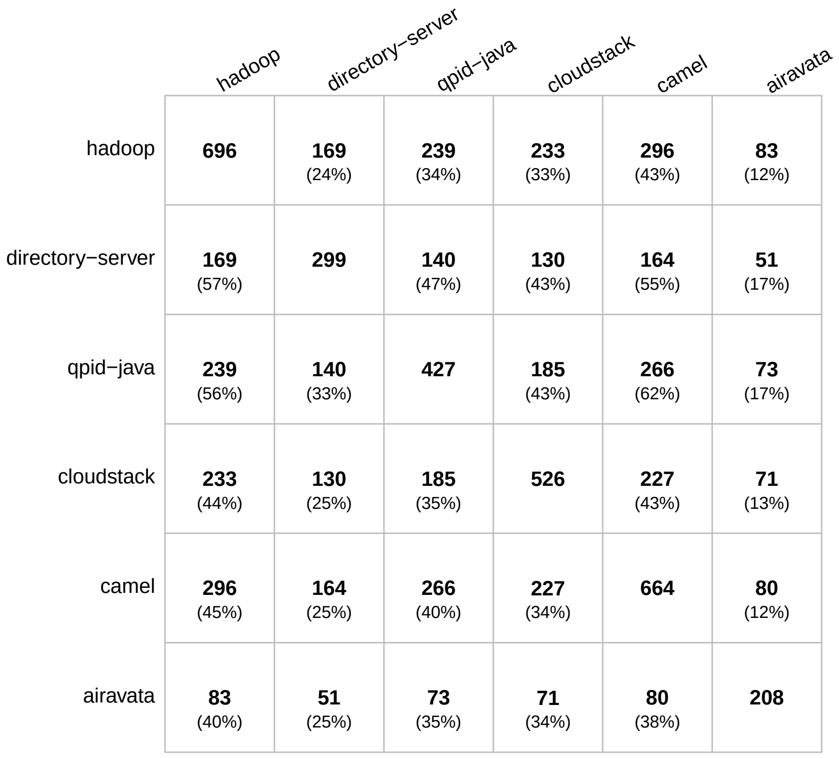
Table 4.9: Number of topics that are shared by $N \in \{1,2,...,6\}$ systems.

# Systems	$N = 0$	$N = 1$	$N = 2$	$N = 3$	$N = 4$	$N = 5$	$N = 6$
# Shared topics	1,359 (45%)	1,130 (38%)	203 (7%)	109 (4%)	77 (3%)	83 (3%)	39 (1%)

Results

All the studied systems share a portion (i.e., 12% to 62%) of their topics with other systems. Table 4.9 lists the number of topics that are shared by $N \in \{1,2,...,6\}$ systems. Among all the 3,000 topics, around half (1,641) of them are important in at least one system, while the rest of them (1,359) are not important in any system. Around one-sixth (511 topics) of the topics are shared by at least two systems, among which only 39 topics are shared by all the six studies systems. Figure 4.8 lists the numbers of common topics that are shared between each pair of systems. For each system, Figure 4.8 also shows the percentage of its topics that are shared with each of the other systems. As shown in the figure, each studied system shares 12% to 62% of its topics with each of the other systems. In general, *Hadoop* and *Camel* share the most topics with other systems, possibly because they are platform or framework applications that contain many modules of various functionalities. In comparison, *Airavata* share the least topics with other systems. Specifically, *Hadoop* and *Camel* share the most topics (296) between them, while *Directory-server* and *Airavata* share the least topics (51).

The likelihood of logging the common topics has a statistically significant correlation of 0.35 to 0.62 among all the studied systems. Figure 4.9 shows the Spearman correlation of the log density between each pair of systems on their common topics. For each pair of systems, their log density values of the common topics have a statistically significant (i.e., p-value < 0.05) correlation of 0.35 to 0.62. In other words,



	hadoop	directory−server	qpid−java	cloudstack	camel	airavata
hadoop	**696**	**169** (24%)	**239** (34%)	**233** (33%)	**296** (43%)	**83** (12%)
directory−server	**169** (57%)	**299**	**140** (47%)	**130** (43%)	**164** (55%)	**51** (17%)
qpid−java	**239** (56%)	**140** (33%)	**427**	**185** (43%)	**266** (62%)	**73** (17%)
cloudstack	**233** (44%)	**130** (25%)	**185** (35%)	**526**	**227** (43%)	**71** (13%)
camel	**296** (45%)	**164** (25%)	**266** (40%)	**227** (34%)	**664**	**80** (12%)
airavata	**83** (40%)	**51** (25%)	**73** (35%)	**71** (34%)	**80** (38%)	**208**

Figure 4.8: The number of topics that are shared between each pair of systems. The numbers in the diagonal cells show the number of important topics per system. The percentage values show the percentage of topics in the system indicated by the row name that are shared with the system indicated by the column name.

the likelihood of logging the common topics is statistically significantly correlated between each pair of the studied systems. The *Hadoop* system and the *Cloudstack* system have the largest log density correlation (0.62) on their common topics. As a distributed computing platform and a cloud computing platform, respectively, these two systems are likely to share similar logging needs for their common topics. The *Qpid-Java* system and the *Airavata* system have the smallest log density correlation (0.35) on their common topics. As a message broker and a framework for managing and executing

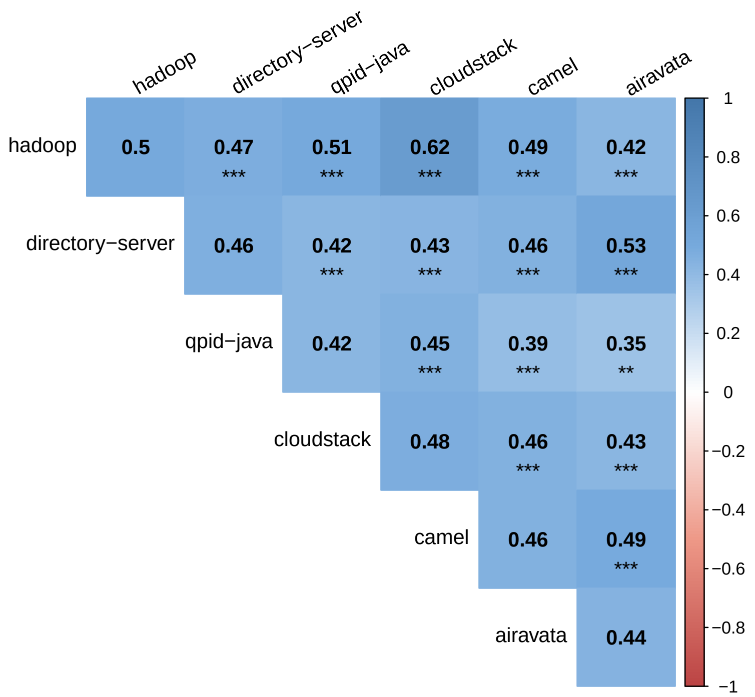


Figure 4.9: The Spearman correlation of the log density of the common topics that are shared between each pair of systems. The values in the diagonal cells show the average log density correlation between each system and other systems on the shared topics. The symbols below the correlation values indicate the statistical significance of the respective correlation: o $p \geq 0.05$; * $p < 0.05$; ** $p < 0.01$; *** $p < 0.001$.

computational jobs, respectively, these two systems are less likely to have similar logging needs.

Discussion

How do similar systems log common topics? In our case study, we chose six systems from different domains. We found that each system shares a portion (12% to 62%) of topics with other systems, and that the likelihood of logging the common topics is statistically significantly correlated among these systems. It is interesting to discuss how similar systems log their common topics. Therefore, we analyzed the common topics that are shared by two similar systems: Qpid-Java and ActiveMQ. Both systems are popular open source message brokers implemented in Java. Specifically, we added the *ActiveMQ* system into our cross-system topic modeling. We still set the number of topics to be 3,000, as we found that adding the new system into our cross-system topic modeling does not significantly change the number of important topics of the existing systems.

Table 4.10 shows the number of common topics between these two systems and their log density correlation. As shown in the table, *ActiveMQ* has a wider range of topics than *Qpid-Java*. The former has 675 important topics while the later has 432 important topics. The larger number of important topics in *ActiveMQ* is likely because *ActiveMQ* is not only a message broker, but it also supports many other features such as enterprise integration patterns[10]. These two systems share 294 common topics. The *Qpid-Java* system shares 68% (the largest percentage for each pair of systems) of its topics with the *ActiveMQ* system. The respective log density values of these common topics have a statistically significant correlation of 0.45, which is not the highest correlation value between each pair of systems. In summary, for similar systems such as *Qpid-Java* and *ActiveMQ*, they may share a relatively large portion of common topics; however, their likelihood of logging such common topics does not necessarily have a larger correlation than a pair of systems from different domains.

Topics shared by all the studied systems. As shown in Table 4.9, there are only 39 topics that are commonly shared among all the studied systems. We measured each

[10]http://activemq.apache.org

Table 4.10: Common topics between two similar systems: *Qpid-Java* and *ActiveMQ*. The symbols below a correlation value indicate the statistical significance of the correlation: *** $p < 0.001$.

System	# Important topics	# Common topics	Log density correlation
Qpid-Java	432	294 (68%)	0.45
ActiveMQ	675	294 (44%)	***

system's log density for these 39 topics and calculated their pairwise Spearman correlations. The log density values of the studied systems have a statistically significant correlation of 0.38 to 0.70. In other words, the likelihood of logging these common topics is statistically correlated among all the studied systems. Table 4.11 also lists the six most log-intensive topics and the six least log-intensive topics among the 39 common topics. After manual analysis and labeling, we found that these two groups of topics have very distinguishable patterns. Most of the top-logged topics are concerned with communication between machines or interactions between threads, such as "stopping server" and "finding host". In comparison, most of the least-logged topics are concerned with low-level data structure operations, such as "hash coding" and "string indexing".

Impact of choosing a different number of topics. In this RQ, we chose 3,000 topics for the cross-system topic modeling. We now examine whether our choice of the number of topics impacts our results. Using the *Hadoop* system as an example, Table 4.12 shows the cross-system topic modeling results when varying the number of topics from 3,000 to 2,000 and 1,000. As we decrease the number of topics from 3,000 to 1,000, the number of important topics for the *Hadoop* system also decreases from 696 to 384, at a lower decreasing ratio. The median number of common topics that are shared between *Hadoop* and other systems also decreases from 233 to 148. However, the percentage of the common topics increases from 33% to 39%. In other words, as we decrease the

Table 4.11: The common topics that are shared by all of the six studied systems: The six most log-intensive topics and the six least log-intensive topics. A topic label marked with a "$*$" symbol or a "†" symbol indicates that the topic is concerned with communication between machines or interaction between threads, respectively.

	Top words	Topic label
Most likely logged topics	stop, except, overrid, stop_except, overrid_stop, servic , except_stop, shutdown, servic_stop, stop_servic	stopping server $*$
	except, except_except, error, thrown, except_thrown, param, occur, error_occur, except_error, thrown_error	throwing exception
	host, host_host, list_host, find, host_type, list, host_list, host_find, type_host, find_host	finding host $*$
	connect, connect_connect, except, except_connect, connect_except, close, connect_close, creat_connect, connect_host, creat	connection management $*$
	event, event_event, handl, event_type, type, event_handler, handler, handler_handl, overrid, event_applic	event handling †
	messag, messag_messag, except, except_messag, messag_except, messag_param, param_messag, object_messag, overrid, object	message exception $*$
Least likely logged topics	hash, code, hash_code, overrid, overrid_hash, code_result, prime, prime_result, result_prime, code_hash	hash coding
	equal, object, overrid, equal_object, overrid_equal, result_equal, equal_equal, object_equal, equal_type, type_equal	equal operation
	append, append_append, builder, builder_builder, overrid, builder_append, overrid_builder, length_append, time_append, type_append	string builder
	system, println, system_println, print, usag, except_system, println_system, exit, println_usag, usag_system	printing
	index, index_index, substr, start_index, param, substr_index, length, length_index, size, list_index	string indexing
	node, node_node, node_list, list_node, param_node, type_node, except_node, node_type, node_param, param	graph node management

number of topics, the topics become more coarse-grained and they are more likely to be shared by multiple systems. Finally, the log density correlation of the common topics between the *Hadoop* system and other systems does not change significantly when we vary the number of topics from 3,000 to 1,000; in fact, the median correlation values remain around 0.5 and the correlations are always statistically significant while we vary the number of topics. Similar observations also hold to the other studied systems. Overall, our results in this research question are not sensitive to the number of topics that is used in the cross-system topic modeling.

Table 4.12: Cross-system topic modeling results when varying the number of topics, using the *Hadoop* system as an example.

System	# Topics	# Important topics	# Common topics (median)	Log density correlation (median)
Hadoop	3,000	696	233 (33%)	0.49
	2,000	584	213 (36%)	0.45
	1,000	384	148 (39%)	0.53

Each studied system shares a portion (12% to 62%) of its topics with other systems. The likelihood of logging the common topics has a statistically significant correlation of 0.35 to 0.62 among all the studied systems. Developers of a particular system can consult other systems when making their logging decisions or when developing logging guidelines.

4.5.3 RQ3: Can topics provide additional explanatory power for the likelihood of a code snippet being logged?

Motivation

In RQ1, we observed that source code that is related to certain topics is more likely to be logged. In this RQ, we further studied the statistical relationship between topics and logging. We are interested in knowing whether our code topics can offer a different view of logging. Namely, we want to study whether adding topic-based metrics to a set of baseline metrics can provide additional explanatory power for the likelihood of a code snippet being logged.

Approach

To answer this research question, we built regression models to study the relationship between the topics in a method and the likelihood of a method being logged. The response variable of our regression models is a dichotomous variable that indicates whether a method should have a logging statement or not, and the explanatory variables are represented by a set of baseline metrics and topic-based metrics. The baseline metrics capture the structural information of a method, while the topic-based metrics capture the semantic information of a method.

Baseline metrics. We used 14 baseline metrics, as listed in Table 4.13, to capture the structural information of a method. Prior studies (Yuan et al., 2012a; Fu et al., 2014; Zhu et al., 2015) found that the structure of a code snippet exhibits a strong relation with its logging needs. Table 4.13 also briefly explains the rationale behind studying each of these baseline metrics.

Topic-based metrics. The topic modeling results give us the membership (θ) assigned for each of the topics in each method. We consider the membership values that are assigned to the topics as the topic-based metrics, denoted by T0-T499. Prior studies also used similar topic-based metrics to predict or understand the relationship between topics and software defects (Nguyen et al., 2011; Chen et al., 2012). We filtered out topic membership values that are less than a threshold (we use 0.01 as the threshold) to remove noise topics for each method (Wallach et al., 2009; Chen et al., 2012).

Model construction. We built LASSO (least absolute shrinkage and selection operator (Tibshirani, 1996)) models to study the relationship between the explanatory metrics of a method and a response variable that indicates whether a method should have a logging statement or not. We use a LASSO model because it uses regularization

Table 4.13: Selected baseline metrics and the rationale behind the choices of these metrics.

Metric	**Definition (d) | Rationale (r)**
LOC	d: Number of lines of code in a method. r: Large methods are likely to have more logging statements.
CCN	d: McCabe's cyclomatic complexity (McCabe, 1976) of a method. r: Complex methods are likely to have more logging statements.
NUM_TRY	d: Number of *try* statements in a method. r: A *try* block indicates developers' uncertainty about the execution outcome of code, thus developers tend to use logging statements for monitoring or debugging purposes.
NUM_CATCH	d: Number of *catch* clauses in a method. r: Exception catching code is often logged (Yuan et al., 2012a; Fu et al., 2014; Zhu et al., 2015; Microsoft-MSDN, 2016; Apache-Commons, 2016).
NUM_THROW	d: Number of *throw* statements in a method. r: A logging statement is sometimes inserted right before a *throw* statement (Fu et al., 2014); developers also sometimes re-throw an exception instead of logging an exception.
NUM_THROWS	d: Number of *throws* clauses in a method declaration. r: Methods that throw exceptions are likely to have logging statements.
NUM_IF	d: Number of *if* statements in a method. r: Developers tend to log logic-branch points for understanding execution traces (Fu et al., 2014).
NUM_ELSE	d: Number of *else* clauses in a method. r: Developers tend to log logic-branch points for understanding execution traces (Fu et al., 2014).
NUM_SWITCH	d: Number of *switch* statements in a method. r: Developers tend to log logic-branch points for understanding execution traces (Fu et al., 2014).
NUM_FOR	d: Number of *for* statements in a method. r: Logging statements inside loops usually record the execution path or status of the loops.
NUM_WHILE	d: Number of *while* statements in a method. r: Logging statements inside loops usually record the execution path or status of the loops.
NUM_RETURN	d: Number of *return* statements in a method. r: More *return* statements indicates a more complex method (i.e., more possible execution outcomes); such a method is more likely to be logged for monitoring or debugging purposes.
NUM_METHOD	d: Number of method invocations in a method. r: Developers tend to check and log a return value from a method invocation (Fu et al., 2014).
FANIN	d: The number of classes that depend on (i.e., reference) the containing class of a method. r: High fan-in classes like libraries might have less logging statements to avoid the generation of too much logging.

to penalize a complex model that leads to over-fitting and it conducts feature selection simultaneously (Tibshirani, 1996; Kuhn and Johnson, 2013). An over-fitted model performs very well on the data on which the model was built, but usually has poor accuracy on a new data sample (Kuhn and Johnson, 2013). It is generally true that more complex models are more likely to lead to over-fitting (Kuhn and Johnson, 2013). The LASSO model uses a λ parameter to penalize the complexity of a model: the larger the λ value, the simpler the model (Tibshirani, 1996). Among the 500 topic-based metrics, many of them have little or no contribution for determining the logging likelihood of a method. A LASSO model, with a proper setting of the λ parameter, enables us to significantly reduce the number of variables in the model and reduce the possibility of over-fitting (Tibshirani, 1996).

We used the stratified random sampling method (Witten and Frank, 2005; Kuhn and Johnson, 2013) to split the dataset of a system into 80% of training dataset and 20% of testing dataset, such that the distributions of logged methods and unlogged methods are properly reflected in both the training and testing datasets. We used the 80% training dataset to construct the model and tune the λ parameter, and left the 20% testing dataset only for testing purpose using the already tuned λ parameter. Similar "80%:20%" splitting approaches were also used by prior studies (Martin et al., 2012; Kuhn and Johnson, 2013). Splitting the dataset into distinct sets for model construction (including parameter tuning) and model evaluation ensures that we avoid overfitting and that we provide an unbiased sense of model performance (Kuhn and Johnson, 2013).

We used 10-fold cross validations to tune the λ value in a LASSO model, using only the training dataset. For each λ value, we used a 10-fold cross validation to measure the performance of the model (represented by AUC) using the λ value, and repeated for different λ values until we find a λ value with the best model performance. In this way, we got a LASSO model with the best cross-validated performance and we can avoid over-fitting. We used the "cv.glmnet" function in the "glmnet" R package (Friedman et al., 2010; Simon et al., 2011) to implement our model tuning process.

Model evaluation. We used *balanced accuracy* (**BA**) as proposed by a prior study (Zhu et al., 2015) to evaluate the performance of our LASSO models. BA averages the probability of correctly identifying a logged method and the probability of correctly identifying a non-logged method. BA is widely used to evaluate the modeling results on imbalanced data (Zhu et al., 2015; Cohen et al., 2004; Zhang et al., 2005), since it avoids over optimism on imbalanced data sets. BA is calculated by Equation (4.5):

$$BA = \frac{1}{2} \times \frac{TP}{TP + FN} + \frac{1}{2} \times \frac{TN}{FP + TN} \tag{4.5}$$

where *TP*, *FP*, *FN* and *TN* represent true positive, false positive, false negative and true negative, respectively.

We also used the area under the ROC (receiver operating characteristic) curve (**AUC**) to evaluate the performance of the LASSO models. While the BA provides a balanced measure on our models' accuracy in classifying logged methods and non-logged methods, the AUC evaluates our models' ability of discrimination, i.e., how likely a model is able to correctly classify an actual logged method as a logged method, rather than classify an actual unlogged method as a logged method. The AUC is the area under the ROC curve which plots the true positive rate $(TP/(TP+FN))$ against false positive rate $(FP/(FP+TN))$. The AUC ranges between 0 and 1. A high value for the AUC indicates a classifier with a high discriminative ability; an AUC of

0.5 indicates a performance that is no better than random guessing.

Evaluating the effect of the metrics on the model output. We evaluated the effect of the metrics (i.e., the explanatory variables) on the model output, i.e., the likelihood of a method being logged, by comparing the metrics' standardized regression coefficients in the LASSO models. Standardized regression coefficients describe the expected change in the response variable (in standard deviation units) for a standard deviation change in a explanatory variable, while keeping the other explanatory variables fixed (Kabacoff, 2011; Bring, 1994). A positive coefficient means that a high value of that particular variable is associated with a higher probability of a method being logged, while a negative coefficient means that a high value of that particular variable is associated with a lower probability of a method being logged. For example, a topic-based metric with a positive coefficient means that a method with a greater membership of that particular topic has a higher chance to be logged. The standardized regression coefficients are not biased by the different scale of different variables in the model. In this work, we calculate the standardized regression coefficients by standardizing each of the explanatory variables to a mean of 0 and a standard deviation of 1, before feeding the data to the LASSO models.

Results

Table 4.14 shows the performance of the models that are built using the baseline metrics, and the models that are built using both the baseline and topic-based metrics. A high AUC indicates that our LASSO models are able to discriminate logged methods versus not-logged methods. A high BA implies that our LASSO models are able to provide accurate classification for the likelihood of a method being logged. The results highlight that developers are able to leverage a model to aid their logging decisions.

Adding topic-based metrics to the baseline models gives a 3% to 13% improvement on AUC and a 6% to 16% improvement on BA for the LASSO models. In order to evaluate the statistical significance of adding the topic-based metrics to our baseline models, we used a Wilcoxon signed-rank test to compare the performance of the models that only use the baseline metrics and the performance of the models that use both the baseline and topic-based metrics. The Wilcoxon signed-rank test is the non-parametric analogue to the paired t-test. We use the Wilcoxon signed-rank test instead of the paired t-test because the former does not assume a normal distribution of the compared data. We use a p-value that is below 0.05 to indicate that the alternative hypothesis (i.e., the performance change is statistically significant) is true. The test on the AUC values and the test on the BA values both result in a p-value of 0.02, which means that adding the topic-based metrics statistically significantly improves the performance of our LASSO models. We also computed Cliff's δ effect size (Macbeth et al., 2011) to compare the performance of the models that only use the baseline metrics versus the performance of the models that use both the baseline metrics and the topic-based metrics. Cliff's δ also has no assumption on the normality of the compared data. The magnitude of Cliff's δ is assessed using the thresholds that are provided by Romano et al. (2006), i.e., $\delta < 0.147$ "negligible", $\delta < 0.33$ "small", $\delta < 0.474$ "medium", and $\delta >= 0.474$ "large". As shown in Table 4.14, the effect size of the AUC improvement is 0.72 (large), and the effect size of the BA improvement is 0.69 (large). Therefore, topic-related metrics provide additional explanatory power to the models that are built using the structural baseline metrics. In other words, topics can provide additional explanatory power for the likelihood of a method being logged.

Table 4.14: Performance of the LASSO models, evaluated by AUC and BA.

Project	Baseline metrics		Baseline + Topics	
	AUC	BA	AUC	BA
Hadoop	0.82	0.72	0.87 (+6%)	0.78 (+7%)
Directory-Server	0.86	0.75	0.94 (+9%)	0.86 (+16%)
Qpid-Java	0.80	0.74	0.90 (+13%)	0.82 (+10%)
Camel	0.86	0.78	0.90 (+4%)	0.82 (+6%)
CloudStack	0.83	0.76	0.88 (+6%)	0.80 (+6%)
Airavata	0.96	0.88	0.99 (+3%)	0.95 (+8%)
Cliff's δ	-	-	0.72 (large)	0.69 (large)
P-value (Wilcoxon)	-	-	0.02 (sig.)	0.02 (sig.)

Both our baseline and topic-based metrics play important roles in determining the likelihood of a method being logged. Table 4.15 shows the top ten metrics for each LASSO model that uses both the baseline metrics and the topic-based metrics. These metrics are ordered by the absolute value of their corresponding standardized coefficients in the models. In each model, **five to seven of the top ten important metrics for determining the likelihood of a method being logged are our topic-based metrics.**

The baseline metrics NUM_TRY, NUM_METHOD, and NUM_CATCH have a strong relationship with the likelihood of a method being logged. Each of these three metrics appears at least four times in the top ten metrics and has a positive coefficient in the LASSO models for all studied systems. Developers tend to log *try* blocks as they are concerned about the uncertainty during the execution of *try* blocks; developers log method invocations as developers usually need to check and record the return values of such method invocations; developers log *catch* blocks as a mean to handle exceptions for debugging purposes (Microsoft-MSDN, 2016; Apache-Commons, 2016). The baseline metrics NUM_THROW, NUM_THROWS and FANIN each appears twice in the top ten metrics. The NUM_THROW metric has a negative coefficient in both of these two occurrences, indicating that developers tend not to throw an exception and log it at the same time; instead, they tend to log when they are catching an exception. In contrast, the NUM_THROWS metric has a positive coefficient, showing that developers tend to add logging statements in methods that specify potential exceptions that might be thrown in that particular method or callee methods (with the latter case being more usual). The FANIN metric has a negative coefficient, indicating that high fan-in code tends to be associated with less logging statements, possibly for reducing logging overheads when called by other methods. Both the LOC and CNN metrics appear only once in the top ten metrics. The LOC metric has a positive coefficient, which is obvious as larger methods are more likely to require logging statements. The CCN metric also has a positive coefficient, indicating that developers tend to log complex methods which may need future debugging (Shang et al., 2015).

The topic-based metrics play important roles in the LASSO models; in particular, the log-intensive topics have a strong and positive relationship with the likelihood of a method being logged. As shown in Table 4.15, we manually derived the topic label for each topic-based metric, by investigating the top words in the topic, the methods that have the largest membership of the topic, and the containing classes of these methods. We use a ‡ symbol to mark the log-intensive metrics that we uncovered in RQ1. The metrics based on the log-intensive topics that are labeled as "cursor operation", "decoder exception", "message exception", "session management", "connection management", "event handling", "resource operation" and "customized logging", have positive coefficients in the LASSO models, indicating that these topics have a positive relationship with the likelihood of a method being logged.

Table 4.15: The top ten important metrics for determining the likelihood of a method being logged and their standardized coefficients. A letter “T” followed by a parenthesis indicates a topic-based metric and the manually derived topic label. A topic label followed by a ‡ symbol indicates that the particular topic is a log-intensive topic as listed in Table 4.5.

Hadoop		Directory-Server		Qpid-Java	
Metric	**Coef**	**Metric**	**Coef**	**Metric**	**Coef**
NUM_METHOD	0.72	NUM_METHOD	0.73	T (message exception) ‡	0.77
NUM_CATCH	0.42	NUM_TRY	0.58	LOC	0.62
T (prototype builder)	-0.31	T (cursor operation ‡)	0.43	NUM_RETURN	-0.54
CCN	0.28	T (decoder exception ‡)	0.31	T (list iteration)	-0.49
T (server protocal)	-0.26	T (cursor exception)	-0.28	NUM_IF	-0.26
NUM_TRY	0.25	T (string builder)	-0.24	T (connection management ‡)	0.25
NUM_THROW	-0.22	T (naming exception)	-0.22	NUM_CATCH	0.25
T (client protocal)	-0.21	FANIN	-0.18	T (object attribute)	-0.20
T (equal operation)	-0.15	T (state transition)	-0.18	T (write flag)	-0.19
T (string builder)	-0.14	T (tree operation)	0.15	T (session management) ‡	0.17
Camel		CloudStack		Airavata	
Metric	**Coef**	**Metric**	**Coef**	**Metric**	**Coef**
NUM_METHOD	1.13	NUM_TRY	0.80	NUM_TRY	2.09
NUM_TRY	0.29	NUM_METHOD	0.62	FANIN	-0.83
NUM_THROWS	0.28	NUM_CATCH	0.44	T (Thrift code - object reader)	-0.69
T (JSON schema)	-0.22	T (search parameter)	-0.25	T (Thrift code - object writer)	-0.69
NUM_CATCH	0.22	T (search entity)	-0.25	NUM_THROWS	0.39
NUM_THROW	-0.17	T (server response)	-0.20	NUM_METHOD	0.37
T (string builder)	-0.16	T (legacy transaction)	-0.16	T (result validation)	-0.33
T (model description)	-0.15	T (search criteria)	-0.15	T (resource operation) ‡	0.31
T (REST configuration)	-0.13	NUM_RETURN	0.14	T (customized logging) ‡	0.23
T (event handling) ‡	0.11	T (equal operation)	-0.14	T (result transfer)	0.17

In particular, the topic labeled as “message exception” has the strongest relationship with the likelihood of a method being logged in the *Qpid-Java* system. The topics that are labeled as “cursor operation” and “decoder exception”, also play the most important roles in determining the likelihood of a method being logged in the *Directory-Server* system. The “tree operation” topic in the *Directory-Server* system and the “result transfer” topic in the *Airavata* system also have a positive relationship with the likelihood of a method being logged. We found that the “tree operation” topic has an LD value of 0.03; and the “result transfer” topic has an LD value of 0.07. These two topics are also considered as log-intensive topics. Other topics that are listed in Table 4.15 have a negative relationship with the likelihood of a method being logged. These topics have an LD value of 0.00 to 0.01, which are much smaller than the log density values of the log-intensive topics (i.e., methods related to these topics most likely do not have any logging statements).

Discussion

Cross-system evaluation. In this research question, we evaluated the performance of our log recommendation models in a within-system setting. It is also interesting to study the performance of the models in a cross-system evaluation, i.e., train a model using one system (i.e., the training system) then use the trained model to predict the likelihood of logging a method in another system (i.e., the testing system). Like what we did in RQ2, we applied cross-system topic modeling on a combined corpus of the six studied systems and set the number of topics to be 3,000. Then we derived topic-based metrics that are used as explanatory variables in our LASSO models.

As discussed in RQ2, however, different systems have different sets of important topics. This issue poses a challenge to our cross-system evaluation, i.e., the training system and the testing system have different variable settings, which results in the poor performance of the cross-system models that leverage topic-based metrics.

Even though we cannot fully overcome the fact that different systems have different sets of important topics which leads to the poor performance of cross-system models, we took two strategies to alleviate the issue:

- When training a LASSO model, we used the common topics between the training system and the testing system as our topic-based topics. We used the method mentioned in RQ2 to get the common topics of each pair of systems.

- When training the LASSO model, we assigned more weight to the methods in the training system that have a larger membership of the important topics in the testing system. Specifically, for each method in the training system, we gave it a weight that is its total membership of all the important topics in the testing system.

Tables 4.16 and 4.17 list the performance (AUC) of the cross-system models that use the baseline metrics and the performance (AUC) of the cross-system models that use both the baseline and topic-based metrics, respectively. For each system, we also calculated the average performance (AUC) of the models that were trained using other systems and tested on that particular system. The average AUC values increase by 1% to 7% when topic-based metrics are added to the baseline models. We also used a Wilcoxon singed-rank test and computed Cliff's δ effect size to compare the average AUC values when using baseline metrics and when using both the baseline and topic-based metrics. The Wilcoxon signed-rank test got a p-value of 0.02, which indicates that the topic-based metrics bring statistically significant improvement to the baseline models. The Cliff's δ effect size is 0.44, which means that the improvement is considered as "medium".

The effect of choosing a different number of topics. In this chapter, we derived 500 topics from the source code of a software system and leveraged these topics to study the relationship between the topics of a method and the likelihood of a method being logged. In order to evaluate the impact of the choice of number of topics on our findings, we conducted a sensitivity analysis to quantitatively measure how the different

Table 4.16: The performance (AUC) of the cross-system models using baseline metrics. The row names indicate the training systems and the column names indicate the testing systems.

	Hadoop	Directory-Server	Qpid-Java	CloudStack	Camel	Airavata
Hadoop	-	0.80	0.66	0.82	0.86	0.88
Directory-Server	0.74	-	0.61	0.74	0.78	0.91
Qpid-Java	0.60	0.69	-	0.53	0.43	0.61
CloudStack	0.78	0.80	0.61	-	0.84	0.93
Camel	0.80	0.81	0.65	0.82	-	0.90
Airavata	0.74	0.81	0.61	0.80	0.78	-
Average	0.73	0.78	0.63	0.74	0.74	0.85

Table 4.17: The performance (AUC) of the cross-system models using both baseline and topic-based metrics. The row names indicate the training systems and the column names indicate the testing systems.

	Hadoop	Directory-Server	Qpid-Java	CloudStack	Camel	Airavata
Hadoop	-	0.82	0.67	0.83	0.86	0.90
Directory-Server	0.78	-	0.63	0.79	0.81	0.92
Qpid-Java	0.74	0.69	-	0.71	0.67	0.82
CloudStack	0.79	0.80	0.70	-	0.84	0.90
Camel	0.82	0.82	0.69	0.82	-	0.90
Airavata	0.74	0.81	0.67	0.80	0.80	-
Average	0.77	0.79	0.67	0.79	0.79	0.89
	(+5%)	(+1%)	(+6%)	(+7%)	(+7%)	(+5%)

number of topics influence the topic model's ability to explain the likelihood of a code snippet being logged. Specifically, we changed the number of topics that we used in RQ3 from 500 to various numbers (i.e., from 20 to 3,000), and built LASSO models that leverage both the baseline metrics and the topic-based metrics. Table 4.18 shows the performance (evaluated using AUC) of these LASSO models that leverage the baseline metrics and the topic-based metrics that are derived from different number of topics. As we increase the number of topics from 20 to 3,000, the AUC values of the LASSO models increase until they reach a plateau. The AUC values of the LASSO models stay

Table 4.18: Performance (AUC) of the LASSO models that leverage the baseline metrics and the topics-based metrics derived from different numbers of topics.

Project	Baseline	Baseline + 20–3,000 topics										
		20	50	100	300	500	800	1,000	1,500	2,000	2,500	3,000
Hadoop	0.82	0.83	0.84	0.84	0.86	0.87	**0.88**	0.88	0.86	0.86	0.87	0.86
Directory-S.	0.86	0.88	0.87	0.90	0.93	**0.94**	0.94	0.94	0.94	0.93	0.94	0.93
Qpid-Java	0.80	0.83	0.85	0.88	**0.90**	0.90	0.90	0.89	0.89	0.89	0.89	0.89
Camel	0.86	0.87	0.88	0.88	**0.90**	0.90	0.90	0.90	0.90	0.90	0.89	0.90
Cloudstack	0.83	0.85	0.86	0.86	**0.89**	0.88	0.88	0.88	0.88	0.87	0.88	0.88
Airavata	0.96	0.98	**0.99**	0.99	0.99	0.99	0.99	0.99	0.99	0.98	0.98	0.99
Cliff's δ[1]	-	0.33^M	0.44^M	0.56^L	0.67^L	0.72^L	0.72^L	0.72^L	0.67^L	0.67^L	0.72^L	0.67^L

[1] The superscripts S, M, and L represent small, medium, and large effect sizes, respectively.

at or slightly fluctuate around the maximum point as we continue to increase the number of topics. Taking the Directory Server system for example, the AUC values of the LASSO models increase from 0.88 to 0.94 as we increase the number of topics from 20 to 500. However, as we continue to increase the number of topics, the AUC values stay around 0.94. As observed by Wallach et al. (2009), the reason may be that as the number of topics increases, the additional topics are rarely used in the topic assignment process. Thus, these additional topics are removed by the LASSO models.

The AUC values reach their maximum points (highlighted in bold) when using 50 to 800 topics for the studied systems. In particular, four out of the six systems reach their maximum AUC values when using 300 topics or less. The LASSO models that leverage both the baseline metrics and topic-based metrics that are derived from 300 topics achieve an 3% to 13% improvement of AUC over the LASSO models that only leverage the baseline metrics.

Table 4.18 also shows the Cliff's δ effect sizes of comparing the performance of the models that only use the baseline metrics versus the performance of the models that use both the baseline metrics and the topic-based metrics. Using 20 or 50 topics improves the AUC of the baseline models with a medium effect size; using 100 or more topics improves the AUC of the baseline models with a large effect size.

The impact of filtering out small methods. In this chapter, we filtered out small methods for each studied system (Section 4.4.2), as intuitively small methods usually implement simple functionalities (e.g., getters and setters) and are less likely to need logging statements. We now examine the effect of filtering out small methods on our models. Table 4.19 shows the performance of the LASSO models without the filtering process. Without filtering out small methods, both the models that leverage baseline metrics and the models that leverage baseline and topic-based metrics have better performance in terms of AUC and BA. Yet the topic-based metrics still bring a 1% to 7% improvement on AUC and a 4% to 14% improvement on BA, over the baseline metrics, for the LASSO models. The AUC improvement has an effect size of 0.53 (large) and the BA improvement has an effect size of 0.72 (large), both of which are statistically significant.

However, the additional explanatory power (i.e., 1% to 7% improvement on AUC and 4% to 14% improvement on BA) is smaller than it is when a filtering process is applied (i.e., 3% to 13% improvement on AUC and 6% to 16% improvement on BA). These results can be explained by the fact that the filtered small methods are much less likely to have logging statements. Taking the *Hadoop* system for example, the filtered small methods make up 60% of all the methods, but they only contain 5% of all the logged methods. The structural metrics (e.g., LOC) can simply be used to predict such small methods as being not logged. In other words, topic-based metrics are less likely to bring additional explanatory power to the small methods. However, such methods

Table 4.19: Performance of the LASSO models (without filtering out small methods), evaluated by AUC and BA.

Project	Baseline metrics		Baseline + Topics	
	AUC	BA	AUC	BA
Hadoop	0.92	0.81	0.94 (+2%)	0.84 (+4%)
Directory-Server	0.89	0.78	0.95 (+7%)	0.89 (+14%)
Qpid-Java	0.89	0.79	0.93 (+4%)	0.84 (+6%)
Camel	0.92	0.83	0.93 (+1%)	0.86 (+4%)
CloudStack	0.95	0.82	0.96 (+1%)	0.89 (+9%)
Airavata	0.97	0.92	0.99 (+2%)	0.97 (+5%)
Cliff's δ	-	-	0.53 (large)	0.72 (large)
P-value (Wilcoxon)	-	-	0.02 (sig.)	0.02 (sig.)

are far less likely to be logged.

Our LASSO models that combine baseline metrics and topic-based metrics achieve an AUC of 0.87 to 0.99 and a BA of 0.78 to 0.95. The topic-based metrics provide an AUC improvement of 3% to 13% and a BA improvement of 6% to 16%, over the baseline metrics. The topics-based metrics play important roles in the LASSO models; in particular, the log-intensive topics have a strong and positive relationship with the likelihood of a method being logged.

4.6 Threats to Validity

External Validity. Different systems are concerned with different topics. The discussions on the specific topics in this chapter may not be generalized to other systems. Findings from additional case studies on other systems can benefit our study. However, through a case study on six systems that are of different domains and sizes, we expect that our general findings (i.e., the answers to the research questions) can stand for other systems. We believe that developers can leverage the specific topics in their own systems to help understand and guide their logging decisions.

Our study focused on the source code (i.e., production code) of the studied systems and excluded the testing code. We are more interested in the production code because the logging in the source code directly impacts the customer's experience about the performance and diagnosability of a system. On the other hand, testing code is mainly used for in-house diagnosis, and the impact of logging is usually less of a concern. However, it is interesting to study the differences between the logging statements in the production code and the testing code. We expect future studies to explore the differences between production code logging and testing code logging.

Internal Validity. The regression modeling results present the relation between the likelihood of a method being logged and a set of software metrics. The relation does not represent the casual effects of these metrics on the likelihood of a method being logged.

In RQ3, we used 14 structural metrics to form the baseline of our models. The selected metrics do not necessarily represent all the structural information of a method. However, we used both the general information (e.g., LOC and CCN) and the detailed information (e.g., the number of if-statements and the number of catch blocks), trying to cover a large spectrum of structural information about a method.

In this chapter, we studied the relationship between logging decisions and the underlying topics in the software systems. Our study was based on the assumption that the logging practices of these projects are appropriate. However, the logging practices of these projects may not always be appropriate. In order to avoid learning bad practices, we chose several successful and widely-used open source systems.

Construct Validity. Interpreting LDA-generated topics may not always be an easy task (Hindle et al., 2015), and the interpretation may be subjective. Thus, the author of the thesis tried to first understand the topics and derive topic labels, and another researcher (i.e., a collaborator) validated the labels. In case a topic that is hard to interpret, we study the source code (i.e., both classes and methods) that are related to the topic.

As suggested by prior studies (Wallach et al., 2009; Chen et al., 2016b), we chose 500 topics for the topic modeling of individual systems in RQ1. However, determining the appropriate number of topics to be used in topic modeling is a subjective process. As our primary purpose of using topic models is for interpretation, the appropriateness of a choice of topic number should be determined by how one plans to leverage the resulting topics for interpreting the meaning of the source code. We found that using 500 topics for each studied system provides reasonable and tractable results for us to interpret the generated topics. Besides, we discuss how the different numbers of topics influence the observations of each RQ.

When running LDA, we applied MALLET's hyper-parameter optimization to automatically find the optimal α and β values. However, the optimization heuristics are designed for natural language documents instead of source code files. As the source code is different from natural language, we may not get the optimal topics. Future in-depth studies are needed to explore this wide-ranging concern across the multitude of uses of LDA on software data (Chen et al., 2016b).

Topic models create automated topics that capture the co-occurrences of words in methods. However, one may be concerned about the rationale of studying the logging practices using topics instead of simply using the words that exist in a method. We use topics instead of words for two reasons: 1) topic models provide a higher-level overview and interpretable labels of a code snippet (Blei et al., 2003; Steyvers and Griffiths, 2007); 2) and using words in a code snippet to model the likelihood of a code snippet being logged is very computationally expensive and the resulting model is more likely to over-fit. Our experiments show that there are 2,117 to 5,474 different words (excluding English stop words and programming language keywords) in our studied systems, hence one would need to build a very expensive model (2,117 to 5,474 metrics) using these words. Our experiments also show that using 2,117 to 5,474 words as explanatory variables provides 3% to 10% (with a median of 4%) additional explanatory power (in terms of AUC) to the baseline models. In comparison, using only 300 topics as explanatory variables provides 3% to 13% (with a median of 6%) additional explanatory power to the baseline models.

4.7 Related Work

4.7.1 Applying Topic Models on Software Engineering Tasks

Topic models are widely used in the Software Engineering research for various tasks (Chen et al., 2016b; Sun et al., 2016), such as concept location (Cleary et al., 2008; Poshyvanyk et al., 2007; Rao and Kak, 2011), traceability linking (Asuncion et al., 2010), understanding software evolution (Thomas et al., 2011; Hu et al., 2015), code search (Tian et al., 2009), software refactoring (Bavota et al., 2014), and software maintenance (Sun et al., 2015a,b). Recent studies explored how to effectively leverage topic models in software engineering tasks (Panichella et al., 2013, 2016). However, there is no study of software logging using topic models (Chen et al., 2016b). Some prior studies (Chen et al., 2012; Nguyen et al., 2011) successfully show that topics in source code are correlated to some source code metrics (e.g., quality). Thus in this chapter, we followed up on that intuition and we studied the relationship between code topics and logging decisions.

Prior studies (De Lucia et al., 2012, 2014) also found that most LDA-generated topics are easy for developers to understand, and these topics can be useful for developers to get a high-level overview of a system (Thomas et al., 2011). In this chapter, we also conducted a manual study on the topics, and our study provides a high-level overview of which topics are more likely to need logging statements in our studied systems.

4.8 Chapter Summary

Inserting logging statements in the source code appropriately is a challenging task, as both logging too much and logging too little are undesirable. We believe that the code snippets of different topics have different logging requirements. In this chapter, we used LDA to extract the underlying topics from the source code, and studied the relationship between the logging decisions and the recovered topics. We found that a small number of topics, in particular, the topics that can be generalized to communication between machines or interaction between threads, are much more likely to be logged than other topics. We also found that the likelihood of logging the common topics has a significant correlation across all the studied systems, thus developers of a particular system can consult other systems when making their logging decisions or developing logging guidelines. Finally, we leveraged the recovered topics in regression models to provide additional explanatory power for the likelihood of a method being logged. Our case study on six open source software systems suggests that topics can statistically help explain the likelihood of a method being logged.

As code topics contain valuable information that is correlated with logging decisions, topic information should be considered in the logging practices of practitioners when they wish to allocate limited logging resources (e.g., by allocating more logging resources to log-intensive topics). Future work on logging recommendation tools should also consider topic information in order to help software practitioners make more informed logging decisions. Furthermore, our findings encourage future work to develop topic-influenced logging guidelines (e.g., which topics need further logging), in addition to the best practices and logging advice that are derived in Chapter 3.

CHAPTER 5

Automated Suggestions for Log Changes

As we find in Chapter 2, prior approaches automatically enhance logging statements as a post-implementation process. Such automated approaches do not take into account developers' domain knowledge and concerns that we discuss in Chapter 3; nevertheless, developers usually need to carefully design the logging statements since logs are a rich source about the field operation of a software system. This chapter empirically studies why developers make log changes and proposes an automated approach to provide developers with log change suggestions as soon as they commit a code change. In particular, we derive a set of measures based on manually examining the reasons for log changes and our experiences. We use these measures as explanatory variables in random forest classifiers to model whether a code commit requires log changes. We perform a case study on four open source projects, and we find that: (i) The reasons for log changes can be grouped along four categories: block change, log improvement, dependence-driven change, and logging issue; (ii) our random forest classifiers can effectively suggest whether a log change is needed: the classifiers that are trained from within-project data achieve a balanced accuracy of 0.76 to 0.82, and the classifiers that are trained from cross-project data achieve a balanced accuracy of 0.76 to 0.80; (iii) the characteristics of code changes in a particular commit and the current snapshot of the source code are the most influential factors for determining the likelihood of a log change in a commit.

An earlier version of this chapter is published in the Empirical Software Engineering Journal (EMSE) (Li et al., 2017b).

5.1 Introduction

Logs are generated at runtime from logging statements in the source code. Logs record valuable run-time information. A logging statement, as shown below, typically specifies a verbosity level (e.g., debug/info/warn/error/fatal), a static text and one or more variables (Fu et al., 2014; Yuan et al., 2012b; Gülcü and Stark, 2003).

logger.error("static text" + variable);

Logs help software practitioners better understand the behaviors of large scale software systems and assist in improving the quality of the systems (Fu et al., 2013; Shang et al., 2014b). Software operators leverage the rich information in logs to guide capacity planning efforts (Sharma et al., 2011; Kavulya et al., 2010), to monitor system health (Bitincka et al., 2010), and to identify abnormal behaviors (Fu et al., 2009; Syer et al., 2013; Xu et al., 2009b). Besides, software developers rely on logs for debugging field failures (Glerum et al., 2009; Yuan et al., 2010). In recent years, the broad usage of logs led to the emergence a new market of *Log Processing Applications* (LPAs) (e.g., Splunk[1] (Bitincka et al., 2010), XpoLog[2], and Logstash[3]), which support the collection, storage, search, and analysis of large amounts of log data.

However, appropriate logging is difficult to reach in practice. Both logging too little and logging too much is undesirable (Fu et al., 2014; Chapter 3). Logging too little may result in the lack of runtime information that is crucial for understanding and diagnosing software systems (Yuan et al., 2010; Chapter 3); while logging too much may lead to runtime overhead and costly log maintenance efforts (Fu et al., 2014; Chapter 3).

[1]Splunk. http://www.splunk.com/
[2]XpoLog. http://www.xpolog.com/
[3]Logstash. http://logstash.net/

Prior work shows that developers spend a large amount of effort for maintaining logging statements, and 33% of the log changes are introduced as after-thoughts (i.e., as follow-up changes instead of being done when the actual surrounding code is being changed) (Yuan et al., 2012b).

Figure 5.1 shows a code snippet that is taken from Hadoop revision 1240413. We use “svn blame”[4] to show the contributing commit of each code line. The code snippet shows that the commit 1190122 added a *try-catch* statement, and an *if* statement in the *catch* clause. In a later commit 1240413, which contributed to a bug fix, the developer added an error logging statement to record important runtime information, in order to help fix bug MAPREDUCE-3711[5]. The information that is recorded by the logging statement even helps understand the execution of the system when fixing bugs in the future (e.g., MAPREDUCE-5501[6] and MAPREDUCE-6317[7]). Suppose that the developer was suggested to add the logging statement in the earlier commit (1190122) with the try-catch statement, the logged information would help developers and operators understand the system's behavior when they are fixing bug MAPREDUCE-3711. In this chapter, we propose an approach that can automatically provide developers with such suggestions for log changes when they commit new code changes.

Log enhancement approaches, such as *Errlog* (Yuan et al., 2012a) and *LogEnhancer* (Yuan et al., 2011), aim to improve software failure diagnosis by automatically adding

[4]svn blame. `http://svnbook.red-bean.com/en/1.7/svn.ref.svn.c.blame.html`
[5]`https://issues.apache.org/jira/browse/MAPREDUCE-3711`
[6]`https://issues.apache.org/jira/browse/MAPREDUCE-5501`
[7]`https://issues.apache.org/jira/browse/MAPREDUCE-6317`

```
/* A code snippet taken from Hadoop revision 1240413, in file:
 * hadoop-mapreduce-project/hadoop-mapreduce-client/
    hadoop-mapreduce-client-app/src/main/java/org/apache/
    hadoop/mapreduce/v2/app/rm/RMContainerAllocator.java
 * Commit 1240413 is part of a patch to fix a bug with JIRA issue ID
    MAPREDUCE-3711
 */
1190122 try {
1190122   response = makeRemoteRequest();
1190122   retrystartTime = System.currentTimeMillis();
1190122 } catch (Exception e) {
1190122   if (System.currentTimeMillis() - retrystartTime >=
   retryInterval) {
1240413     LOG.error("Could not contact RM after " + retryInterval +
1240413               " milliseconds.");
1190122     eventHandler.handle(new JobEvent(this.getJob().getID(),
1190122                                  JobEventType.INTERNAL_ERROR));
1190122     throw new YarnException("Could not contact RM after " +
1190122                             retryInterval + " milliseconds.");
1190122   }
1190122   throw e;
1190122 }
```

Figure 5.1: An “svn blame” example showing that a later commit (1240413) added a logging statement that was missing in an earlier commit (1190122).

more logged information to the existing code, as a post-implementation process. However, these automatic log enhancement approaches never take into account developers’ domain knowledge and concerns that we discuss in Chapter 3. In practice, developers need to carefully design logging statements since logs contain valuable information for both software developers and operators (Yuan et al., 2012b).

Recent studies investigate where developers insert logging statements (Fu et al., 2014) and automatically suggest locations in need of logging statements (Zhu et al., 2015). In particular, the authors conducted source code analysis to investigate the types of code snippets (e.g., catch block) in which developers often insert logging statements. The study provides post-coding guidelines for inserting logging statements into the source code. However to the best of our knowledge, there exists no studies to guide developers during coding, i.e., providing guidance about whether to change (add, delete or modify) logging statements when developers are committing code changes.

In this chapter, we propose an approach that can provide just-in-time suggestions as to whether a log change is needed when a code change occurs. The term "just-in-time" is based on prior research by Kamei et al. (2013) that advocates the benefits of providing suggestions to developers at commit time. Follow-up studies (Kamei et al., 2016; Tourani and Adams, 2016; Fukushima et al., 2014) also use the term "just-in-time" to describe commit-time suggestions or alerts. In this chapter, we leverage prior commits to build classifiers in order to suggest whether log changes are needed for a new commit. We perform a case study on four open source systems (*Hadoop*, *Directory Server*, *Commons HttpClient*, and *Qpid*), to answer the following three research questions:

RQ1: What are the reasons for changing logging statements?

Through a manual analysis of a statistically representative sample of logging statements, we find that the reasons for log changes can be grouped along four categories: block change, log improvement, dependence-driven change, and logging issue.

RQ2: How well can we provide just-in-time log change suggestions?

We build random forest classifiers using software measures that are derived from our manual study in RQ1, and from our experience, in order to model the drivers for log changes in a code commit. We evaluate our classifiers in both a within-project and a cross-project evaluation. For our within-project evaluation, we build a random forest classifier for every code commit using all previous code commits as training data, in order to suggest whether a log change is needed for the current commit. The random forest classifiers that are built from historical data from the same project achieve a balanced accuracy of 0.76 to 0.82. For our cross-project evaluation, we build random forest classifiers that are trained from three out of four studied projects and suggest log changes in the remaining project. We repeat the process for each of the studied projects. The classifiers reach a balanced accuracy of 0.76 to 0.80 and an AUC of 0.84 to 0.88.

RQ3: What are the influential factors that explain log changes?

Factors which capture characteristics about the changes to the non-logging code in a commit (i.e., change measures, such as the number of changed control flow statements) and factors that capture characteristics of the current snapshot of the source code (i.e., product measures, such as the number of existing logging statements) are the most influential factors for explaining log changes in a commit. In particular, change measures are the most influential explanatory factors for log additions, while product measures are the most influential explanatory factors for log modifications.

Chapter organization. The remainder of the chapter is organized as follows. Section 5.2 describes the studied systems and our experimental setup. Section 5.3 explains the approaches that we used to answer the research questions and presents the results of our case study. Section 5.4 discusses the characteristics of commits that only change logging statements without changing the non-logging code. Section 5.5 discusses the threats of validity. Finally, Section 5.6 draws conclusions based on our presented findings.

5.2 Case Study Setup

This section describes the subject systems and the process that we used to prepare the data for our case study.

5.2.1 Subject Systems

This chapter studies the reasons for log changes and explores the feasibility of providing accurate just-in-time suggestions for log changes through a case study on four open source projects: *Hadoop*, *Directory Server*, *Commons HttpClient*, and *Qpid*. All the four projects are mature Java projects with years of development history and from different domains. Table 5.1 shows the studied development history for each project. We use the “svn log”[8] command to retrieve the development history for each project (i.e., the svn commit records). We analyze the development history of the main branch (trunk) of each project, and focus on Java source code (excluding Java test code). Some commits import a large number of atomic commits from a branch into the trunk (a.k.a. merge commits), which usually contain a large amount of code changes and log changes. Such merge commits would introduce noise in our study (Zimmermann et al., 2004; Hassan and Holt, 2004; Hassan, 2008) of log changes in a commit. We unroll each merge commit into the various commits of which it is composed (using the “use-merge-history” option of the “svn log” command).

[8]svn log. `http://svnbook.red-bean.com/en/1.7/svn.ref.svn.c.log.html`

Table 5.1 also presents an overview of the studied systems. The source lines of code (SLOC) of each project is measured at the end of the studied development history. The *Hadoop* project is the largest project. It has 458K lines of source code, while *HttpClient* is the smallest project, with an SLOC of 18K. We study 5,401, 4,968, 949 and 3,538 commits for Hadoop, DirectoryServer, HttpClient and Qpid, respectively. These commits include all the commits in the studied development history that change at least one Java source code file. Table 5.1 also shows the numbers and percentages of commits that change (i.e., add, modify, or delete) logging statements, for each project. 30.0% (1,621 out of 5,401) of *Hadoop*'s commits are accompanied with log changes, while the percentage of commits that change logging statement ranges from 22.7% to 26.6% for *Directory Server*, *HttpClient*, and *Qpid*. The last column of Table 5.1 lists the number of log changes (a log change is an occurrence of either adding, deleting, or modifying a logging statement) that occurred during the studied development history. The DirectoryServer project has the most log changes within the studied history. We provide our dataset[9] for all four studied projects for better replication.

5.2.2 Data Extraction

Figure 5.2 presents an overview of our data extraction and data analysis approaches. From the version control repositories of each subject system, we analyze the code changes in each commit and identify the commits that contain changes to logging statements. As a result, we are able to create a log-change database (i.e., a collection of log changes) and label each commit as to whether it contains log changes or not. The log-change database is used in our manual analysis (RQ1), and the labeled commit

[9] `http://sailhome.cs.queensu.ca/replication/JITLogSuggestions/dataset.zip`

Table 5.1: Overview of the studied systems.

Project	#SLOC	Studied history	#Commits	#Log-changing commits	#Log changes
Hadoop	458 K	2009-05-19 to 2014-07-02	5,401	1,621 (30.0%)	9,503
Directory Server	119 K	2006-01-03 to 2014-06-30	4,968	1,130 (22.7%)	11,883
Http- Client	18 K	2001-04-25 to 2012-12-16	949	252 (26.6%)	2,333
Qpid	271 K	2006-09-19 to 2014-07-01	3,538	908 (25.7%)	8,761

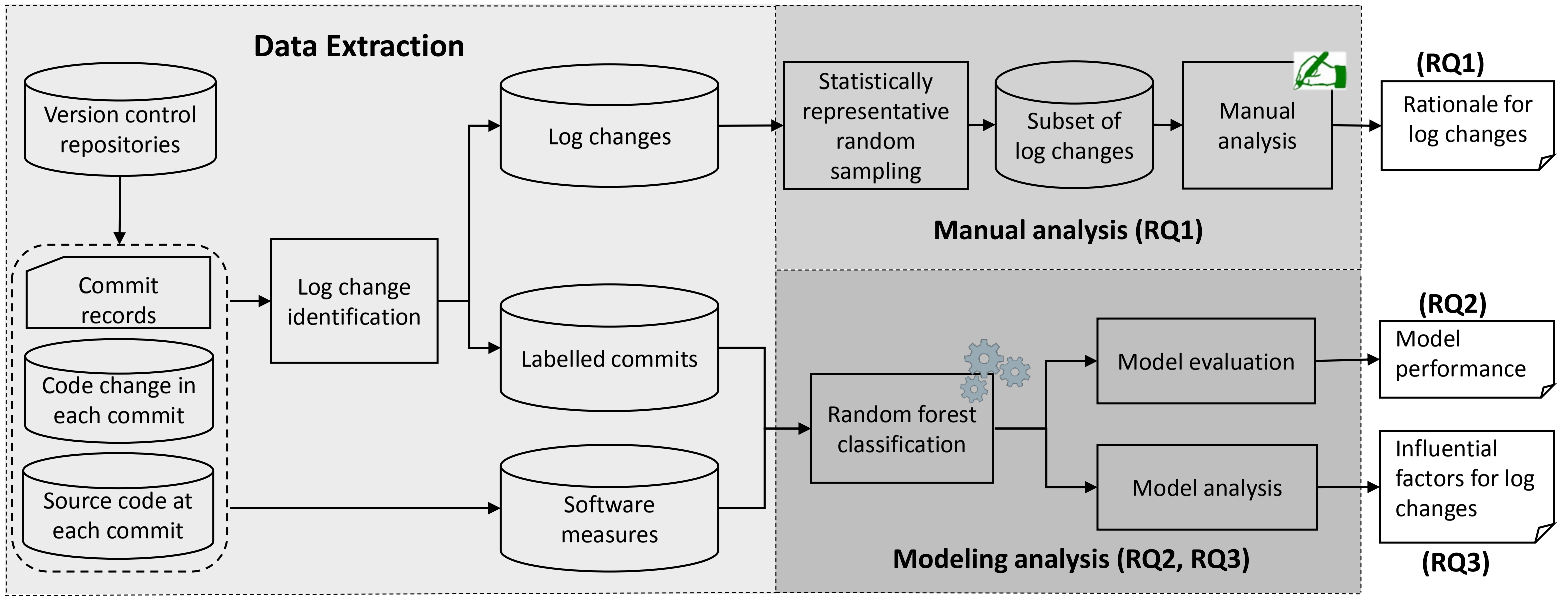

Data Extraction
Version control repositories
Commit records
Code change in each commit
Source code at each commit
Log change identification
Log changes
Labelled commits
Software measures
Statistically representative random sampling
Subset of log changes
Manual analysis
Manual analysis (RQ1)
Random forest classification
Model evaluation
Model analysis
Modeling analysis (RQ2, RQ3)
(RQ1)
Rationale for log changes
(RQ2)
Model performance
Influential factors for log changes
(RQ3)

Figure 5.2: An overview of our data extraction and analysis approaches.

data is employed in our modeling analysis (RQ2 and RQ3).

Within the commits that change logging statements, there are only 1.2% to 4.2% of them that do not change other source code (i.e., log-changing-only commits). Since our models aim to provide developers with just-in-time suggestions for log changes when they are changing other source code, we exclude these log-changing-only commits in our modeling analysis. We revisit the characteristics of the log-changing-only commits at the end of the chapter, in Section 5.4.

5.2.3 Log Change Identification

In order to represent the term *log change* more accurately, we define the following four terms:

- **Log addition**, measures the new logging statements that are added in a commit.
- **Log deletion**, measures the obsolete logging statements that are deleted in a commit.
- **Log modification**, measures the existing logging statements that are modified in a commit.
- **Log change**, measures any kind of change (addition, deletion, and modification) that is made to logging statements in a commit.

The studied projects leverage standard logging libraries (e.g., Apache Commons Logging[10], Log4j[11] and SLF4J[12]) for logging. The usage of the standard libraries brings uniform formats (e.g., *logger.error(message)*) to the logging statements, thus we can accurately identify the logging statements.

[10]http://commons.apache.org/proper/commons-logging
[11]http://logging.apache.org/log4j/2.x
[12]http://www.slf4j.org

We use regular expressions to identify the added and deleted logging statements across commits (see on-line replication package for the used regular expressions[13]). If a pair of an added logging statement and a deleted one are within the same code snippet and they are textually similar to each other, the pair of logging statements are considered as a log modification. Otherwise they are considered as one log addition and one log deletion. We measure the textual similarity between two logging statements by calculating the Levenshtein distance ratio (Levenshtein, 1966) between their concatenation of static text and variable names. Two logging statements are considered similar if the Levenshtein distance ratio between them is larger than a specified threshold for which we choose 0.5 in this chapter (see Section 5.5 for a sensitivity analysis of the impact of this threshold on the identification of log modifications).

5.3 Case Study Results

In this section, we present the results of our research questions. For each research question, we present the motivation of the research question, the approach that we used to address the research question, and our experimental results.

5.3.1 RQ1: What are the reasons for changing logging statements?

Motivation

Before proposing an approach that can provide just-in-time suggestions for log changes, we first conduct a manual study in order to investigate the reasons for changing logging statements. Our manual observation will assist us in defining

[13] `http://sailhome.cs.queensu.ca/replication/JITLogSuggestions/log_change_regex.zip`

appropriate measures that we can use later on to build models to provide just-in-time suggestions for log changes when developers commit code changes.

Approach

There is a total of 32,480 logging statement changes in the studied commits of the four studied projects (9,503 for Hadoop, 11,883 for DirectoryServer, 2,333 for HttpClient and 8,761 for Qpid). Each commit may contain multiple logging statement changes. We randomly selected a statistically representative sample (95% confidence level with a ±5% confidence interval) of 380 log changes. Among the 380 log changes, there are 204 log additions, 91 log modifications, and 85 log deletions. We manually examine the possible reasons for these log changes. For each log change, we check the log change itself, the co-changed code, the commit message, and the associated issue report if an issue id is noted in the commit message. Certain log change reasons (e.g., a typo) can be detected by only looking at the log change itself. Examining the co-changed code can help us determine the log change reasons such as "a logging statement is changed because the logged variables are changed". The commit message and the issue report directly communicate the intention of the developer and the issue owner for a log change. Two researchers including the author of the thesis and a collaborator work together by manually examining all log changes from the random sample. We examine the log change, code change, commit message and the associated issue report to understand the reason of a log change. If the reason is new, we add it to the list of identified reasons. If there is a disagreement during the process, the two authors discuss and reach a consensus.

Table 5.2: Log-change reasons and the distribution: manual analysis result.

Reason Category	Log Change Reason	Log Change Number	Log Change Type	Total Log Change Number
block change	adding/deleting try-catch block	80	add,delete	260
	adding/deleting method	69	add,delete	
	adding/deleting branch	52	add,delete	
	adding/deleting if-null branch	49	add,delete	
	adding/deleting loop	10	add,delete	
log improvement	improving debugging capability	19	add,modify	63
	improving readability	13	add,modify	
	leveraging message translation	11	modify	
	improving runtime information	9	add,modify	
	redundant log information	6	delete	
	log library migration	4	modify	
	security issue	1	delete	
dependence-driven change	logger change	20	modify	39
	variable change	14	modify	
	method change	2	modify	
	class change	2	modify	
	dependence removal	1	modify	
logging issue	inappropriate log level	13	modify	18
	inappropriate log text	4	modify	
	incorrect message translation	1	modify	

Results

We find 20 reasons for log changes across four categories: changing context code, improving logging, dependency-driven changes and fixing logging issues. Table 5.2 summarizes the log change reasons. We present below the four categories of reasons for log changes.

Block change. Logging statements are added (or deleted) as a result of the change of the surrounding code blocks. According to our manual analysis, logging statements are added (or deleted) when developers are adding (or deleting) try-catch blocks, adding (or deleting) methods, adding (or deleting) branches (if branches and switch

```
/* Project: DirectoryServer; Commit: 664015
 * File: directory/apacheds/branches/bigbang/core-integ/src/main/java/
     org/apache/directory/server/core/integ/state/NonExistentState.java
 */
+ try
+ {
+     create( settings );
+ }
+ catch ( NamingException ne )
+ {
+     LOG.error( "Failed to create and start new server instance: " + ne );
+     notifier.testAborted( settings.getDescription(), ne );
+     return;
+ }
```

Figure 5.3: An example of log changes with the reason category *block change*.

branches), adding (or deleting) if-null branches (if branches checking an abnormal condition), and adding (or deleting) loops (for loops and while loops). For example, the code snippet shown in Figure 5.3 indicates that a logging statement is added to record the error information as part of the newly added try-catch block. (Note: the plus sign (+) or minus sign (-) leading a code line indicates that the code line is added or deleted in that particular commit.)

Log improvement. Logging statements are added, deleted or modified to achieve a better logging practice. Developers change logging statements (e.g., by adding a logging statement which tracks the value of a variable) to improve the debugging capability of the logged information. They also change a logging statement to improve the readability of the logged information; for example, they rephrase a logging statement such that the log message would be easier to understand. Some logging statements are changed to leverage log message translation method (i.e., using predefined code such as “I18n.ERR_115” to represent a log message). Developers also change logging

```
/* Project: HttpClient; Commit: 159615
 * File: /jakarta/commons/proper/httpclient/trunk/src/java/
    org/apache/commons/httpclient/HttpMethodDirector.java
 * Commit message: "Some extra debug log entries for the authenticaton
    process".
 */
  private Credentials promptForProxyCredentials(
      final AuthScheme authScheme,
      final HttpParams params,
      final AuthScope authscope)
  {
+     LOG.debug("Proxy credentials required");
      /* other operations */
  }
```

Figure 5.4: An example of log changes with the reason category *log improvement*.

statements, for example, by adding a logging statement to record the occurrence of an event, to improve the logged runtime information. Removing redundant log information is another way to improve the logging of a system; the redundant log information includes duplicated log information and unnecessary log information. Developers sometimes improve their logging by migrating from an old logging style (e.g., “System.out”) to a more advanced logging library (e.g., Log4j) (i.e., log library migration (Kabinna et al., 2016a)). Finally, we also find that a logging statement is removed because of a security issue that is mentioned in the associated issue report. Figure 5.4 shows that a logging statement is added to a method in order to enhance the debugging capability. The commit message states that the developer added “some extra debug log entries for the authentication process”.

Dependence-driven change. Logging statements are changed because they depend on other code elements (e.g., variables) that are changed by developers. A log change might be driven by the change of a logger (i.e., an class object that is used to

```
/* Project: Hadoop; Commit: 1308205
* File: hadoop/common/trunk/hadoop-hdfs-project/hadoop-hdfs/src/main/java/
    org/apache/hadoop/hdfs/DFSClient.java
* Commit message: "HDFS-3144. Refactor DatanodeID#getName by use".
* Issue report (HDFS-3144): "DataNodeID#getName is no longer available.
    The following are introduced so each context in which we use the
    "name" has it's own method: toString - for logging".
*/
- LOG.debug("write to " + datanodes[j].getName() + ": "
+ LOG.debug("write to " + datanodes[j] + ": "
      + Op.BLOCK_CHECKSUM + ", block=" + block);
```

Figure 5.5: An example of log changes with the reason category *dependence-driven change*.

invoke a logging method), a variable, a method or a class. We also find that a logging statement is changed to remove its dependence to a different module to remove the coupling between modules. The example in Figure 5.5 shows that a logging statement is modified because the method ("DatanodeID:getName") that it depended on has been replaced by a new method ("toString"). The reason of the log changes is recorded in the commit message and the associated issue report[14].

Logging issue. Logging statements are modified because issues (e.g., defects) are discovered in the existing logging statements. Some logging statements are modified due to an inappropriate log level. Some logging statements are modified because the old logging statement has an inappropriate log text (e.g., a typo). We also find a log change which is caused by an incorrect log message translation. In the example shown in Figure 5.6, the level of a logging statement is downgraded from *info* to *debug* because the *info* level caused too much noise, as noted in the commit message.

The manually identified reasons for log changes assist us in defining measures

[14]https://issues.apache.org/jira/browse/HDFS-3144

```
/* Project: Qpid; Commit: 1298555
* File: qpid/trunk/qpid/java/client/src/main/java/
   org/apache/qpid/client/BasicMessageConsumer.java
* Commit message: "it reduces noise by downgrading most log messages from
info to debug".
*/
  public void close(boolean sendClose) throws JMSException
  {
-     if (_logger.isInfoEnabled())
+     if (_logger.isDebugEnabled())
      {
-         _logger.info("Closing consumer:" + debugIdentity());
+         _logger.debug("Closing consumer:" + debugIdentity());
      }
      /* other operations */
  }
```

Figure 5.6: An example of log changes with the reason category *logging issue*.

to model the drivers for log changes. The log change reasons in the *block change* category motivate us to consider measures that capture the changes in the commit itself. These measures may include the number of changed method declarations, try-catch, if/if-null, and for/while statements in a commit. The log change reasons from the *dependence-driven change* category also suggest us to consider measures that capture the changes in the commit itself, since the code elements that a logging statement depends on might get changed in the commit. The log change reasons from the categories of *log improvement* and *logging issue* suggest that we should consider measures that capture the current snapshot of the source code, such as log density, number of logs, average log length, average log level, average number of log variables and complexity measures. The log change reasons from the *dependence-driven change* category also motivate us to consider measures that capture the current snapshot of the source code, as logging statements with higher dependence on other source code (e.g., more log variables) are more likely to be changed.

We find four categories of log change reasons: block change, log improvement, dependence-driven change, and logging issue. The log change reasons give us valuable insight for defining measures to model the drivers for log changes.

5.3.2 RQ2: How well can we provide just-in-time log change suggestions?

Motivation

We want to provide developers with just-in-time suggestions on whether a log change (log addition, log deletion, or log modification) is needed when they are changing the code. We need a classifier that can tell whether a code commit should contain log changes. By evaluating the accuracy of the classifier, we can understand whether developers can depend in practice on the suggestions that can be provided by our approach.

Approach

We use random forest classifiers to provide just-in-time suggestions for log changes. A random forest classifier models a **binary response variable** which measures the likelihood of a log change occurring in a particular code commit.

In order to model the drivers for log changes, we extract and calculate a set of measures from three dimensions: *change measures*, *historical measures*, and *product measures*. Table 5.3 presents a list of measures that we collect for each dimension. Table 5.3 also describes our proposed measures and explains our motivation behind each measure. We build classifiers at the granularity of a code commit, thus we calculate all of

Table 5.3: Software measures used to model the drivers for log changes, measured per each commit.

Dimension	**Measures**	**Definition (d) | Rationale (r)**
Change measures	class declaration	d: Number of changed class declarations in the commit.
		r: Developers might add logging statements in a new class so that they can better observe the behavior of the class.
	method declaration	d: Number of changed method declarations in the commit.
		r: Developers might add logging statements in a new method so that they can better observe the behavior of the method.
	try statement	d: Number of changed *try* statements in the commit.
		r: Logging statements often reside inside *try* blocks; hence logging statements are likely to co-change with *try* statements.
	catch clause	d: Number of changed *catch* clauses in the commit.
		r: Exception catching code is often logged (Yuan et al., 2012b; Fu et al., 2014; Zhu et al., 2015); hence logging statements are likely to co-change with *catch* clauses.
	throw statement	d: Number of changed *throw* statements in the commit.
		r: A logging statement is often inserted right before a *throw* statement (Fu et al., 2014); hence developers changing a *throw* statement are likely to change the corresponding logging statement.
	throws clause	d: Number of method definitions with *throws* clauses (which declare that a method can throw exceptions) changed in the commit.
		r: Methods that throw exceptions are likely to have logging statements; thus logging statements might co-change with *throws* clauses.
	if statement	d: Number of changed *if* statements in the commit.
		r: Logging statements are usually inside *if* branches (Fu et al., 2014; Zhu et al., 2015); thus logging statements are likely to co-change with *if* statements.
	if-null statement	d: Number of changed *if-null* statements (if statements with null condition, e.g., “if (outcome == NULL)”) in the commit.
		r: *if-null* branches are usually corner-case execution paths which are likely to be logged (Fu et al., 2014; Zhu et al., 2015); thus logging statements might co-change with *if-null* blocks.
	else clause	d: Number of changed *else* clauses in the commit.
		r: Logging statements are usually inside *if-else* branches (Fu et al., 2014; Zhu et al., 2015); thus logging statements are likely to co-change with *else* clauses.
	for statement	d: Number of changed *for* statements in the commit.
		r: Logging statements inside *for* loops usually record the execution path or status of the *for* loops; hence these logging statements are likely to co-change with the *for* statements.
	while statement	d: Number of changed *while* statements in the commit.
		r: Logging statements inside *while* loops usually record the execution path or status of the *while* loops; hence these logging statements are likely to co-change with the *while* statements.
	commit type	d: Change type of the commit: Bug/Improvement/New Feature/Task/Sub-task/Test.
		r: Change type characterized the context of a code change, thus it might affect developers' logging behavior.
Historical measures	log churn in history	d: Number of changed logs in the development history of the involved files.
		r: Files experiencing frequent log changes in the past might expect frequent log changes in the future.
	log churn ratio in history	d: Ratio of the number of changed logging statements to the number of changed lines of code in the development history of the involved files.
		r: Files experiencing frequent log changes in the past are likely to exhibit frequent log changes in the future.

Dimension	Measures	Definition (d) | Rationale (r)
Historical measures	log-changing commits in history	d: Number of commits involving log changes in the development history of the involved files.
		r: Files experiencing frequent log changes in the past are likely to exhibit frequent log changes in the future.
	code churn in history	d: Number of changed lines of code in the development history of the involved files.
		r: Frequently changed code are problem-prone thus are more likely to be logged.
	commits in history	d: Number of commits in the development history of the involved files.
		r: Frequently changed code are problem-prone thus are more likely to be logged.
Product measures	log number	d: The number of logging statements in the files that are involved in the commit.
		r: Code snippets with more logging statements are more likely to require frequent log changes.
	log density	d: The density of logging statements in the files that are involved in the commit, calculated by dividing the total number of logging statements by the lines of source code across all the involved files.
		r: Issues with the existing logging statements might cause log changes. Thus code snippets with denser logging statements are more likely to require log changes.
	average log length	d: Average length of the existing logging statements in the changed files.
		r: Longer logging statements are more likely to require continuous maintenance.
	average log level	d: Average level of the existing logging statements in the changed files. Obtained by quantifying the log levels into integers and calculating the average.
		r: Logs with a lower verbosity level might get changed more often since they are more likely to be used for debugging.
	average log variables	d: Average number of variables in the existing logging statements in the changed files.
		r: Logs with more variables are likely more coupled with the code, hence they may be changed more often.
	SLOC	d: Number of source lines of code in the changed files.
		r: Large source files are likely to have more logging statements, thus they get more chances for log changes.
	McCabe complexity	d: McCabe's cyclomatic complexity of the changed files.
		r: Complex source files are likely to have more logging points, thus they are more likely to exhibit log changes.
	fan-in	d: The number of classes that depend on (i.e., reference) the changed code.
		r: Classes with a high fan-in, such as library classes, tend to have less logging statements.

our proposed measures for very commit during the studied development history. We describe below each dimension of measures:

- **Change measures** capture the changes in the commit itself, represented by the changes of control flow statements (e.g., *try statement*, *if statement*), and the type of a commit (*commit type*, Bug/Improvement/New Feature/Task/Subtask/Test). We choose the *change measures* according to our manual analysis results. As shown in the results of RQ1, most log changes are accompanied with contextual code changes (i.e., *block change* and *dependence-drive change*). For example, *adding/deleting try-catch block* is one of the most frequent reasons for a log change. The *commit type* captures the context or purpose of the code change; we use the "type" field of the JIRA issue report that is linked to the commit.
- **Historical measures** capture the code changes throughout the development history (before the considered commit) of the changed files. Based on our intuition, source code files undergoing frequent log changes in the past may have log changes in the future. Besides, prior research shows that files with high churn rate are more defect-prone (Nagappan et al., 2006; Nagappan and Ball, 2007), and developers are likely to add more logs in defect-prone source code files (Shang et al., 2015).
- **Product measures** capture the current snapshot of the source code, represented by the status of logging statements and other source code, of the software system just before the considered commit. For example, *log number* and *log density* capture the log-intensiveness of the changed code. Our manual analysis in RQ1 shows that many log changes are committed to improve existing logging

```
if (LOGGER.isDebugEnabled()) {
  LOGGER.debug("This is a debug message ");
}
```

Figure 5.7: An example of a logging guard.

(i.e., the *log improvement* reasons) or fix logging issues (i.e., the *logging issue* reasons). Thus the code changes on log-intensive code are more likely to involve log changes. In addition, based on our intuition, the appropriateness of logging statements should be related to their contextual source code. Therefore, we also calculate several product measures that capture the contextual source code of logging statements (i.e., *SLOC*, *McCabe complexity* and *fan-in*).

For the *if statement* measure from the dimension of *change measures*, we do not consider *if* statements that act as logging guards. An example of such *if* statements is shown in Figure 5.7.

Correlation analysis. Prior to constructing the classifiers for log changes, we check the pairwise correlation between our proposed measures using the Spearman rank correlation test (ρ). Specifically, we use the “varclus” function in the “Hmisc” R package to cluster measures based on their Spearman rank correlation. We choose the Spearman rank correlation method because it is robust to non-normally distributed data (McIntosh et al., 2014). In this work, we choose the correlation value 0.8 as the threshold to remove collinearity. In other words, if the correlation between a pair of measures is greater than 0.8 ($|\rho| > 0.8$), we keep one of the two measures in the classifier. We find that the measures that are listed in Table 5.3 present similar patterns of correlation across all four studied projects, thus we drop (i.e., do not consider in the classifier) the same measures for all the studied projects. Dropping the same set of measures for the projects enables us to build cross-project classifiers as discussed in the “Cross-project Evaluation” part that follows. We combine the data of the four projects together and perform correlation analysis on the combo data. Figure 5.8 shows the result of the correlation analysis on the combo data, where the horizontal bridge between each pair of measures indicates the correlation, and the red dotted line represents the threshold value (0.8 in this case). The results of our correlation analysis on each individual project can be downloaded at a public link[15]. To ease the interpretation of the classifier, we try to keep the one that is easier to understand and calculate from each pair of highly-correlated measures. For example, the *SLOC* and *McCabe complexity* measures have a correlation higher than 0.8, we keep the *SLOC* measure and drop the *McCabe complexity* measure. Based on the result shown in Figure 5.8, we drop the following measures: *log churn in history*, *log-changing commits in history*, *McCabe complexity*, *commits in history*, *log churn ratio in history*, *try statement* and *if-null statement*, as they are highly correlated with other measures.

Modeling technique. We build random forest classifiers to model the drivers for log changes. A random forest is a classifier consisting of a collection of decision tree classifiers and each tree casts a vote for the most popular class for a given input (Breiman, 2001). Random forests construct each tree using a different bootstrap sample (i.e., if the number of instances in the training set is N, randomly sample N instances with replacement) of the input data as the training set. The random forest classifier uses a bootstrap approach internally to get an unbiased evaluation of the performance of a classifier (Breiman, 2001). In addition, unlike standard decision trees where each decision node is split using the best split among all variables, random forests split each node using the best among a randomly chosen subset of variables from each of the

[15] http://sailhome.cs.queensu.ca/replication/JITLogSuggestions/correlation.zip

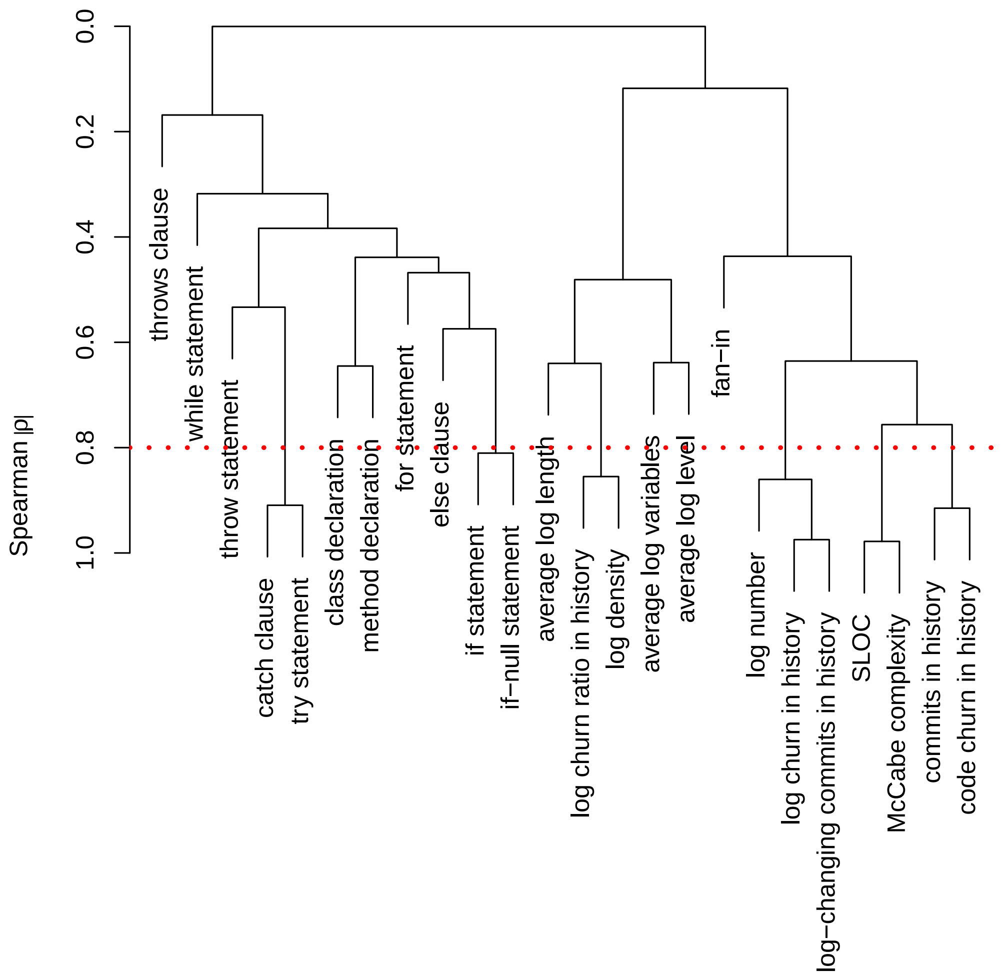


Figure 5.8: Correlation analysis using Spearman hierarchical clustering (for the combo data).

constructed trees (Liaw and Wiener, 2002). Random forests are naturally robust against overfitting, and they perform very well in terms of accuracy (Breiman, 2001). Random forest provides us a way to do sensitivity analysis on the measures so that we can understand the most influential factors in our classifiers (Breiman, 2002; Liaw and Wiener, 2002). Besides, a recent study (Ghotra et al., 2015) compares 31 classifiers in software defects prediction and suggests that Random Forest outperforms other classifiers.

Within-project Evaluation. We build a random forest classifier to determine the likelihood of a log change for each commit based on the development history prior to that particular commit. Specifically, for each commit, we build a random forest classifier using all the prior commits as training data, and use the classifier to determine whether a log change is needed for that particular commit. A classification result can be "true" (i.e., the likelihood of a log change is higher than 0.5) or "false" (i.e., the likelihood of a log change is lower than 0.5). A "true" classification result suggests the need of log changes in that code commit, while a "false" classification result suggests that no log change is needed for that commit. Then, we update the classifier with the new commit and use the updated classifier to determine the likelihood of a log change for the following commit, and so on. Evaluating the classification result for a commit can have one of four outcomes: **TP** - true positive, **FP** - false positive, **FN** - false negative, and **TN** - true negative. The outcomes are illustrated in the confusion matrix that is shown in Table 5.4.

We use *balanced accuracy* (**BA**) as prior research (Zhu et al., 2015) to evaluate the performance of our within-project evaluation. BA averages the probability of correctly identifying a log-changing commit and the probability of correctly identifying a non-log-changing commit. BA is widely used to evaluate the modeling results on imbalanced data (Zhu et al., 2015; Cohen et al., 2004; Zhang et al., 2005), because it avoids over-optimism on imbalanced data. BA is calculated by Equation (5.1):

$$BA = \frac{1}{2} \times \frac{TP}{TP+FN} + \frac{1}{2} \times \frac{TN}{FP+TN} \tag{5.1}$$

We determine the likelihood of a log change for each commit throughout the lifetime of a project, using a random forest classifier that is trained from all the prior commits of the same project, and get an outcome that is represented by one of TP, FP, FN and TN. We train the first classifier for each project when there are 50 commits in the development history; in other words, we evaluate our first classifier on the 51st commit. For each project, we sum up the TP, FP, FN and TN for all commits (except for the first 50 commits) and apply Equation 5.1 to calculate the overall performance of our just-in-time suggestions that is represented by a BA. Moreover, in order to observe the evolution of our classifier's performance over the lifetime of each project, we use a "sliding window" technique to calculate the BA for each commit. Specifically, the "sliding window" of a particular commit contains 101 consecutive commits, including 50 preceding commits, the commit itself, and 50 following commits. In order to get the BA for the particular commit, we sum up the TP, FN, TN and FP in the "sliding window" and then calculate the averaged BA using Equation (5.1). Each time we move the sliding window forward by one commit to calculate the BA for the next commit. The BA for each commit that is calculated from the "sliding window" enables us to examine the stability of the performance of our just-in-time suggestions, and whether the suggestions are accurate when there are only a small number of commits available to train a classifier at the start of a project.

Table 5.4: Confusion matrix for the classification results of a commit.

		Actual	
		Logging	Non-logging
Classified	Logging	TP	FP
	Non-logging	FN	TN

Cross-project Evaluation. Since small projects or new projects might not have enough history data for log change classification, we also evaluate our classifiers' performance in cross-project classification. We train a classifier using a combo data of $N-1$ projects (i.e., the training projects), and use the classifier to determine the likelihood of a log change for each of the commits of the remaining one project (i.e., the testing project).

We evaluate the BA of the cross-project classifiers. For each testing project, we sum up the TP, FN, TN and FP that are computed from determining the likelihood of a log change of all the commits of the project, and apply Equation (5.1) to calculate the BA.

We also use the area under the ROC curve (**AUC**) to evaluate the performance of the cross-project classifiers. While the BA measures our classifiers' accuracy in log change classification, the AUC evaluates how well our classifiers can discriminate log-changing commits and non-log-changing commits. The AUC is the area under the ROC curve which plots the true positive rate ($TP/(TP+FN)$) against false positive rate ($FP/(FP+TN)$). The AUC ranges between 0 and 1. A high value for the AUC indicates a high discriminative ability of the classifiers; an AUC of 0.5 indicates a performance that is no better than random guessing.

To avoid the unbalanced number of commits for each project in the training data, we leverage up-sampling to balance the training data such that each project has the same number of entries in the training data. Specifically, we keep unchanged the largest training project in the training data; while we randomly up-sample the entries of the other training projects with replacement to match the number of entries of the largest training project. In order to reduce the non-determinism caused by the random sampling, we repeat the "up-sampling - training - testing" process for 100 times and calculate the average BA and AUC values.

Results

Our random forest classifiers can effectively provide just-in-time suggestions for log changes using historical data from the same project. The overall BA values when considering all commits in Hadoop, DirectoryServer, HttpClient, and Qpid are 0.76, 0.83, 0.77, 0.77, respectively (see Table 5.5). Figure 5.9, Figure 5.10, Figure 5.11 and Figure 5.12 illustrate the within-project classification results using the "sliding window" technique for Hadoop, DirectoryServer, HttpClient, and Qpid, respectively. For each figure, the horizontal axis denotes the commit index while the vertical axis shows the BA value. The black solid curve plots the BA value at each commit, and the red dashed line indicates an average BA over all the commits of the project. These figures show that our random forest classifiers achieve an average BA of 0.76 to 0.82. In other words, given a commit, our classifiers can tell whether this commit should change logging statements or not, with an average accuracy of 0.76 to 0.82. Table 5.6 presents the detailed TP, FN, TN, and FP numbers that are used to calculate the overall BAs using Equation 5.1. Taking the Hadoop project for example, among the 1,541 actual logging commits (TP + FN), 76% (1,166) of them are correctly classified as logging commits, and 24% (375) of them are incorrectly classified as non-logging commits; among the 3,718 actual non-logging commits (TN + FP), 76% (2,822) of them are correctly classified as non-logging commits, and 24% (896) of them are incorrectly classified as logging commits. On average, training such a random forest classifier for a large system like Hadoop on a workstation (Intel i7 CPU, 8G RAM) takes about 2 seconds, and classifying the log changes for a particular commit takes about 0.02 seconds. For each commit, we would only need to perform the classification step in real-time, while the training can be done offline. These results indicate that our within-project classifiers

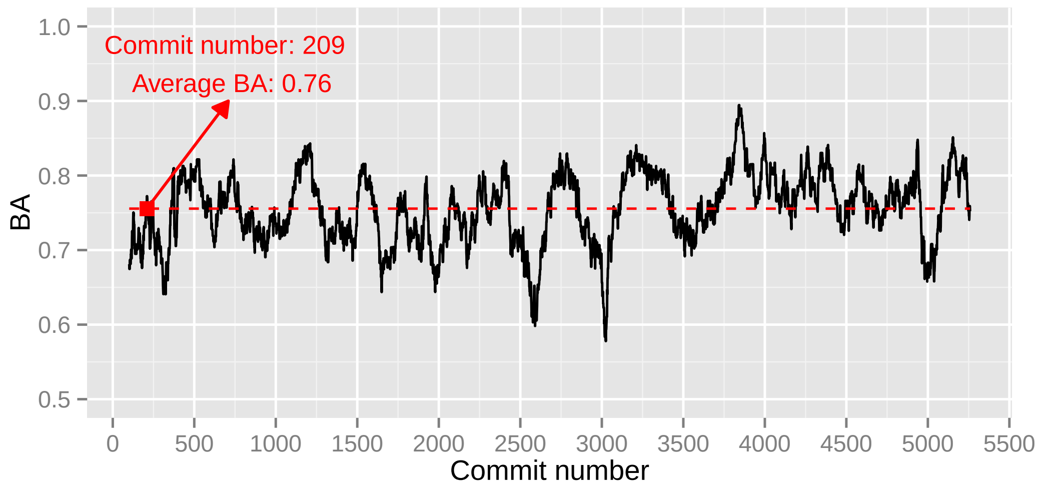


Figure 5.9: The balanced accuracy of the within-project classifiers for Hadoop.

can effectively provide just-in-time suggestions for log changes.

Our classifiers achieve the average balanced accuracy with a small number of commits as training data. We measure when a project accumulates sufficient commit data to train a classifier that reaches the average performance in terms of BA. In Figure 5.9, Figure 5.10, Figure 5.11 and Figure 5.12, we have marked the commit number where each classifier reaches its average BA for the first time. We find that the within-project classifiers for Hadoop, DirectoryServer, HttpClient, and Qpid reach their average BAs when the projects got 209, 193, 211 and 155 commits, respectively. The results indicate that the classifiers can learn an average classification power after a relatively small number of commits.

The performance of our classifiers fluctuates over time. As shown in Figure 5.9 through 5.12, the performance (in terms of the BA) of the within-project classifiers fluctuates over time, and the fluctuation does not follow any clear trend. These results might be explained by the assumption that developers follow different logging practices at different development stages. For example, developers might be less focused

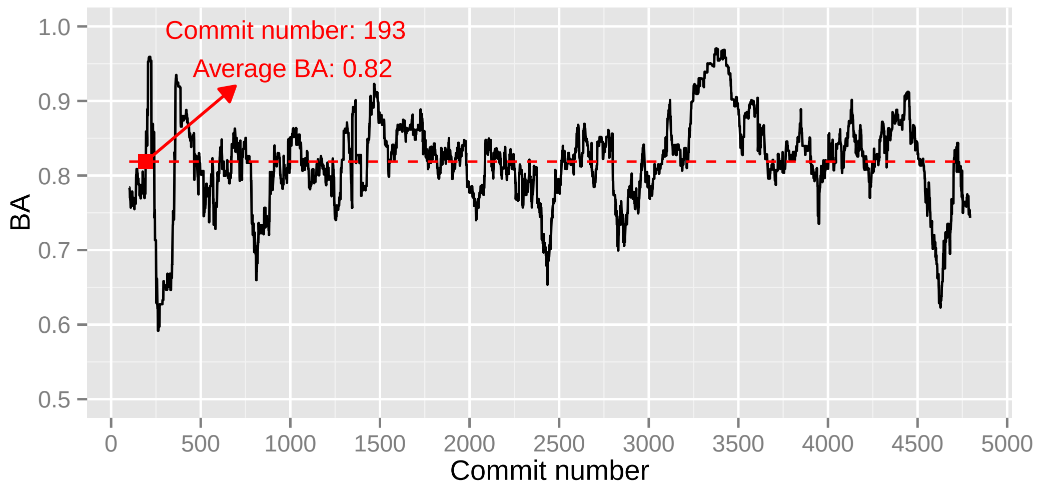


Figure 5.10: The balanced accuracy of the within-project classifiers for DirectoryServer.

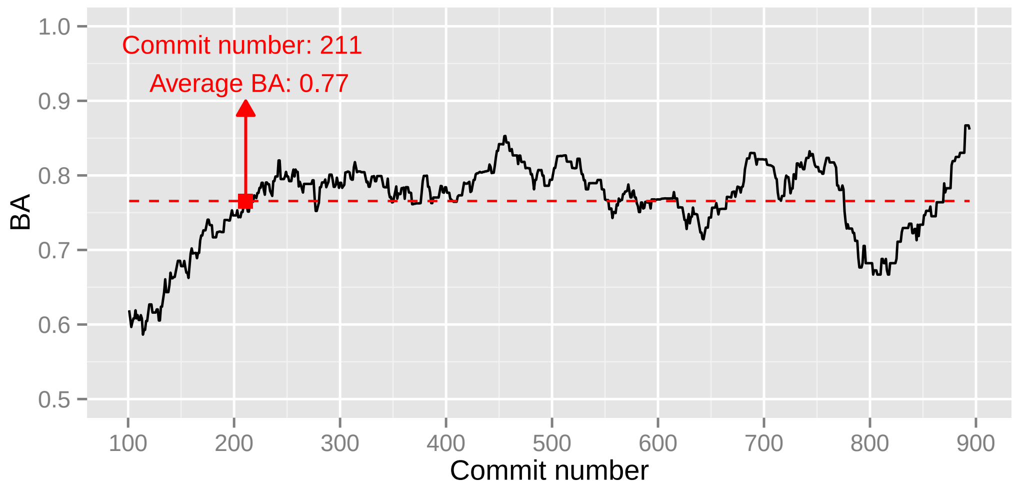


Figure 5.11: The balanced accuracy of the within-project classifiers for HttpClient.

on logging at the beginning of a release cycle and might pay more attention on logging when a product is approaching its release (and final testing is being performed on it). Table 5.5 shows the average BA, 5 percentile BA and 95 percentile BA for the within-project classifiers over time. Only 5% of the commits get a BA smaller than 0.68, 0.71, 0.66 and 0.68 for Hadoop, DirectoryServer, HttpClient, and Qpid, respectively. And 5% commits get a BA bigger than 0.82, 0.92, 0.83 and 0.88, respectively.

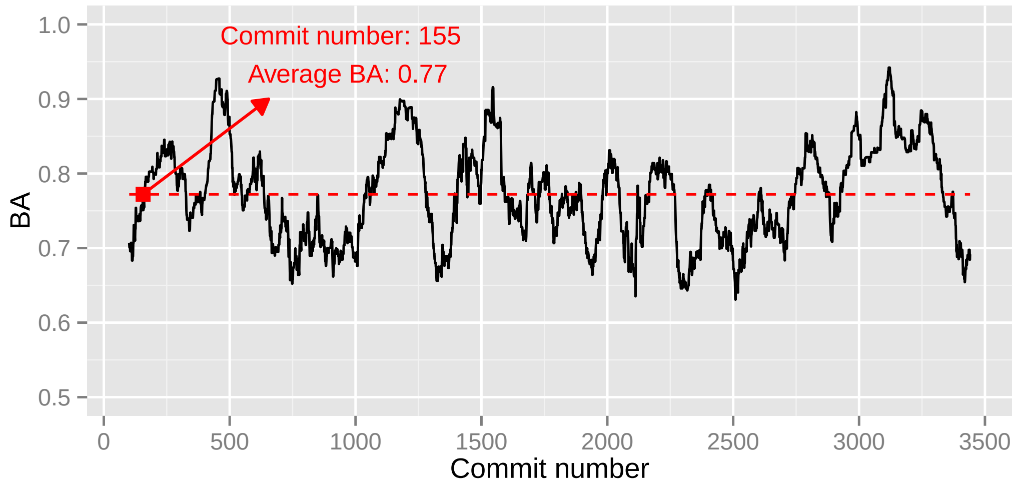


Figure 5.12: The balanced accuracy of the within-project classifiers for Qpid.

Table 5.5: The BA results for the within-project evaluation.

Project	Sliding window			Overall BA
	Average BA	5% BA	95% BA	
Hadoop	0.76	0.68	0.82	0.76
DirectoryServer	0.82	0.71	0.92	0.83
HttpClient	0.77	0.66	0.83	0.77
Qpid	0.77	0.68	0.88	0.77

Table 5.6: The TP, FN, TN, FP results for the within-project evaluation.

Project	**TP**	**FN**	**TN**	**FP**
Hadoop	1,166	375	2,822	896
DirectoryServer	907	167	2,998	721
HttpClient	193	50	490	161
Qpid	664	196	2,006	575

Our random forest classifiers can effectively provide just-in-time suggestions for log changes using the development history of other projects. Table 5.7 lists the performance of the cross-project classifiers, expressed by the BA and the AUC measures. Each row of the table shows the performance of the classifier that uses the specified

Table 5.7: The BA and AUC results for the cross-project evaluation.

Project	**BA**	**AUC**
Hadoop	0.76	0.84
DirectoryServer	0.80	0.88
HttpClient	0.79	0.87
Qpid	0.78	0.86

project as testing data and all the other projects as training data. The cross-project classifiers reach a BA of 0.76 to 0.80, indicating that the cross-project classifiers can effectively determine the likelihood of a log change for each commit of a project with little development history.

Our random forest classifiers can effectively discriminate log-changing commits and non-log-changing commits. As shown in Table 5.7, the cross-project classifiers achieve an AUC of 0.84 to 0.88. The high AUC values indicate that the classifiers perform much better than random guessing in discriminating the log-changing commits and non-log-changing commits.

Cross-project classification and within-project classification achieve similar performance. The within-project classifiers achieve an average BA of 0.76, 0.82, 0.77, 0.77 for Hadoop, DirectoryServer, HttpClient, and Qpid, respectively; while the cross-project classifiers reach a BA of 0.76, 0.80, 0.79 and 0.78 for these four projects, respectively. These results show that developers of a new software project that do not have a large amount of development history can leverage classifiers that are built from other projects to provide just-in-time log change suggestions.

Our random forest classifiers can effectively provide just-in-time suggestions for log changes, with a balanced accuracy of 0.76 to 0.82 in a within-project evaluation and a balanced accuracy of 0.76 to 0.80 in a cross-project evaluation. Developers of the studied systems can leverage such classifiers to guide their log changes in practice.

5.3.3 RQ3: What are the influential factors that explain log changes?

Motivation

In order to quantitatively understand the reasons for log changes, we analyze the random forest classifiers to find out the most influential factors that are associated with log changes. There are three types of log changes: log additions, log deletions, and log modifications. The influential factors for log changes, log additions, log deletions and log modifications may be different. Therefore, we analyze the influence of factors for log changes, log additions, log deletions, and log modifications, separately. From our manual analysis in RQ1, we find that the occurrences of deleting a logging statement is usually associated with deleting the enclosing code block (see Table 5.2). Therefore, we do not analyze the factors that influence log deletions.

Approach

Bootstrap analysis. In order to study the influential factors that affect log changes, we use the bootstrap method to repeatedly sample training data and build a large number of random forest classifiers, so as to statistically analyze the influence of the factors. Bootstrap (Efron, 1979) is a general approach to infer the relationship between a sample data and the overall population, by resampling the sample data with replacement and analyzing the relationship between the resample data and the sample data. The bootstrap analysis is implemented in the following steps:

- Step 1. From the original dataset with N instances, we choose a random bootstrap sample of N instances with replacement.
- Step 2. We build a random forest classifier using the bootstrap sample.
- Step 3. We collect the influence of each factor in the classifier.
- Step 4. We repeat the above steps 1,000 times.

By repeatedly using the bootstrap samples to analyze the influence of the factors, we can avoid the bias that might be caused by a single round of modeling analysis.

Variable influence in Random Forest. The random forest classifier evaluates the influence of each factor by permuting the values of the corresponding measure of that factor while keeping the values of the other factors unchanged in the testing data (the so-called "OOB" data) (Breiman, 2001). The classifier measures the impact of such a permutation on the classification error rate (Breiman, 2002; Liaw and Wiener, 2002). In each round of the 1,000 bootstraps, we use the "importance" function in the R package "randomForest" to evaluate the influence of the factors.

Scott-Knott clustering. In this step, we compare the average influence of the factors from the 1,000 bootstrap iterations. However, the differences among the influence of some factors might actually be due to random variability. Thus we need to partition all the factors into statistically homogeneous groups so that the influence means of the factors within the same group are not significantly different (i.e, p-value ≥ 0.05). The Scott-Knott (SK) algorithm is a hierarchical clustering approach that can partition the results into distinct homogeneous groups by comparing their means (Scott and Knott, 1974; Jelihovschi et al., 2014). The SK algorithm hierarchically divides the factors into groups and uses the likelihood ratio test to judge the significance of difference among the groups. The SK method generates statistically distinct groups of factors, i.e., each two groups of factors have a p-value < 0.05 in a likelihood ratio test of their influence values.

In this work, we use an enhanced SK approach (Tantithamthavorn et al., 2017), which considers the effect size in addition to the statistical significance of the difference between groups, to divide the factors into distinct groups according to their influence in the random forest classifiers.

We repeat our approach (bootstrap analysis, variable influence and Scott-Knott clustering) for the log addition classifier and the log modification classifier to study the most influential factors for log additions and log modifications respectively.

We also compare the influential factors with the log change reasons that we observed in our manual analysis step in RQ1. The connections between the two outcomes help us better understand the reasons behind developers' decision of changing logs.

Results

***Change measures* and *product measures* are the most influential factors for log changes.** Table 5.8, Table 5.9 and Table 5.10 present the influence values of the 10 most influential predictor variables for the log change classifier, the log addition classifier and the log modification classifier, respectively. For each studied project, we measure the mean value of each factor's influence. The factors are divided into statistically distinct groups by a Scott-Knott test on the influence values. Overall, the most influential factors in these classifiers include the *if statement* (with a median group ranking of 2), *catch clause* (with a median group ranking of 3) and *method*

Table 5.8: The influence mean of the top 10 factors (measures) for the log change classifier, divided into distinct homogeneous groups by Scott-Knott clustering.

Hadoop			Directory Server		
Group	**Factor**	**Influence Mean**	**Group**	**Factor**	**Influence Mean**
1	log density	0.149	1	catch clause	0.169
2	if statement	0.135		if statement	0.169
3	catch clause	0.122	2	log density	0.134
4	log number	0.107	3	throw statement	0.125
5	method declaration	0.072	4	method declaration	0.085
6	average log length	0.069	5	log number	0.080
7	average log variables	0.068	6	average log variables	0.076
8	average log level	0.066	7	SLOC	0.072
9	SLOC	0.062	8	else clause	0.068
10	else clause	0.061	9	average log length	0.065
HttpClient			Qpid		
Group	**Factor**	**Influence Mean**	**Group**	**Factor**	**Influence Mean**
1	if statement	0.119	1	log density	0.155
2	log density	0.081	2	catch clause	0.125
	log number	0.080	3	method declaration	0.095
3	average log length	0.074		log number	0.095
4	average log level	0.072	4	if statement	0.094
5	average log variables	0.066	5	average log level	0.082
6	method declaration	0.060	6	average log variables	0.073
7	catch clause	0.057	7	else clause	0.069
8	code churn in history	0.052	8	average log length	0.064
	SLOC	0.051	9	SLOC	0.057

declaration (with a median group ranking of 5) measures from the dimension of *change measures*, as well as the *log density* (with a median group ranking of 2), *log number* (with a median group ranking of 3.5), *average log variables* (with a median group ranking of 6), *average log length* (with a median group ranking of 6.5) and *average log level* (with a median group ranking of 6.5) measures from the dimension of *product measures*.

The strong influence of the measures from the *change measures* dimension indicates that log changes are highly associated with other code changes, and this is in accordance with the fact that the *block change* reasons are the most frequent

Table 5.9: The influence mean of the top 10 factors (measures) for the log addition classifier, divided into distinct homogeneous groups by Scott-Knott clustering.

Hadoop			Directory Server		
Group	**Factor**	**Influence Mean**	**Group**	**Factor**	**Influence Mean**
1	catch clause	0.152	1	catch clause	0.204
2	if statement	0.145	2	if statement	0.182
3	log density	0.141	3	throw statement	0.130
4	log number	0.102	4	log density	0.124
5	else clause	0.092	5	SLOC	0.096
6	average log level	0.079	6	else clause	0.087
7	method declaration	0.070	7	method declaration	0.086
8	SLOC	0.067	8	average log length	0.081
9	average log length	0.065	9	log number	0.070
10	fan-in	0.064	10	fan-in	0.066
HttpClient			Qpid		
Group	**Factor**	**Influence Mean**	**Group**	**Factor**	**Influence Mean**
1	if statement	0.121	1	catch clause	0.162
2	log density	0.091	2	log density	0.145
3	catch clause	0.085	3	if statement	0.103
4	throw statement	0.081	4	method declaration	0.091
	log number	0.080	5	log number	0.088
5	average log level	0.071	6	average log variables	0.082
6	average log variables	0.066	7	average log level	0.073
	average log length	0.064	8	else clause	0.071
7	throws clause	0.059	9	fan-in	0.065
8	method declaration	0.055	10	class declaration	0.064

reasons that we observed in our manual analysis. In particular, the influence of the *catch clause* measure suggests a strong association between exception handling and logging, and this matches with the *adding/deleting try-catch block* reason that is observed in our manual analysis. This result also quantitatively supports best practices recommendation that exception-handling code should log the information that is associated with the exception being handled (Apache-Commons, 2016; Microsoft-MSDN, 2016). The strong influence of the *if statement* and *if-null statement* measures (*if-null statement* has a high correlation with *if statement*) shows that developers tend

Table 5.10: The influence mean of the top 10 factors (measures) for the log modification classifier, divided into distinct homogeneous groups by Scott-Knott clustering.

Hadoop			Directory Server		
Group	**Factor**	**Influence Mean**	**Group**	**Factor**	**Influence Mean**
1	log density	0.191	1	log number	0.177
2	log number	0.160	2	log density	0.168
3	if statement	0.142	3	average log length	0.155
4	average log variables	0.106	4	if statement	0.117
5	method declaration	0.100	5	method declaration	0.109
6	average log length	0.097	6	average log variables	0.098
7	average log level	0.089	7	SLOC	0.092
	fan-in	0.088	8	average log level	0.076
8	SLOC	0.084	9	fan-in	0.071
9	code churn in history	0.071	10	throw statement	0.063
HttpClient			**Qpid**		
Group	**Factor**	**Influence Mean**	**Group**	**Factor**	**Influence Mean**
1	log number	0.140	1	log density	0.184
2	if statement	0.110	2	log number	0.165
3	log density	0.096	3	if statement	0.106
	average log variables	0.096	4	average log variables	0.101
4	method declaration	0.093		catch clause	0.101
5	average log level	0.091	5	average log level	0.095
6	SLOC	0.076	6	average log length	0.090
7	average log length	0.068	7	method declaration	0.087
8	code churn in history	0.058	8	SLOC	0.070
9	fan-in	0.047	9	fan-in	0.068

to change logging statements while changing conditional branches, and this corresponds to the *adding/deleting branch* and *adding/deleting if-null branch* reasons that we observed in RQ1. As another influential factor from the *change measures* family, the *method declaration* measure suggests that the change of a method declaration is strong indicator for log changes. Again, this result consents with manually detected reason *adding/deleting method*.

The great influence of the measures from the *product measures* dimension implies that the current snapshot of the source code impacts the logging decisions of developers. Specifically, the *log density* and *log number* measures being influential factors indicates that log changes occur more in code snippets with higher log density, and this agrees with our manually-observed *log improvement* and *logging issue* reason categories; the influence of the *average log level*, *average log length* and *average log variables* measures shows that the characteristics of the existing logging statements impact developers' decision on whether to change logs in a commit, which also corresponds to the manually detected reason categories *log improvement* and *logging issue*.

***Change measures* are the most influential measures for the log addition classifier, while *product measures* are the most influential ones for the log modification classifier.** Developers tend to add logging statements when adding contextual code. The current snapshot of the source code is the best indicator for log modifications. This result agrees with the observations from our manual analysis that the most frequent reasons for log additions are from the *block change* category, while the reasons for log modifications are from the *log improvement, dependence-drive change* and the *logging issue* categories. The *catch clause* (with a median group ranking of 1 in the log addition classifier) and *if statement* (with a median group ranking of 2 in the log addition classifier) measures from the *change measures* dimension play the most influential roles in the log addition classifier. Developers tend to add logging statements when they are adding exception catching block (catch) or dealing with a conditional branch. The *log density* and *log number* measures (both with a median group ranking of 1.5 in the log modification classifier) from the *product measures* dimension are the most influential measures for the log modification classifier, which means that log modifications occur more often in code snippets with high log density.

Different projects present different log change practices. The *catch clause* measure is one of the most influential indicators for a log change in the *Hadoop, Directory*

Server, and *Qpid* projects, with a group ranking of 3, 1, and 2, respectively; however, the *catch clause* measure is a less influential indicator for log changes in the *HttpClient* project (with a group ranking of 7). This might imply that developers for the *HttpClient* project are less likely to log exception-handling blocks; or it maybe because that there are significantly less changes of *catch clauses* for the *HttpClient* project compared to other three projects. However, the latter inference is unlikely since 18.5% of the commits for the *HttpClient* project contain changes to *catch clauses*, while that proportions are 21.0%, 20.3% and 27.5% for the *Hadoop*, *Directory Server*, and *Qpid* projects, respectively.

For the log addition classifier, the *throw statement* measure shows much stronger explanatory power in the *Directory Server* and *HttpClient* projects (with a group ranking of 3 and 4, respectively) than that in the other two projects. This difference might indicate that developers for the *Directory Server* and *HttpClient* projects are more likely to add logging statements when they throw an exception; or it maybe due to the rareness of changes of *throw statement* in the *Hadoop* and *Qpid* projects. Again, the latter inference is not likely since the proportions of commits that contain changes to *throw statement* are 26.5% and 28.6% for the *Hadoop* and *Qpid* projects, respectively, and these proportions for the *Directory Server* and *HttpClient* projects are 25.8% and 22.8% respectively.

Code changes in a commit and the current snapshot of the source code are the most influential indicators for a log change in a commit. *Change measures* are the most influential factors for the log addition classifier, while *product measures* are the most influential factors for the log modification classifier.

5.4 Discussion

Discussion regarding log-changing-only commits. Our random forest classifiers aim to provide developers with just-in-time log change suggestions when they commit code changes. However, developers sometimes also change logging statements without affecting other source code (we call such commits as log-changing-only commits). In such cases, our approach cannot provide just-in-time log change suggestions. As shown in Table 5.11, the log-changing-only commits take up 1.2% to 4.2% of all the commits that change logging statements. By manually examining all these log-changing-only commits, we find that a log-changing-only commit occurs when either 1) the developers have not correctly implemented the needed logging statements in the first place, or 2) the requirements of logging have changed afterwards. The example in Figure 5.13 shows that a logging statement was missing in an exception handling code, which increased the difficulty of finding the root cause of a reported bug (i.e., QPID-1352[16]). A log-changing-only commit (704187) added a logging statement in the exception handling block so that “hopefully the next time it shows up we have a bit more info”.

The commit shown in Figure 5.14, in contrast, removes two logging statements which were previously inserted for debugging purposes, because the requirement of logging has changed: they don’t need the debug message anymore.

Table 5.11 presents the number of log-changing-only commits that add, delete and modify logging statements, respectively. 88 out of 129 (68.2%) log-changing-only commits modify existing logging statements, while 38 (29.5%) of them add new logging statements. Developers are least likely to delete a logging statement without changing

[16] https://issues.apache.org/jira/browse/QPID-1352

```
/* Project: Qpid; Commit:704187
 * File: /incubator/qpid/trunk/qpid/java/client/src/main/java/
    org/apache/qpid/client/AMQConnection.java
 * Commit message: "log the original exception so we don't lose the stack
    trace"
 * JIRA issue report: "I've committed a change to log the original error.
    I can't reproduce this on my system, so hopefully the next time it
    shows up we have a bit more info."
 */
  catch (AMQException e)
  {
+   _logger.error("error:", e);
    JMSException jmse = new JMSException("Error closing connection: " + e);
    jmse.setLinkedException(e);
    throw jmse;
  }
```

Figure 5.13: An example of adding a missing logging statement.

```
/* Project: Qpid; Commit: 1230013
 * File: /qpid/trunk/qpid/java/broker/src/main/java/
    org/apache/qpid/server/subscription/DefinedGroupMessageGroupManager.java
 * Commit message: "Remove debugging log statements"
 */
  if(newState == QueueEntry.State.ACQUIRED)
  {
-   _logger.debug("Adding to " + _group);
    _group.add();
  }
  else if(oldState == QueueEntry.State.ACQUIRED)
  {
-   _logger.debug("Subtracting from " + _group);
    _group.subtract();
  }
```

Figure 5.14: An example of removing logging statements due to changed logging requirements.

other source code: only 24 (18.6%) of the log-changing-only commits delete logging

Table 5.11: Number of log-changing-only commits.

Project	Log-changing commits	Log-changing-only commits			
		Change logs	Add logs	Del. logs	Mod. logs
Hadoop	1621	68 (4.2%)	20	9	49
Directory S.	1130	32 (2.8%)	7	6	24
HttpClient	252	3 (1.2%)	0	2	1
Qpid	908	26 (2.9%)	11	7	14
Total	3911	129 (3.3%)	38	24	88

statements. If developers forget to add, delete or modify a logging statement, leading to a log-changing-only commits afterwards, our approach would help developers avoid forgetting to change logs in the first place. If current logging statements need to be fixed or improved, our approach cannot provide such suggestion without the context of other code changes. In the studied projects, however, the log-changing-only commits represent 1.2% to 4.2% of the commits that change logging statements.

5.5 Threats to Validity

5.5.1 External Validity

The external threat to validity is concerned with the generalization of the results. In our work, we investigated four open source projects that are of different domains and sizes. However, since other software systems may follow different logging practices, the results may not generalize to other systems. Further, we only analyze Java source code in this study, thus the results may not generalize to systems that are developed in non-Java languages.

5.5.2 Internal Validity

The manual analysis for log change reasons is subjective by definition, and it is very difficult, if not impossible, to ensure the correctness of all the inferred log change reasons. We classified the log change reasons into four categories; however, there may be different categorizations. Nevertheless, there is a strong agreement between the results of our manual and automated analysis.

The random forest modeling results present the relation between log changes and a set of software measures. The relation does not represent the casual effects of these measures on log changes. In order to learn log change reasons from the modeling results, we link the influential factors in the random forest classifiers back to the manually detected reasons while analyzing the importance factors in these classifiers. Future studies should conduct longitudinal developer studies or interviews to further understand the rationale for log changes.

In this study we only analyze Java source code. However, the project *Qpid* is developed in multiple languages including Java, C++, C#, Perl, Python and Ruby. Although a large percent of *Qpid* code (2,995 Java files out of totally 4,757 source code files) is developed in Java, the results still can not fully represent its log change practices.

In this chapter we learn developers' logging practices in the past and leverage the learned knowledge to provide suggestions for future log changes. Our study is based on the assumption that these projects' logging practices are appropriate and are good practices that future changes should follow. However, the logging practices in these projects may not be always appropriate. In order to avoid learning the bad practices, we choose several successful open source projects which follow a strict code review process.

In this chapter we study the development of logging statements in the source code. However, we don't consider whether these logging statements would actually output log messages during the system execution (i.e., the dynamic impact). Whether a logging statement can output log messages depends on various dynamic information such as: 1) whether the path of the logging statement is executed; 2) whether the level of the logging statement is turned on to print messages. Extending our chapter by leveraging dynamic information is a promising avenue for future work.

In the results of RQ3, we analyze the influential factors that impact the log changes. The explanations are inferred based on the modeling results and our manual exploration of log change reasons. However, they are not necessarily the actual causes that lead to log changes, instead they are just possible explanations to the logging practices of the studied projects.

5.5.3 Construct Validity

The construct threat is concerned with how we identify log changes. We identify log changes using a set of predefined regular expressions. The regular expressions may not identify all the log changes. For example, developers may define their own logging functions which are difficult to track. However, our approach can detect all the logging statements that leverage the standard logging libraries with a precision of 100%. Future studies might consider using a static analysis approach. Nevertheless, a manual verification of the non-standard (i.e., defined by developers themselves) logging functions will always be needed since it is impossible to automatically determine whether a function is a logging wrapper function.

We identify a log modification by calculating the Levenshtein distance ratio between a pair of added and deleted logging statements. Two logging statements are considered similar if the Levenshtein distance ratio between them is larger than a specified threshold (L-threshold). In this chapter, we choose an L-threshold value of 0.5 to identify a log modification. However, this approach is not guaranteed to identify all log modifications. In order to address this threat, we perform a sensitivity analysis to measure the impact of the L-threshold value on the identification of log modifications for all the log changes in the Hadoop project. Specifically, we change the L-threshold to 0.4 and 0.6 and examine the difference of the identification results for these two thresholds. Table 5.12 shows summary statistics of the identification results using the 0.4, 0.5 and 0.6 L-thresholds. We identify 1,861 log modifications using the 0.5 threshold; while we identify 1,932 (+3.8%) and 1,788 (−3.9%) modifications for the 0.4 and 0.6 L-thresholds, respectively. The results show that using a different L-threshold does not lead to a large change in the number of identified log modifications.

We examine the amount of added logging statements that are classified differently (i.e., classified as a log addition or a part of a log modification) when using different L-thresholds. We find that there are only 248 out of 7,215 (3.4%) added logging statements that are classified differently using different L-thresholds. We manually examine these 248 added logging statements that are classified differently and identify which L-threshold can accurately classify the added logging statements. We find that using 0.5 as a threshold has the highest accuracy. In particular, 106 out of the 248 (42.7%) added logging statements that are classified differently are correctly classified by using the 0.4 threshold; 162 (65.3%) of the added logging statements are correctly classified using the 0.5 threshold; and 149 (60.1%) of the added logging statements are correctly

Table 5.12: L-threshold's impact on the identification of log modifications (for the Hadoop project). The values in the brackets are the percentages of difference when compared with the 0.5 L-threshold.

L-threshold	log addition	log deletion	log modification
0.4	5,283 (−1.3%)	2,217 (−3.1%)	1,932 (+3.8%)
0.5	5,354	2,288	1,861
0.6	5,427 (+1.4%)	2,361 (+3.2%)	1,788 (−3.9%)

classified using the 0.6 threshold.

In this chapter we choose 25 software measures (as listed in Table 5.3) based on our manual analysis results and our intuition. However, there may be other measures, such as OO measures (D'Ambros et al., 2012), that can assist in suggesting log changes. Some feature learning techniques may also help improve the performance of our classifiers. As a first step of suggesting proper log changing practices, we aim to show the benefit of leveraging historical information for providing log change suggestions in the future. We expect more measures or feature learning techniques to be proposed in follow-up research in order to provide more accurate classifiers to suggest log changes.

This chapter proposes an approach that can provide developers with suggestions on whether to change a logging statement when they commit code changes. Future work may include real developers' feedback to better evaluate and further improve the usefulness of our approach.

5.6 Chapter Summary

In this work, we leverage machine learning classifiers to provide just-in-time suggestions for changing logs when developers commit code changes. We firstly manually investigate a statistically representative random sample of log changes from four open source projects *Hadoop, Directory Server, Commons HttpClient, and Qpid*, in order to understand the reasons behind log changes. Based on the results of our manual analysis and our experiences, we derive measures as input into random forest classifiers to model the drivers for log changes. Our experimental results show that the random forest classifiers can accurately provide just-in-time log change suggestions using a within and across projects evaluation. Finally, we study which measures play influential roles in the models and thereby influence log changing practices the most. Some of the key findings of our study are as follows:

- Log change reasons can be grouped along four categories: block change, log improvement, dependence-driven change, and logging issue.
- Our random forest classifiers can provide accurate just-in-time suggestions for log changes. The classifiers trained from historical data of the same project achieve a balanced accuracy of 0.76 to 0.82; the classifiers trained from other projects reach a balanced accuracy of 0.76 to 0.80 and an AUC of 0.84 to 0.88.
- Code changes in a commit and the current snapshot of the source code are the most influential factors for determining the likelihood of a log change in a commit.

Our work provides insights about the reasons why developers change (add, modify or delete) logging statements in their code. These reasons together with the logging concerns derived in Chapter 3 can help developers and researchers better understand the current logging practices. Developers can also leverage our automated approach to guide their log changing practices.

CHAPTER 6

Automated Suggestions for Choosing Log Levels

In Chapter 4 and Chapter 5, we propose automated approaches to provide developers with suggestions for where to log and log changes. As we observe in Chapter 3, developers often have difficulties when determining the appropriate levels for their logging statements. In this chapter, we propose an approach to help developers determine the appropriate log level when they add a new logging statement. We analyze the development history of four open source projects, and leverage ordinal regression models to automatically suggest the most appropriate level for each newly-added logging statement. First, we find that our ordinal regression model can accurately suggest the levels of logging statements with an AUC (area under the curve; the higher the better) of 0.75 to 0.81 and a Brier score (the lower the better) of 0.44 to 0.66, which is better than randomly guessing the appropriate log level (with an AUC of 0.50 and a Brier score of 0.80 to 0.83) or naively guessing the log level based on the proportional distribution of each log level (with an AUC of 0.50 and a Brier score of 0.65 to 0.76). Second, we find that the characteristics of the containing block of a newly-added logging statement, the existing logging statements in the containing source code file, and the content of the newly-added logging statement play important roles in determining the appropriate log level for that logging statement.

An earlier version of this chapter is published in the Empirical Software Engineering Journal (EMSE) (Li et al., 2017a).

6.1 Introduction

LOGS are widely used by software developers to record valuable run-time information about software systems. Logs are produced by logging statements that developers insert into the source code. A logging statement, as shown below, typically specifies a log level (e.g., debug/info/warn/error/fatal), a static text and one or more variables.

logger.error("static text" + variable);

However, appropriate logging is difficult to reach in practice. Both logging too little and logging too much is undesirable (Fu et al., 2014; Chapter 3). Logging too little may result in the lack of runtime information that is crucial for understanding software systems and diagnosing field issues (Yuan et al., 2010; Chapter 3). On the other hand, logging too much may lead to system runtime overhead and cost software practitioners' effort to maintain these logging statements (Fu et al., 2014; Chapter 3). Too many logs may contain noisy information that becomes a burden for developers during failure diagnosis (Chapter 3).

The mechanism of "log levels" allows developers and users to specify the appropriate amount of logs to print during the execution of the software. Using log levels, developers and users can enable the printing of logs for critical events (e.g., errors), while suppressing logs for less critical events (e.g., bookkeeping events) (Gülcü and Stark, 2003). Log levels are beneficial for both developers and users to trade-off the rich information in logs with their associated overhead. Common logging libraries such as Apache Log4j[1], Apache Commons Logging[2] and SLF4J[3] typically support six log levels,

[1]http://logging.apache.org/log4j/2.x
[2]http://commons.apache.org/proper/commons-logging
[3]http://www.slf4j.org

including *trace*, *debug*, *info*, *warn*, *error*, and *fatal*. The log levels are ordered by the verbosity level of a logged event: "trace" is the most verbose level and "fatal" is the least verbose level. Users can control the verbosity level of logging statements to be printed out during execution. For example, if a user sets the verbosity level to be printed at the "warn" level, it means that only the logging statements with the "warn" level or with a log level that is less verbose than "warn" ("error" and "fatal") would be printed out.

As we observe in Chapter 3, developers often have difficulties when determining the appropriate levels for their logging statements. Prior work also finds that developers spend much effort on adjusting the levels of their logging statements (Yuan et al., 2012b). Oliner et al. (2012) explains this issue by arguing that developers rarely have complete knowledge of how the code will ultimately be used. For example, JIRA issues HADOOP-10274[4] and HADOOP-10015[5] are both about an inappropriate choice of log level. The logging statement was initially added with an *error* level. However, the log level of the logging statement was later changed to the *warn* level (see code patch in Figure 6.1), as it was argued that "the error may not really be an error if client code can handle it" (HADOOP-10274); the log level of the same logging statement was finally changed to the *debug* level (see patch in Figure 6.2) after active discussions among the stakeholders (Hadoop-10015). The discussion involved eight people to decide on the most appropriate log level of the logging statement and to make the code changes. Besides, as detailed in the "Discussion" section (see Section 6.5), we observe 491 logging statements in the studied projects that experienced at least a subsequent log level change after their initial commits. These observations indicate that developers do maintain and update log levels over the lifetime of a project.

[4]https://issues.apache.org/jira/browse/HADOOP-10274
[5]https://issues.apache.org/jira/browse/HADOOP-10015

```
  } catch (PrivilegedActionException pae) {
    Throwable cause = pae.getCause();
-   LOG.error("PriviledgedActionException as:"+this+" cause:"+cause);
+   LOG.warn("PriviledgedActionException as:"+this+" cause:"+cause);
```

Figure 6.1: Patch for JIRA issue HADOOP-10274 (svn commit number: 1561934).

```
  } catch (PrivilegedActionException pae) {
    Throwable cause = pae.getCause();
-   LOG.warn("PriviledgedActionException as:"+this+" cause:"+cause);
+   if (LOG.isDebugEnabled()) {
+     LOG.debug("PrivilegedActionException as:" + this + " cause:" + cause);
+   }
```

Figure 6.2: Patch for JIRA issue HADOOP-10015 (svn commit number: 1580977).

To the best of our knowledge, there exists no prior research regarding log level guidelines. Yuan et al. (2012b) build a simple log level checker to detect inconsistent log levels. Their checker is based on the assumption that if the logging code within two similar code snippets have inconsistent log levels, at least one of them is likely to be incorrect. In other words, the checker only detects inconsistent levels but does not suggest the most appropriate log levels.

In this chapter, we propose an automated approach to help developers determine the most appropriate log level when adding a logging statement. Admittedly, it is hard, if not impossible, to evaluate whether the log level of a logging statement is correct, because different projects would have different logging requirements. However, we believe that it is a good practice for a single project to follow a consistent approach for setting the log level for its logging statements. In this chapter, we assume that in most cases developers of a project can keep consistent logging practices, and we define "appropriateness" of a log level as whether the log level is consistent with the common practice of choosing a log level within a project.

Our preliminary study shows that logging statements have a different distribution of log levels across the different containing code blocks, and particularly, in different types of exception handling blocks. Based on our preliminary study and our intuition, we choose a set of software metrics and build ordinal regression models to automatically suggest the most appropriate level for a newly-added logging statement. We leverage ordinal regression models in automated log level prediction because log level has a small number (e.g., six) of categorical values and the relative ordering among these categorical values is important, hence neither a logistic regression model nor a classification model is as appropriate as an ordinal regression model. We also carefully analyze our models to find the important factors for determining the most appropriate log level for a newly-added logging statement. In particular, we aim to address the following two research questions.

RQ1: How well can we model the log levels of logging statements?

Our ordinal regression models for log levels achieve an AUC (the higher the better) of 0.75 to 0.81 and a Brier score (the lower the better) of 0.44 to 0.66, which is better than randomly guessing the appropriate log level (with an AUC of 0.50 and a Brier score of 0.80 to 0.83) or naively guessing the log level based on the proportion of each log level (with an AUC of 0.50 and a Brier score of 0.65 to 0.76).

RQ2: What are the important factors for determining the log level of a logging statement?

We find that the characteristics of the containing block of a newly-added logging statement, the existing logging statements in the containing file, and the content of the newly-added logging statement play important roles in determining the appropriate log level for that particular logging statement.

This is the first work to support developers in making informed decisions when determining the appropriate log level for a logging statement. Developers can leverage our models to receive automatic suggestions on the choices of log levels for their newly-added logging statements. Our results also provide an insight on the factors that influence developers when determining the appropriate log level for a newly-added logging statement.

Chapter organization. The remainder of the chapter is organized as follows. Section 6.2 describes the studied software projects and our experimental setup. Section 6.3 performs an empirical study on the log level distribution in the studied projects. Section 6.4 explains the approaches that we used to answer the research questions and presents the results of our case study. Section 6.5 discusses the topics about cross-project evaluation and the log level changes. Section 6.6 discusses threats to the validity of our findings. Finally, section 6.7 draws conclusions.

6.2 Case Study Setup

This section describes the subject projects and the process that we used to prepare the data for our case study.

6.2.1 Subject Projects

We study how to determine the appropriate log level of a logging statement through a case study on four open source projects: *Hadoop*, *Directory Server*, *Hama*, and *Qpid*.

We choose these projects as case study projects for the following reasons: 1) All four projects are successful and mature projects with more than six years of development history. 2) They represent different domains, which ensures that our findings are not limited to a particular domain. *Hadoop* is a distributed computing platform, developed in Java; *Directory Server* is an embeddable directory server written in Java; *Qpid* is an instant message tool, developed in Java, C++, C#, Perl, Python and Ruby; *Hama* is a general-purpose computing engine for speeding up computing-intensive jobs, written in Java. 3) The Java source code of these projects makes extensive use of standard Java logging libraries such as *Apache Log4j*, *SLF4J* and *Apache Commons Logging* libraries, which support six log levels, i.e., from *trace* (the most verbose) to *fatal*(the least verbose).

We analyze the log levels of the newly-added logging statements during the development history of the studied projects, considering only the Java source code (excluding the Java test code). We focus our study on the development history of the main branch (trunk) of each project. We use the "svn log"[6] command to retrieve the development history for each project (i.e., the svn commit records). Some revisions import a large number of atomic revisions from a branch into the trunk (a.k.a. merge revisions), which usually contain a large amount of code changes and log changes. Such merge revisions would introduce noise (Zimmermann et al., 2004; Hassan and Holt, 2004; Hassan, 2008) in our study of log level in a newly-added logging statement. We unroll each merge revisions into the various revisions of which it is composed (using the "use-merge-history" option of the "svn log" command).

Table 6.1 presents the size of these projects in terms of source lines of code (SLOC),

[6]svn log. http://svnbook.red-bean.com/en/1.7/svn.ref.svn.c.log.html

the studied development history, the number of added logging statements in the history, and the number of added logging statements that experience a log level change afterwards. The *Hadoop* project is the largest project with 458K of SLOC, while *Hama* is the smallest project, with an SLOC of 39K. In the studied history, the number of added logging statements within these projects ranges from 1,683 (for *Hama*) to 5,388 (for *Hadoop*); 1.2% to 4.6% of the added logging statements experience a log level change eventually.

6.2.2 Data Extraction

Figure 6.3 presents an overview of our data extraction and model analysis approaches. From the version control repository of each subject project, we collect all the revisions during the development history of the subject project. For each revision, we use the "svn diff" command to obtain the code changes in that revision. Then, we use a regular expression to identify the newly-added logging statements in each revision and extract the log level of each logging statement. The regular expression is derived from the format of the logging statements as specified by the used logging libraries. To achieve an accurate model, we remove all newly-added logging statements that experience a log level change afterwards, because the levels of these changed logging statements may have been inappropriate in the first place.

6.3 Preliminary Study

We first perform an empirical study on the usage of log levels in the four studied open source projects.

Table 6.1: Overview of the studied projects.

Project	**SLOC**	**Studied development history**	**Added logging statements**	**Log level changes**[1]
Hadoop	458 K	2009-05 to 2014-07	5,388	163 (3.0%)
Directory Server	119 K	2006-01 to 2014-06	5,035	58 (1.2%)
Hama	39 K	2008-06 to 2014-07	1,683	54 (3.2%)
Qpid	271 K	2006-09 to 2014-07	4,712	216 (4.6%)
TOTAL	887 K	-	16,818	491 (2.9%)

[1] The number of added logging statements that experience a modification of their log level after their introduction

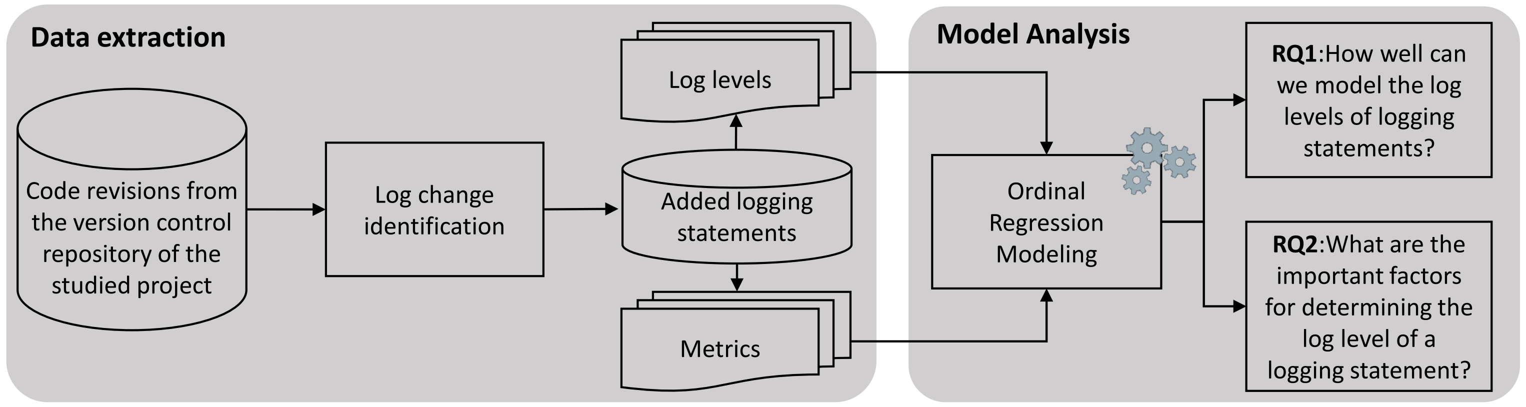

Data extraction
Code revisions from the version control repository of the studied project
Log change identification
Added logging statements
Log levels
Metrics
Model Analysis
Ordinal Regression Modeling
RQ1:How well can we model the log levels of logging statements?
RQ2:What are the important factors for determining the log level of a logging statement?

Figure 6.3: An overview of our data extraction and analysis approaches.

6.3.1 Log level distribution in the studied projects

No single log level dominates all other log levels. As shown in Figure 6.4, developers tend to use a variety of log levels. Compared to *trace* and *fatal*, the four middle levels (*debug, info, warn, and error*) are used more frequently. For the *Directory Server*, *Hadoop* and *Hama* projects, more than 95% of the logging statements use one of the four middle levels. For the *Qpid* project, 86% of the logging statements use one of the four middle levels. As the least verbose log level, the *fatal* level is the least frequently used level (less than 2%) in the four projects. The reason may be that *fatal* issues are unlikely to appear in these projects. As the most verbose log level, the *trace* level is only used in less than 4% of the logging statements in the *Directory Server, Hadoop* and *Hama* projects. The low usage of the trace level may be due to developers not typically using logs to trace their software, but rather they might use existing tracing tools such as JProfiler[7]. However, the *trace* level is used in 14% of the logging statements in the *Qpid* project.

Each project exhibits a varying distribution of log levels. Three of the four studied projects leverage all six log levels (*trace, debug, info, warn, error, and fatal*), while *Directory Server* uses only five log levels (no *fatal*). *Directory Server* shows frequent usages of the *debug* and *error* levels, while the logging statements that are inserted in the *Hadoop* project are more likely to use the *info*, *debug* and *warn* levels. For both *Hama* and *Qpid*, the *debug, info* and *error* levels are most frequently used in their logging statements. On the one hand, the different distributions can be explained by different usage of logs in these open source projects that are from different domains. For example, if logs are used mainly for bookkeeping purposes, there might be more *info* logs; *debug* logs are

[7] http://www.ej-technologies.com/products/jprofiler/overview.html

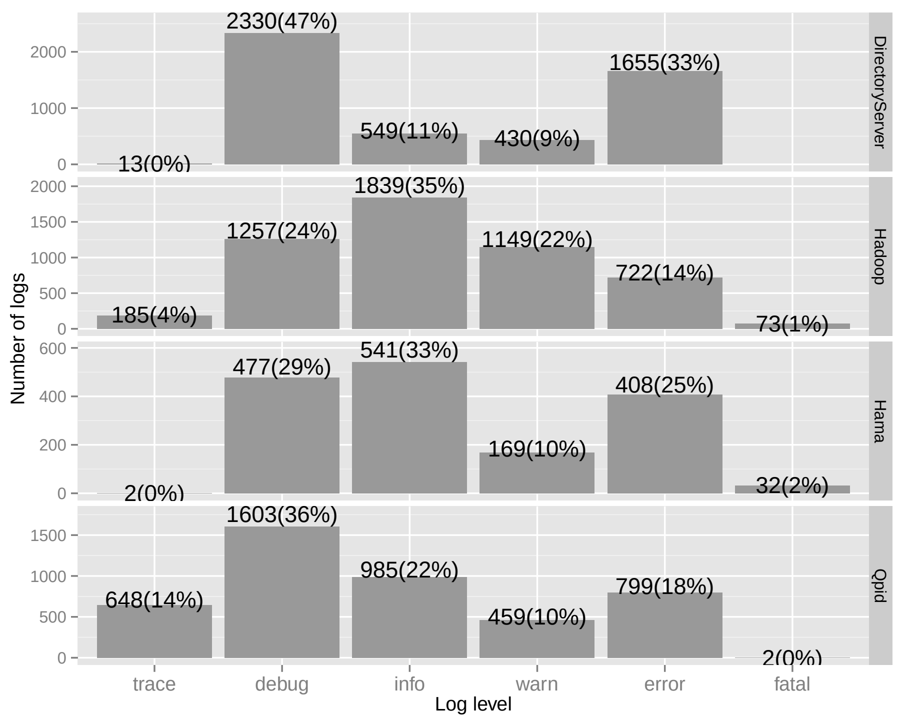


Figure 6.4: Log level distribution in the added logging statements.

widely used for debugging purposes; if developers use logs for monitoring, there might be more *warn* and *error* logs. On the other hand, such differences might be the results of a lack of standard guidelines for determining log levels, which motivates our work to assist developers in determining the most appropriate log level for their logging statements. Our preliminary analysis highlights that each studied project appears to follow a different pattern for its use of log levels. Hence, we believe that our choice of projects ensure a heterogeneity in our studied subjects.

6.3.2 Log level distribution in different blocks

Logging statements have different distribution of log levels across the different containing blocks. In this chapter, a "block" (or "code block") refers to a block of source code which is treated as a programming unit. For example, a *catch* block is a block of source code which completes a *catch* clause. The "containing block" of a logging statement is the smallest (or innermost) block that contains the logging statement. In other words, there is no intermediate block that is contained in the particular block and that contains the particular logging statement. Similarly, when we say a logging statement is "directly inserted into a block, we mean that there is no other intermediate block that contains the particular logging statement. We use an abstract syntax tree (AST) parser provided by the Eclipse JDT[8] to identify the containing block of each logging statement. We consider seven types of containing blocks that cover more than 99% of all the logging statements inserted in the studied projects: *try* blocks, *catch* blocks, *if-else* (or *if* for short) blocks, *switch-case-default* (or *switch* for short) blocks, *for* blocks, *while* or *do-while* (using *while* for short) blocks, and *methods*. We do not consider other types of blocks (e.g., *finally* blocks) because very few logging statements are inserted in the other types of blocks (less than 1% in total). Figure 6.5 shows the distributions of log levels for the logging statements that are inserted directly in the seven types of blocks. The percentage numbers marked on the stacked bars describe the distribution of log levels that are used in the logging statements that are inserted in each type of block; the number above each stack shows the total number of logging statements that are inserted in each type of block. We find that the top two most-frequently used log levels for each type of block are used in more than 60% of the logging statements that are inserted in that particular type of block. In the *Directory Server* project, for example, more than 79% of the logging statements that are inserted in each type of block use the top two most-frequently used log levels. However, our findings highlight that the

[8]https://eclipse.org/jdt/

choice of log level is not a simple one that can be easily determined by simply checking the containing block of a logging statement. Instead determining the appropriate log level requires a more elaborated model and such a model is likely to vary across projects.

Logging statements that are directly inserted in *catch* blocks tend to use less verbose log levels, while logging statements in *try* blocks are more likely to use more verbose log levels. 77% to 91% of the logging statements that are inserted in *catch* blocks use the *warn*, *error* and even *fatal* log levels. In contrast, 96% to 100% of the logging statements inserted in *try* blocks adopt more verbose log levels (i.e., *info*, *debug* and *trace*). *Logging statements* in *try* blocks are triggered during the normal execution of an application, thus they usually print the normal run-time information of the application using the more verbose log levels; while *logging statements* in *catch* blocks are only triggered when exceptions occur, hence they often log abnormal conditions using the less verbose log levels.

Logging statements that are directly inserted in loop blocks (i.e., *for* and *while* blocks) and *methods* are usually associated with more verbose log levels. 87% to 97% of the logging statements that are directly inserted in *while* blocks, 94% to 100% of the logging statements in *for* blocks and 86% to 97% of the logging statements in *methods* choose more verbose log levels (i.e., *info*, *debug* and *trace*). The logging statements in loop blocks might be executed a large number of times, but they may not print logs in field execution when the verbosity level is set at a less verbose level (e.g., *warn*); in other words, these logging statements will take effect only when the verbosity level is set at a more verbose level (e.g., *debug*), i.e., when application users need the detailed information from logs. The logging statements directly inserted in *methods* typically

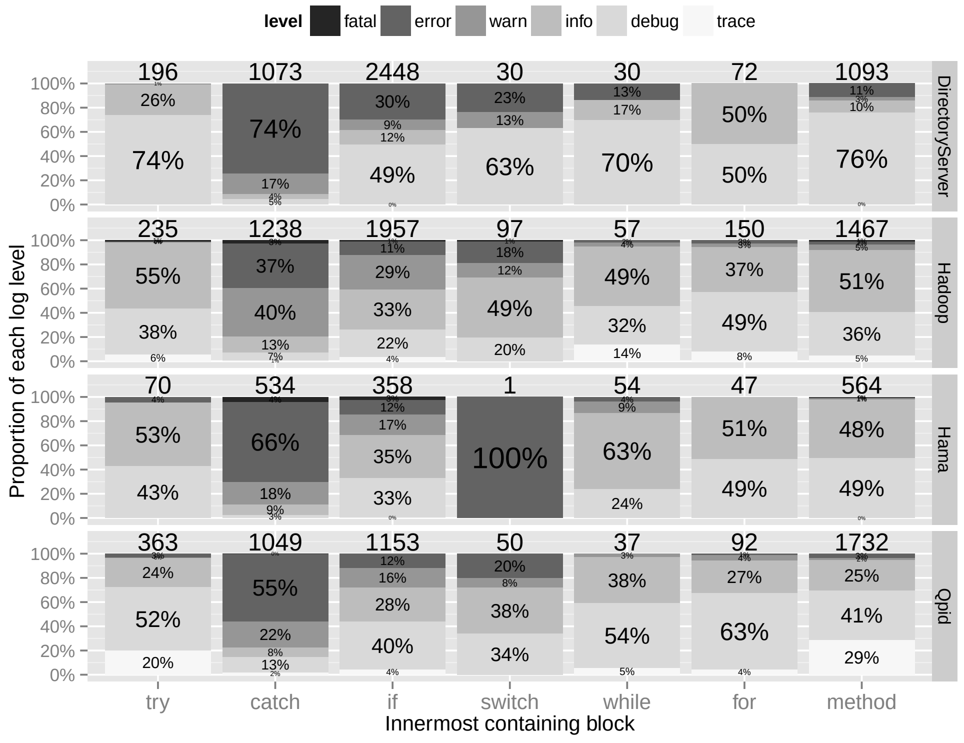


Figure 6.5: Log level distribution in the added logging statements in different types of blocks.

record some expected runtime events, such as startup or shutdown, thus they usually use more verbose log levels such as *info* or *debug*. Figure 6.5 also shows that different projects log loops and methods at different log levels. For example, *Hadoop* tends to log *methods* at the *info* level, while *DirectoryServer* uses more *debug* level logging statements in the *method* blocks. Again, this might be explained by different usage of logs in these open source projects from different domains.

6.3.3 Log level distribution in catch blocks

A common best practice for exception-handling is to log the information associated with the exception (Microsoft-MSDN, 2016). Logging libraries like Log4j even provide special methods for logging exceptions. In our preliminary study, we also find that logging statements present much higher density in *catch* blocks than any other blocks. However, experts argue that "not all exceptions are errors" (Eberhardt, 2014), and that exceptions are sometimes anticipated or even expected (Zhu et al., 2015). Therefore, blindly assigning the same log level for all logging statements that are within *catch* blocks may result in inappropriate log levels.

Logging statements that are directly inserted into *catch* blocks present different distribution of log levels for different types of handled exceptions. Figure 6.6 illustrates the log level distribution of the logging statements inserted in the top 12 types of exception-catching blocks that contain the most logging statements, for the *Qpid* project. For some types of exception-handling blocks, such as the *catch* blocks that handle the *DatabaseException* and the *OpenDataException*, all the inserted logging statements use the *error* or *warn* levels, indicating that these exceptions lead to problems and the developers or users may need to take care of the abnormal condition. On the other hand, 89% of the logging statements inserted in *catch* blocks dealing with the *QpidException* choose the *debug* log level, which implies that the exception is not a serious one and that the code can itself handle the condition; or that the exception is simply used by developers for debugging purpose. For each exception type, the top two most-frequently used log levels cover more than 60% of the logging statements that are inserted in the particular exception handling blocks.

Not all the exceptions are logged as *warn*, *error* or *fatal*, which matches with

experts' knowledge that "not all exceptions are errors" (Zhu et al., 2015; Eberhardt, 2014). For most types of exceptions (10 out of 12 as shown in Figure 6.6), there are at least a small portion of logging statements inserted in the handling *catch* blocks to choose more verbose log levels (i.e., *info*, *debug* or *trace*). Therefore, developers should be careful when they insert the *warn* or less verbose level logging statements into exception-handling blocks.

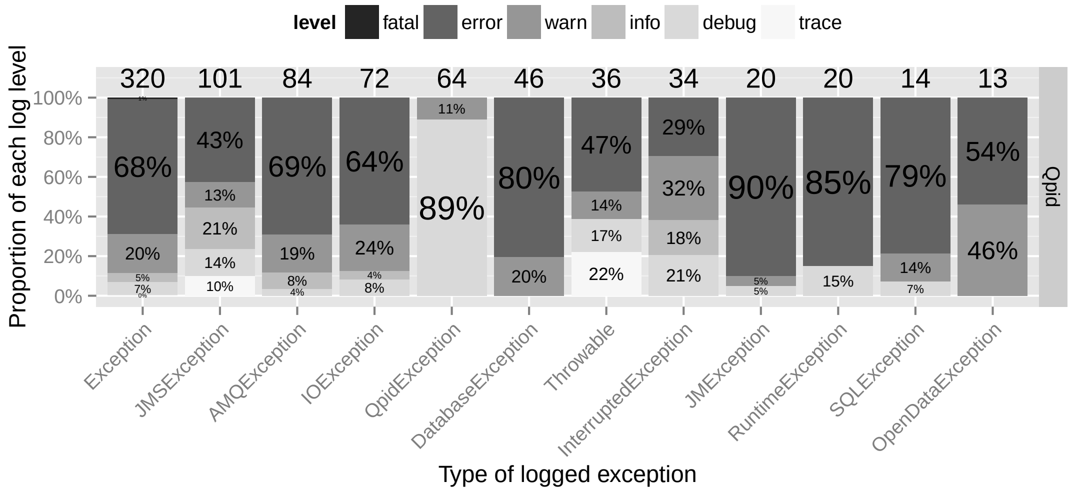


Figure 6.6: Log level distribution in the added logging statements in different types of exception-catching blocks (Qpid).

6.4 Case Study Results

In this section, we present the results of our research questions. For each research question, we present the motivation of the research question, the approach that we used to address the research question, and our experimental results.

6.4.1 RQ1: How well can we model the log levels of logging statements?

Motivation

In order to help developers select the appropriate log level for a newly-added logging statement, we build a regression model to predict the appropriate log level using a set of software metrics. Developers can leverage such a model to receive suggestions on the most appropriate log level for a newly-added logging statement or to receive warnings on an inappropriately selected log level.

Approach

In order to model the log levels of the newly-added logging statements, we extract and calculate five dimensions of metrics: logging statement metrics, file metrics, change metrics, historical metrics, and containing block metrics.

- **Logging statement metrics** measure the characteristics of the newly-added logging statement itself. It is intuitive that the level of a logging statement is highly influenced by the content of the logging statement itself, e.g., the static text.
- **Containing block metrics** characterize the blocks that contain the newly-added logging statements. The containing block determines the condition under which a logging statement would be triggered, thus it is reasonable to consider the containing block when choosing the log level for a logging statement.
- **File metrics** measure the characteristics of the file in which the logging statement is added. Logging statements in the same file may share the same purpose of logging or log the same feature. Hence information derived from the containing

file may influence the choice of the appropriate log level.

- **Change metrics** measure information about the actual code changes associated with the newly-added logging statement. The characteristics of the code changes in a revision might indicate developers' purpose of adding logging statements in that revision thereby affecting the choice of log levels.
- **Historical metrics** record the code changes in the containing file in the development history. Stable code might no longer need detailed logging, hence the newly-added logging statements in stable code are more likely to use less verbose log levels (e.g., *error, warn*). The source code undergoing frequent changes might contain logging statements with more verbose log levels for debugging purposes.

Table 6.2 presents a list of all the metrics that we collected along the five dimensions. Table 6.2 describes the definition of each metric and explains our motivation behind the choice of each metric.

Re-encoding categorical metrics. In order to integrate the *containing block type* metric (with categorical values) as an independent variable in our regression analysis, we need to convert the categorical metric to a quantitative variable. A categorical variable of k categories can be coded into $k-1$ dummy variables, which contain the complete information of the categorical variable. In this chapter we use the weighted effect coding method, since it is most appropriate when the values of a categorical variable have a substantially unbalanced distribution (e.g., the unbalanced distribution of the containing blocks of logging statements, as shown in Figure 6.5) (Aguinis, 2004; Cohen et al., 2013). We use the weighted effect coding method to convert the categorical variable *containing block type* (with seven categories) into six dummy variables: *try block*,

Table 6.2: Software metrics used to model log levels.

Dimension	Metric name	Definition (d) | Rationale (r)
Logging statement metrics	Text length	d: The length of the static text of the logging statement.
		r: Longer logging statements are desirable for debugging purposes where detailed log information is needed; however, a too long logging statement might cause noise in scenarios like event monitoring where only less verbose log information is needed.
	Variable number	d: The number of variables in the logging statement.
		r: Logging more variables is desirable for debugging purposes while too many logged variables might cause noise in scenarios in need of only less verbose log information.
	Log tokens[1]	d: The tokens that compose the content (static text and variables) of the logging statement, represented by the frequency of each token in the logging statement.
		r: The content of a logging statement communicates the logging purpose thereby affecting the log level.
Containing block metrics	Containing block SLOC	d: Number of source lines of code in the containing block.
		r: The length of code in the containing block of the logging statement indicates the amount of effort that is spent on handling the logged event, which might be associated with the seriousness (i.e., verbosity) level of the logged event.
	Containing block type (Categorical)[2]	d: The type of the containing block of the logging statement, including seven categories: *try* block, *catch* block, *if* block, *switch* block, *for* block, *while* block and *method* block.
		r: Different types of blocks tend to be logged with different log levels. For example, *catch* blocks are more likely to be logged with less verbose log levels; logging statements inside *try* blocks are more likely to have more verbose log levels; and logging statements inside a loop are more likely to have more verbose log levels.
	Exception type	d: The exception type of the containing *catch* block, represented by the average level of **other** logging statements that are inserted in the *catch* blocks that handle the same type of exception. This metric is only valid when a logging statement is enclosed in a *catch* block.
		r: Different types of exceptions are logged using different log levels.
File metrics	Log density	d: The number of logging statements divided by the number of lines of code in the containing file.
		r: Source code with denser logs tends to record detailed run-time information and have more verbose logging statements.
	Log number	d: The number of logging statements in the containing file.
		r: Source code with more logs tend to record detailed run-time information and have more verbose logging statements.
	Average log length	d: Average length of the static text of the logging statements in the containing file.
		r: The average log length in a file might indicate the overall logging purpose in the file, e.g., having shorter and simpler log text is more likely to be associated with debugging purpose. The overall logging purpose affects the choice of log level for individual logging statements.
	Average log level	d: Average level of **other** logging statements in the containing file, obtained by quantifying the log levels into integers and calculating the average.
		r: The level of the added logging statement is likely to be similar with other existing ones in the same file.
	Average log variables	d: Average number of variables in the logging statements in the containing file.
		r: The average number of log variables in a file might indicate the overall logging purpose in the file, e.g., having more and detailed log variables is more likely to be associated with debugging purpose. The overall logging purpose affects the choice of log level for individual logging statements.
	SLOC	d: Number of source lines of code in the containing file.
		r: Large source code files are often bug-prone (Shihab et al., 2010; D'Ambros et al., 2012), thus they are likely to have more verbose logging statements for debugging purposes.
	McCabe complexity	d: McCabe's cyclomatic complexity of the containing file.
		r: Complex source code files are often bug-prone (Shihab et al., 2010; D'Ambros et al., 2012), thus they are likely to have more verbose logging statements for debugging purposes.
	Fan in	d: The number of classes that depend on (i.e., reference) the containing class of the logging statement.
		r: Classes with a high fan in, such as library classes, are likely to use less verbose logging statements; otherwise these logging statements will generate noise in the dependent code.

Dimension	Metric name	Definition (d) | Rationale (r)
Change metrics	Code churn	d: Number of changed source lines of code in the revision.
		r: When developers change a large amount of code, they might add more-verbose log information (i.e., for tracing or debugging purposes).
	Log churn	d: Number of changed logging statements in the revision.
		r: When many logging statements are added in a revision, these loggings statements tend to record detailed run-time information and have more verbose levels. For example, developers are more likely to add a large number of debugging or tracing logging statements rather than a large number of logging statements that record error information.
	Log churn ratio	d: Ratio of the number of changed logging statements to the number of changed lines of code.
		r: A lower log churn ratio indicates that developers only use logging statements for more important events (i.e., using less verbose log levels), while a high log churn ratio indicates that developers also use logging statements for less important events (i.e., using more verbose log levels).
Historical metrics	Revisions in history	d: Number of revisions in the development history of the containing file.
		r: Frequently-changed code is often bug-prone (Graves et al., 2000; Hassan, 2009; D'Ambros et al., 2012). Such code tends to have more more-verbose level logging statements for debugging purpose.
	Code churn in history	d: The total number of lines of code changed in the development history of the containing file.
		r: Source code that experienced large code churn in history is often bug-prone (Graves et al., 2000; Hassan, 2009; D'Ambros et al., 2012). Such code tends to have more more-verbose level logging statements for debugging prupose.
	Log churn in history	d: The total number of logs changed in the development history of the containing file.
		r: The log churn in history might reflect the overall logging purposes thereby affecting the choices of log levels. For example, frequently-changed logging statements are more likely to be used by developers for debugging or tracing purposes, and the logging statements that generate less verbose information are expected to be stable.
	Log churn ratio in history	d: Ratio of the number of changed logging statements to the number of changed lines of code in the development history of the containing file.
		r: The log churn ratio in history might reflect the overall logging purposes thereby affecting the choices of log levels. For example, a high log churn ratio in history might indicate that logging statements are used to record detailed events thus more verbose log levels should be used.
	Log-changing revisions in history	d: Number of revisions involving log changes in the development history of the containing file.
		r: The number of log-changing revisions in history might reflect the overall logging purposes thereby affecting the choices of log levels. For example, a file that experienced many log-changing revisions might indicate that the functionalities in the file is not stable, thus more detailed logging statements might be used for debugging or tracing purposes.

[1] Each token actually represents an independent variable (i.e., taken-based variable) in the ordinal regression model, and the value of a token-based variable is the frequency of the token in the logging statements, or zero if the token does not exist in the logging statement. The vast majority of these token-based variables are filtered out in the "backward (step-down) variable selection" step.
[2] The metric is re-encoded into several dummy variables to be used int the ordinal regression model. This section has a detailed description about the re-encoding approach.

catch block, *if block*, *switch block*, *for block*, and *while block*, where each of them represents whether a logging statement is directly inside the corresponding code block; and we use the "method block" value of the *containing block type* metric as a "reference group" (Aguinis, 2004; Cohen et al., 2013). For example, if the value of the *containing block type* metric is "try block", we code the six dummy variables as 1, 0, 0, 0, 0, and 0, respectively; if the value of the *containing block type* metric is "method block", we code the six variables as $-n_t/n_m$, $-n_c/n_m$, $-n_i/n_m$, $-n_s/n_m$, $-n_f/n_m$ and $-n_w/n_m$, respectively, where n_m, n_t, n_c, n_i, n_s, n_f and n_w represent the number of instances of the *containing block type* metric that have the value of *method block*, *try block*, *catch block*, *if block*, *switch block*, *for block* and *while block*, respectively.

Interaction between metrics. The *excecption type* metric is only valid when a logging statement is enclosed in a *catch* block. Therefore, there is a significant interaction between the *exception type* metric and the encoded variable *catch block*. We use the interaction (i.e., the product) between these two variables in our modeling analysis, and ignore the two individual variables. We hereafter use the term "catch block" to represent the interaction.

Correlation analysis. Before constructing our ordinal regression models, we calculate the pairwise correlation between our collected metrics using the Pearson correlation test (r). Specifically, we use the `varclus` function (from the R package `Hmisc` (Harrell et al., 2014)) to cluster metrics based on their Pearson correlation. In this work, we follow prior work (McIntosh et al., 2014) and choose the correlation value of 0.7 as the threshold to remove collinear metrics; Kuhn and Johnson (2013) also suggests a similar choice of the threshold value (0.75). If the correlation between a pair of metrics is greater than 0.7 ($|r| > 0.7$), we only keep one of the two metrics in the model.

Figure 6.7 shows the result of the correlation analysis for the *Directory Server* project, where the horizontal bridge between each pair of metrics indicates the correlation, and the red line represents the threshold value (0.7 in our case). To make the model easy to interpret, from each group of highly-correlated metrics, we try to keep the one metric that is more directly associated with logs. For example, the *log churn* and *code churn* metrics have a correlation higher than 0.7, thus we keep the *log churn* metric and drop (i.e., do not consider in the model) the *code churn* metric. Based on the result shown in Figure 6.7, we drop the following metrics: *code churn*, *SLOC*, *McCabe complexity*, *code churn in history*, *log-changing revisions in history* and *revisions in history*, due to the high correlation between them and other metrics. We find that our selected metrics present similar patterns of correlation across all four studied projects, thus we drop the same metrics for all the studied projects. Dropping the same set of metrics for the projects enables us to compare the metric importance of different projects (in "RQ2") and perform cross-project evaluation (in the "Discussion" section).

Ordinal regression modeling. We build ordinal regression models (McKelvey and Zavoina, 1975; McCullagh, 1980), to suggest the most appropriate log level for a given logging statement. The ordinal regression model is an extension of the logistic regression model; instead of predicting the dichotomous values as what the logistic regression does, the ordinal regression model is used to predict an ordinal dependent variable, i.e., a variable with categorical values where the relative ordering between different values is important. In our case, the ordinal response variable (log level) has six values, i.e., *trace*, *debug*, *info*, *warn*, *error* and *fatal*, from more verbose levels to less verbose levels. We use the "orm" function from the R package "rms" (Harrell, 2015b).

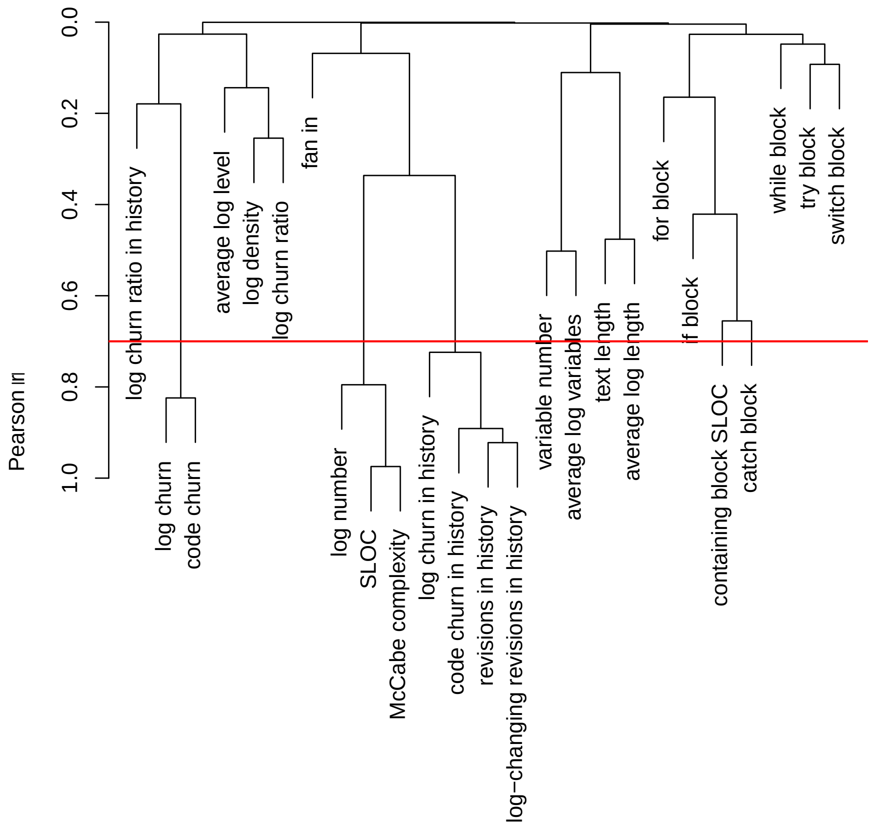


Figure 6.7: Correlation analysis using Spearman hierarchical clustering (for Directory Server). The red line indicates the threshold (0.7) that is used to remove collinear metrics.

The outcome of an ordinal regression model is the cumulative probabilities of each ordinal value[9]. Specifically, in this case the ordinal regression model generates the cumulative probability of each log level, including $P[level \geq debug]$, $P[level \geq info]$, $P[level \geq warn]$, $[level \geq error]$ and $P[level \geq fatal]$. The list does not include $P[level \geq trace]$ because the probability of log levels greater than or equal to *trace* is always 1.

[9] http://www.inside-r.org/packages/cran/rms/docs/orm

We calculate the predicted probability of each log level by subtracting between the cumulative probabilities:

- $P[level = trace] = 1 - P[level \geq debug]$
- $P[level = debug] = P[level \geq debug] - P[level \geq info]$
- $P[level = info] = P[level \geq info] - P[level \geq warn]$
- $P[level = warn] = P[level \geq warn] - P[level \geq error]$
- $P[level = error] = P[level \geq error] - P[level \geq fatal]$
- $P[level = fatal] = P[level \geq fatal]$

We then select the log level with the highest probability as the predicted log level.

Backward (step-down) variable selection. We use the metrics defined in Table 6.2 as the independent (predictor) variables to build the ordinal regression models. However, not all the independent variables are statistically significant in our models. Therefore, we use the backward (step-down) variable selection method (Lawless and Singhal, 1978) to determine the statistically significant variables that are included in our final regression models. The backward selection process starts with using all the variables as predictor variables in the model. At each step, we remove the variable that is the least significant in the model. This process continues until all the remaining variables are significant (i.e., $p < 0.05$). We choose the backward selection method since prior research shows that backward selection method usually performs better than the forward selection approach (i.e., adding one statistically significant variable to the model at a time) (Mantel, 1970). We use the `fastbw` (Fast Backward Variable Selection) function from the R package `rms` (Harrell, 2015b) to perform the backward variable selection process.

Evaluation technique. We measure the performance of our ordinal regression models using the multi-class Brier score and AUC metrics.

Brier score (BS) is commonly used to measure the accuracy of probabilistic predictions. It is essentially the mean squared error of the probability forecasts (Wilks, 2011). The Brier score was defined by Brier (1950) to evaluate multi-category prediction. It is calculated by

$$BS = \frac{1}{N}\sum_{i=1}^{N}\sum_{j=1}^{R}(f_{ij} - o_{ij})^2 \tag{6.1}$$

where N is the number of prediction events (or number of added logging statements in our case), R is the number of predicted classes (i.e., or number of log levels in the level modeling case), f_{ij} is the predicted probability that the outcome of event i falls into class j, and o_{ij} takes the value 1 or 0 according to whether the event i actually occurred in class j or not. The Brier score ranges from 0 to 2 (Brier, 1950). The lower the Brier score, the better the performance of the model.

AUC (area under the curve) is used to evaluate the degree of discrimination that is achieved by a model. The AUC is the area under the ROC (receiver operating characteristic) curve that plots the true positive rate against the false positive rate. The AUC value ranges between 0 and 1. A high value for the AUC indicates a high discriminative ability of a model; an AUC of 0.5 indicates a performance that is no better than random guessing. In this chapter, we use an R implementation (Cullmann, 2015) of a multiple-class version of the AUC, as defined by Hand and Till (2001).

The *Brier score* measures the error between the predicted probabilities and the actual observations, i.e., how likely the predicted log level is equal to the actual log level. A probabilistic prediction may assign an extremely high possibility (e.g., 100%) to the correct log level, or assign a probability to the correct category that is only slightly higher than the probability of an incorrect log level. The Brier score evaluation favors the former case to measure the model's ability to predict the correct category accurately.

The *AUC* give us an insight on how well the model can discriminate the log levels, e.g., how likely a model is able to predict an actual *error* level as *error* correctly, rather than predict an actual *info* level as *error* with false positive. In particular, the AUC provides a good performance measure when the distribution of the predicted categories is balanced (Kuhn and Johnson, 2013). Figure 6.4 shows that some log levels are less frequently used in the studied projects, hence harder to predict. A higher AUC ensures the model's performance when predicting such log levels. A prior study (Mant et al., 2009) also uses a combination of AUC and Brier score to get a more complete evaluation of the performance of a model.

Bootstrapping and optimism. The Brier score and AUC provide us an insight on how well the models fit the observed dataset, but they might overestimate the performance of the models when applied to future observations (Efron, 1986; McIntosh et al., 2016). Bootstrapping (Efron, 1979) is a general approach to infer the relationship between sample data and the population, by resampling the sample data with replacement and analyzing the relationship between the resample data and the sample data. In order to avoid overestimation (or *optimism*), we subtract the bootstrap-averaged *optimism* (Efron, 1986) from the original performance measurement (i.e., Brier score and AUC). The *optimism* values of the Brier score and AUC are calculated by the following steps, similar to prior research (McIntosh et al., 2016):

- Step 1. From the original dataset with N logging statements (i.e., instances), we randomly select a bootstrap sample of N instances with replacement. On average, 63.2% of the logging statements in the original dataset are included in the bootstrap sample for at least once (Kuhn and Johnson, 2013).

- Step 2. We build an ordinal regression model using the bootstrap sample (which contains averagely 63.2% of the logging statements in the original dataset).
- Step 3. We test the model (built from bootstrap sample) on the bootstrap and original datasets separately, and calculate the Brier score and AUC on both datasets.
- Step 4. We measure the *optimism* by calculating the difference between the model's performance (Brier score and AUC) on the bootstrap sample and on the original dataset.

The steps are repeated for 1,000 times to ensure that our random sampling is not biased (Harrell, 2015a). We calculate the average *optimism* values of the Brier score and AUC, and then obtain the *optimism-reduced* Brier score and AUC by subtracting the *optimism* from the original values. The *optimism-reduced* Brier score and AUC values give us an indication of how well we can expect the model to fit to the entire dataset (but not just the sample or the observed data).

Baseline models. We compare the performance of our ordinal regression models with two baseline models: a random guessing model, and a naive model based on the proportional distribution of log levels. The random guessing model predicts the log level of a logging statement to be each candidate level with a identical probability of $1/R$, where R is the number of candidate levels. The intuition of the naive model is that when a developer does not know the appropriate log level, a default log level of the project may be chosen for the logging statement. However, we do not know the default log level of each project. Therefore, for each project we calculate the proportion of each log level used in the logging statements, and use the proportion as the predicted probability of that particular level. In other words, the naive model allocates the predicted probability of the candidate log levels according to the proportional distribution of the log levels in that particular project.

Results

The ordinal regression model achieves an optimism-reduced Brier score of 0.44 to 0.66 and an optimism-reduced AUC of 0.75 to 0.81. An AUC of 0.75 to 0.81 indicates that the ordinal regression model performs well in discriminating the six log levels, and that the model can accurately suggest log levels for newly-added logging statements. As shown in Table 6.3, the original Brier score for the ordinal regression model, which measures the model's performance when both of the training data and the testing data is the whole data, ranges from 0.43 to 0.65. The optimism-reduced Brier score of the ordinal regression model, which measures the model's performance when the model is trained on a subset (bootstrapped resample) of the data while tested on the whole data, ranges from 0.44 to 0.66, with only a difference between 0 and 0.01. The difference between the original AUC and the optimism-reduced AUC is also very small, ranging from 0 to 0.01. The negligible difference between the original and the optimism-reduced performance values indicates that the model is not over-fitted to the training data. Instead the model can also be effectively applied to a new data set. In other words, the performance of our models, when applied on new data in practice, would only exhibit very minimal degradation.

We also measure the computational cost for determining the appropriate log level for a newly-added logging statement. On average, training an ordinal regression model for a large project like Hadoop on a workstation (Intel i7 CPU, 8G RAM) takes less than 0.3 seconds, and predicting the log level for a newly-added logging statement takes less than 0.01 seconds. For each commit, we would only need to perform the prediction step in real-time, while the training can be done offline. Our approach can provide suggestions for log levels in an interactive real-time manner.

The performance of the ordinal regression model outperforms that of the naive model and the random guessing model. For all the four studied projects, as presented in Table 6.3, the ordinal regression model achieves a higher AUC than the baseline models (the random guessing model and the naive model), by 0.25 to 0.31. The significantly higher AUC of the ordinal regression model indicates that it outperforms the baseline models in discriminating the six log levels. For three out of the four studied projects, *Directory Server*, *Hama* and *Qpid*, the Brier score of the ordinal regression model is better than that of the baseline models. The Brier score of the ordinal regression model outperforms the random guessing model by 0.28 to 0.36. The Brier score of the ordinal regression model outperforms the naive model by 0.21 to 0.22.

For the *Hadoop* project, the Brier score of the ordinal regression model is less significantly higher than that of the baseline models: the ordinal regression model gets a Brier score of 0.66, while the naive model and the random guessing model get a Brier score of 0.75 and 0.83, respectively. A possible reason for the ordinal regression model's performance degradation in the *Hadoop* project is that the *Hadoop* project presents a more variant usage of log levels in each type of block. As shown in Figure 6.5, for the *Hadoop* project, the most-frequently used log level in each type of block only covers

Table 6.3: Comparing the performance of the ordinal regression model with the random guessing model and the naive model.

Project	**Ordinal model**		**Naive model**		**Random guess**	
	Brier Score[1]	AUC[2]	Brier Score	AUC	Brier Score	AUC
D. Server	0.44 (0.43)	0.78(0.78)	0.65	0.50	0.80	0.50
Hadoop	0.66 (0.65)	0.76(0.76)	0.75	0.50	0.83	0.50
Hama	0.51 (0.50)	0.75(0.76)	0.73	0.50	0.83	0.50
Qpid	0.55 (0.55)	0.81(0.82)	0.76	0.50	0.83	0.50

[1] The value outside the parenthesis is the optimism-reduced Brier score, and the value inside the parenthesis is the original Brier score.
[2] The value outside the parenthesis is the optimism-reduced AUC, and the value inside the parenthesis is the original AUC.

33% to 55% of the logging statements that are inserted in that particular type of block. However, the proportions of the most-frequently used log level in each type of block are more dominating in other studied projects, ranging 49% - 76%, 35% - 66%, and 38% - 63% for the *Directory Server*, *Hamma* and *Qpid* projects, respectively. For the *Hadoop* project, the top two most-frequently used log levels in each type of block are used in 62% to 93% of the logging statements that are inserted in that particular type of block. The top two most-frequently used log levels in each type of block cover 79% to 100%, 68% to 100%, and 68% to 92% of the logging statements for the *Directory Server*, *Hama*, and *Qpid* projects, respectively. The variant usage of log levels in each type of block makes it hard for a model to achieve an accurate probabilistic prediction. However, a high AUC of 0.76 still indicates that the ordinal regression model performs well in discriminating the six log levels for the logging statements in the *Hadoop* project.

The ordinal regression models can effectively suggest log levels with an AUC of 0.75 to 0.81 and a Brier score of 0.44 to 0.66, which outperforms the performance of the naive model based on the log level distribution and a random guessing model.

6.4.2 RQ2: What are the important factors for determining the log level of a logging statement?

Motivation

In order to understand which factors (metrics) play important roles in determining the appropriate log levels, we analyze the ordinal regression models to get the relative importance of each variable. Understanding the important factors can provide software practitioners insight regarding the selection of the most appropriate log level for a newly-added logging statement.

Approach

Wald Chi-Square (χ^2) test. We use the Wald statistic in a Wald χ^2 maximum likelihood test to measure the importance of a particular variable (i.e., calculated metric) on model fit. The Wald test tests the significance of a particular variable against the null hypothesis that the corresponding coefficient of that variable is equal to zero (i.e., $H_0 : \theta = 0$) (Harrell, 2015a). The Wald statistic about a variable is essentially the square of the coefficient mean ($\hat{\theta}$) divided by the variance ($var(\theta)$) of the coefficient (i.e., $W = \frac{\hat{\theta}^2}{var(\theta)}$). A larger Wald statistic indicates that an explanatory variable has a larger impact on the model's performance, i.e., model fit. The Wald statistic can be compared against a chi-square (χ^2) distribution to get a p-value that indicates the significance level of the coefficient. We use the term "Wald χ^2" to represent the Wald statistic hereafter. Prior research has leveraged Wald χ^2 test in measuring the importance of variables (McIntosh et al., 2016; Sommer and Huggins, 1996). We perform the Wald χ^2 test using the `anova` function provided by the `rms` package (Harrell, 2015b) of R.

Joint Wald Chi-Square (χ^2) test. In order to control for the effect of multiple metrics in each dimension, we use a joint Wald χ^2 test (a.k.a, “chunk test”) (Harrell, 2015a) to measure the joint importance of each dimension of metrics. For example, we test the joint importance of the *text length*, *variable number* and *log tokens* metrics, and get a single Wald χ^2 value to represent the joint importance of the *logging statement metrics* dimension. The Wald χ^2 value resulted form a joint Wald test on a group of metrics is not simply the sum of the Wald χ^2 values that are resulted from testing the importance of the corresponding individual metrics; instead, the Wald test measures the joint importance of a group of metrics by testing the null hypothesis that all metrics in the group have a coefficient of zero (i.e., $H_0 : \theta_0 = \theta_1 = ...\theta_{k-1} = 0$, where k is the number of metrics in the group for joint Wald test) (Harrell, 2015a). The larger the Wald χ^2 value, the larger the joint impact that a group of metrics have on the model's performance. We also use the `anova` function from the R package `rms` (Harrell, 2015b) to perform the joint Wald χ^2 test.

Results

The *containing block metrics*, which characterize the surrounding block of a logging statement, play the most important roles in the ordinal regression models for log levels. Table 6.4 shows the Wald χ^2 test results for all the individual metrics that are used in our final ordinal regression models. As listed in the “Sig.” columns, all the final metrics that we used in our ordinal regression models are statistically significant in our models (i.e., $p < 0.05$), since we used the backward variable selection approach to remove those insignificant metrics. Table 6.5 shows the joint Wald χ^2 test results for each dimension of metrics. The dimension of *containing block metrics* is the most important one (i.e., with the highest χ^2) in the models for the *Haodop* and the *Hama* projects; and it is the second most important dimension of metrics in the models for the *Directory Server* and the *Qpid* projects. Specifically, both the *containing block type* and the *containning block SLOC* metrics are statistically significant in the models for all four studied projects. Moreover, the *containing block type* metric as a individual metric plays the most important role in the models for the *Hama* project; and it is the second most important metric for the models for the other three projects. The *containing block SLOC* metric is the fourth important metric in the models for the *Hadoop* and *Hama* projects, while it plays less important roles in the models for the other two projects. Developers need to consider the characteristics of the surrounding block (e.g., block type) of a newly-added logging statement to determine the most appropriate log level for the particular logging statement.

The *file metrics*, which capture the overall logging practices in the containing files, are also important factors for predicting the log level of a newly-added logging statement. As shown in Table 6.5, the *file metrics* is the most important dimension in the ordinal regression models for the *Qpid* project, the second most important for the *Hadoop* project, and the third most important for the *Directory Server* and *Hama* projects. As shown in Table 6.4, in particular, the *average log level* metric is the most important individual metric for determining the log levels for the *Hadoop* and *Qpid* projects; and it is the third and fourth important one in the *Hama* and *Directory Server* projects, respectively. Such a result suggests that the logging statements that are inserted in the same file are likely to use similar log levels; this might be explained by the intuition that these logging statements share the same purposes of logging and they

Table 6.4: Variables' importance in the ordinal models, represented by the Wald Chi-square. The percentage following a Wald χ^2 value is calculated by dividing that particular Wald χ^2 value by the "TOTAL" Wald χ^2 value.

Directory Server			**Qpid**		
Variable name	**Wald χ^2**	**Sig.**[1]	**Variable name**	**Wald χ^2**	**Sig.**
Log tokens	555 (29.3%)	***	Average log level	937 (34.8%)	***
Containing block type	507 (26.8%)	***	Containing block type	460 (17.1%)	***
Variable number	217 (11.4%)	***	Log number	238 (8.8%)	***
Average log level	200 (10.6%)	***	Log tokens	207 (7.7%)	***
Text length	123 (6.5%)	***	Log churn	78 (2.9%)	***
Average log variable	46 (2.4%)	***	Average log variable	37 (1.4%)	***
Average log length	24 (1.3%)	***	Containing block SLOC	27 (1.0%)	***
Log number	23 (1.2%)	***	Fan in	20 (0.7%)	***
Log density	23 (1.2%)	***	Log density	11 (0.4%)	**
Containing block SLOC	8 (0.4%)	**	Text length	9 (0.3%)	**
Fan in	5 (0.3%)	*	TOTAL	2692 (100.0%)	***
Log churn ratio	5 (0.3%)	*			
TOTAL	1894 (100.0%)	***			
Hadoop			**Hama**		
Variable name	**Wald χ^2**	**Sig.**	**Variable name**	**Wald χ^2**	**Sig.**
Average log level	494 (23.1%)	***	Containing block type	257 (29.4%)	***
Containing block type	385 (18.1%)	***	Log tokens	96 (10.9%)	***
Log tokens	171 (8.0%)	***	Average log level	88 (10.1%)	***
Containing block SLOC	136 (6.4%)	***	Containing block SLOC	37 (4.2%)	***
Log number	76 (3.6%)	***	Log number	20 (2.2%)	***
Log churn in history	42 (2.0%)	***	TOTAL	876 (100.0%)	***
Text length	29 (1.4%)	***			
Average log variable	20 (0.9%)	***			
Log churn ratio in hist.	14 (0.6%)	***			
TOTAL	2134 (100.0%)	***			

[1] Statistical significance of explanatory power according to Wald χ^2 likelihood ratio test:
o $p \geq 0.05$; * $p < 0.05$; ** $p < 0.01$; *** $p < 0.001$

log the same or closely-connected features. Other metrics in the *file metrics* dimension - the *log number, log density, average log variables, average log length* and *fan in* metrics - are also statistically significant in the log level models. These metrics together capture the overall characteristics of the logging practices in a source code file. Developers should always keep in mind the overall logging characteristics (e.g., the log level

Table 6.5: The joint importance of each dimension of metrics in the ordinal models, calculated using the joint Wald test. The percentage following a Wald χ^2 value is calculated by dividing that particular Wald χ^2 value by the "TOTAL" Wald χ^2 value.

	Directory Server		**Hadoop**		**Hama**		**Qpid**	
Metric dimension	**Wald χ^2**	**Sig.**[1]	**Wald χ^2**	**Sig.**	**Wald χ^2**	**Sig.**	**Wald χ^2**	**Sig.**
Containing block metrics	640 (33.8%)	***	894 (41.9%)	***	577 (65.8%)	***	723 (26.9%)	***
File metrics	258 (13.6%)	***	549 (25.7%)	***	91 (10.3%)	***	1134 (42.1%)	***
Logging statement metrics	889 (46.9%)	***	189 (8.9%)	***	96 (10.9%)	***	224 (8.3%)	***
Change metrics	5 (0.3%)	*	0 (0.0%)	o	0 (0.0%)	o	78 (2.9%)	***
Historical metrics	0 (0.0%)	o	67 (3.1%)	***	0 (0.0%)	o	0 (0.0%)	o
TOTAL	1894 (100.0%)	***	2134 (100.0%)	***	876 (100.0%)	***	2692 (100.0%)	***

[1] Statistical significance of explanatory power according to Wald χ^2 likelihood ratio test:
o $p \geq 0.05$; * $p < 0.05$; ** $p < 0.01$; *** $p < 0.001$

of the existing logging statements) in the same file when determining the appropriate log level for a newly-added logging statement.

The *logging statement metrics*, which measure the content of a logging statement, also play important roles in explaining the log level for a newly-added logging statement. As shown in Table 6.5, the *logging statement metrics* is the most important dimension for explaining the log level for a newly-added logging statement in the *Direcotory Server* project. It is the second most important metric for explaining the log levels for the *Hama* project, and the third important metric for explaining the log levels for the *Hadoop* and *Qpid* projects. In particular, the *log tokens* metric is the most import individual metric for explaining the log levels for the *Directory Server* project; and it is the second, third, and forth most important metric for explaining the log levels for the *Hama*, *Hadoop* and *Qpid* projects, respectively. We investigated why the *log tokens* metric is more important for explaining the log levels for the *Directory Server* project than other projects. We find that the *Directory Server* project uses the least variety of tokens in their logging statements. Specifically, on average each logging statement in the *Directory Server* project contributes 0.08 unique tokens, while on average each logging statement contributes 0.12 to 0.21 unique tokens in other studied projects. The uniqueness of tokens in *Directory Server* makes it more certain to determine the appropriate log levels using such information. Other metrics in the *logging statement metrics* dimension - the *text length* and *variable number* metrics - are also among the most important metrics for explaining the log levels for the *Directory Server* project; however, these two metrics are less important or even not statistically significant for explaining the log levels for the other three projects. In order to find the root cause of this discrepancy, we dig into the *text length* and *variable number* metrics for the four studied projects. We find that the *Directory Server* project generally uses significantly shorter text but more variables in the error level logs which are the most popular ones in the *Directory Server* project (see Figure 6.4). Therefore, these two metrics have great explanatory power to the levels of the logging statements in the *Directory Server* project. However, we do not find similar patterns in the other three projects. In order to determine the most appropriate log level for a newly-added logging statement, developers should not only refer to the overall logging characteristics of the containing file and the containing block, but should also pay attention to the content of the logging statement itself.

The *change metrics* and *historical metrics* are the least important in explaining the choices of log levels. As shown in Table 6.5, the *change metrics* and *historical metrics* are the least important dimensions in the ordinal regression models for all studied projects. The log level of a newly-added logging statement is not significantly impacted by the characteristics of the code change that introduces the logging statement. The log level of a newly-added logging statement is also not significantly impacted by the characteristics of the previous code changes that affect the containing file of the logging statement. The appropriate log level is more likely to be influenced by the static characteristics of the source code, rather than the change history of the source code. These results suggest that we should focus our effort on the current snapshot of the source code, rather than the development history, to determine the appropriate log level for a newly-added logging statement.

The characteristics of the containing block, the existing logging statements in the containing file, and the content of a newly-added logging statement play important roles in determining the appropriate log level for the newly-added logging statement. The appropriate log level is more likely to be influenced by the current snapshot of the source code, rather than the development history of the source code.

6.5 Discussion

Cross-project Evaluation. Since small projects or new projects might not have enough history data for log level prediction, we also evaluate our model's performance in cross-project prediction. We train a model using a combo data of $N-1$ projects (i.e., the training projects), and use the model to predict the log levels of the newly-added logging statements in the remaining one project (i.e., the testing project). We use the AUC and the Brier score to evaluate the performance of the cross-project prediction models.

To avoid the unbalanced number of newly-added logging statements for each project in the training data, we leverage up-sampling to balance the training data such that each project has the same number of newly-added logging statements in the training data. Specifically, we keep unchanged the largest training project in the training data; while we randomly up-sample the entries of the other training projects with replacement to match the number of entries of the largest training project. In order to reduce the non-determinism caused by the random sampling, we repeat the "up-sampling - training - testing" process for 100 times and calculate the average AUC and Brier score values.

Table 6.6 lists the performance of the cross-project models. Each row of the table shows the performance of the model that uses the specified project as testing data and

Table 6.6: The results of the cross-project evaluation.

Project	**Brier score**	**AUC**
DirectoryServer	0.67	0.76
Hadoop	0.77	0.72
Hama	0.57	0.71
Qpid	0.70	0.80

all the other projects as training data. The cross-project models reach a Brier score of 0.57 to 0.77 and an AUC of 0.71 to 0.80.

Comparing Table 6.6 and Table 6.3, we find a significant performance degradation of the ordinal regression model when applied in cross-project prediction. The accuracy of probability prediction decreases significantly: the Brier score increases by 0.06 to 0.23. A likely explanation for the performance degradation is about the different distribution of log levels in different projects, as shown in Figure 6.4. Another explanation might be that the most important factors for the log level models are different among the studied projects. However, the AUC only decreases by 0.01 to 0.04. The cross-project models still have the ability to discriminate different log levels for newly-added logging statements. Overall, our cross-project evaluation results suggest that one is better off building separate models for each individual project.

Log Level Changes. We have left out all the newly-added logging statements that experience a later log level change in our study, as the levels of these logging statements may have been inappropriate in the first place. Developers change the log level of a logging statement either to fix an inappropriate log level or because the logging requirement has changed.

As shown in Table 6.1, there are 491 added logging statements that experience a later log level change in the four studied projects. Table 6.7 summarizes the patterns

Table 6.7: Summary of the patterns of log level changes in the four studied projects. Each row represents a original log level and each column represents a new log level.

	trace	debug	info	warn	error	fatal
trace	0	25	2	0	0	0
debug	16	9[1]	41	3	7	1
info	8	211	7	13	4	0
warn	0	23	35	0	16	3
error	0	12	23	23	2	4
fatal	0	1	0	1	1	0

[1] Sometimes a log level is eventually changed back to the same level after some changes.

of log level changes that these 491 logging statements experience. Each row in the table represents the original level of an added logging statement, and each column represents the final level of the logging statements after one or more log level changes. 211 out of the 491 logging statements undergo a log level change from the *info* level to the *debug* level. On the other hand, 41 out of the 491 logging statements have their log level changed from the *debug* level to the *info* level. It seems that developers often change between the *info* and *debug* levels; the changes between *info* and *debug* levels represent 51% of all the log level changes. Other notable level change patterns include: *warn* to *info*, *trace* to *debug*, *warn* to *debug*, *error* to *info*, *error* to *warn*, *warn* to *error*, and *debug* to *trace*.

We notice that 354 (72%) out of the 491 logging statements have undergone a log level change from a less verbose level to a more verbose level (e.g., from *info* to *debug*); these changes will cause the software systems to generate lesser log information when more verbose log levels are not enabled. 119 (24%) out of the 491 logging statements have their log levels changed from a more verbose log level to a less verbose log level (e.g., from *debug* to *info*); these changes will cause the software systems to generate more log information when more verbose log levels are not enabled. 18 (4%) out of the 491 logging statements have their log levels changed to a different level, but eventually changed back to their initial log levels.

We also find that 385 (78%) out of the 491 logging statements have undergone a log level change to a log level that is only one level away from the original log level (e.g., from *debug* to *info* or from *error* to *warn*). 470 (96%) out of the 491 logging statements have their log levels changed to a level that is no more than 2 levels away (e.g., from *warn* to *debug*). Developers are likely to adjust their log levels between adjacent log levels rather than between log levels that are far apart. In this chapter we study the log levels of logging statements generally; however, future study should explore the confusion and distinction between adjacent log levels.

6.6 Threats to Validity

External Validity. The external threat to validity is concerned with the generalization of our results. In our work, we investigate four open source projects that are of different domains and sizes. However, since other software projects may use different logging libraries and apply different logging rules, the results may not generalize to other projects. Further, we only analyze Java source code in this study, thus the results may not generalize to projects programmed in non-Java languages. For example, other logging libraries in other programming languages may not support all the six log levels. Findings from more case studies on other projects, especially those with other programming languages and other logging libraries, can benefit our study.

Our preliminary study finds that different projects have different distribution of log levels. The results for RQ2 shows that the most important factors for the log level models are different among the studied projects. Besides, our cross-project evaluation highlights a significant performance degradation when our ordinal regression models are applied across projects. Therefore, in order to provide the most appropriate suggestion for the log level of a newly-added logging statement, it is recommended to build separate models for each individual project.

Internal Validity. The ordinal regression modeling results show the relationship between log levels and a set of software metrics. The relationship does not represent the casual effects of these metrics on log levels. The choice of log levels can be associated with many factors other than the metrics that we used to model log level. Future studies may extend our study by considering other factors.

In this chapter we study the appropriate choice of log level for a newly-added logging statement. We assume that in most cases developers of the studied projects are able to determine the most appropriate (i.e., consistent) log levels for their logging statements. However, the choices of log levels in the studied projects might not be always appropriate. To address this issue, we choose several successful and widely-used open source projects, and we remove all newly-added logging statements that experience a log level change later on in their lifetime.

This chapter studies the log level, which is fixed in the code, for a logging statement. The users of software systems that leverage log levels can configure at which verbosity level the logging statements should be printed, for different usage scenarios (e.g, debugging). In this chapter we do not capture the usage scenarios of the logging statements of these software systems. We may need to consider different usage scenarios of the logging statements to better determine the appropriate log levels in future work.

Construct Validity. This chapter proposes an approach that can provide developers with suggestions on the most appropriate log level when they add a new logging statement. Future work should conduct user studies with real developers to better evaluate how well our approach would perform in a real life setting.

We choose five dimensions of software metrics to model the appropriate log level of a logging statement. However, there might be other metrics, such as the characteristics of the logged variables, that can help improve the performance of our models. We expect future work to expand this study and consider more relevant software metrics.

6.7 Chapter Summary

Prior studies highlight the challenges that developers face when determining the appropriate log level for a newly-added logging statement. However, there is no standard or detailed guideline on how to use log levels appropriately; besides, we find that the usage of log levels varies across projects. We propose to address this issue (i.e., to provide suggestions on the choices of log levels) by learning from the prior logging practices of software projects. We first study the development history of four open source projects (*Directory Server*, *Hadoop*, *Hama*, and *Qpid*) to discover the distribution of log levels in various types of code snippets in these projects. Based on the insight from the preliminary study and our intuition, we propose an automated approach that leverage ordinal regression models to learn the usage of log levels from the existing logging practices, and to suggest the appropriate log levels for newly-added logging statements. Some of the key findings of our study are as follows:

- The distribution of log levels varies across different types of blocks, while the log levels in the same type of block show similar distributions across different projects.

- Our automated approach based on ordinal regression models can accurately suggest the appropriate log level of a newly-added logging statement with an AUC of 0.76 to 0.81 and a Brier score of 0.44 to 0.66. Such performance outperforms the performance of a naive model based on the log level distribution and a random guessing model.
- The characteristics of the containing block of a newly-added logging statement, the existing logging statements in the containing source code file, and the content of the newly-added logging statement play important roles in determining the appropriate log level for that logging statement.

Making appropriate choices of log levels is critical to balance the benefits and costs of logging (Chapter 3). Developers can leverage our models to receive automatic suggestions on the choices of log levels or to receive warnings on inappropriate usages of log levels. Our results also provide an insight on which factors (e.g., the containing block of a logging statement, the existing logging statements in the containing source code file, and the content of a logging statement) that developers consider when determining the appropriate log level for a newly-added logging statement.

CHAPTER 7

Automated Suggestions for Logging Stack Traces

As discussed in Chapter 3, logging the stack trace of an exception is important to assist developers in failure diagnosis. However, stack traces can grow log files very fast and easily frustrate the end users of a software system. In Chapter 3, we also observe that developers have difficulties to decide whether to log the stack trace of an exception. Therefore, in this chapter, we propose an automated approach to help developers make informed decisions about whether to print the stack trace of an exception in a logging statement. We use static program analysis to extract the contextual information of a logging statement in a catch block (e.g., exception type), and construct Random Forest models to suggest whether a logging statement in a catch block should log the stack trace of an exception. Our experimental results on four open source projects show that our automated approach can accurately suggest whether to print the stack trace of an exception in a logging statement, with a precision of 0.81 to 0.87, a recall of 0.81 to 0.89, and an AUC of 0.85 to 0.94. Our findings also provide developers and researchers insights into the important factors (e.g. exception types) that drive developers' decisions of logging exception stack traces.

7.1 Introduction

MODERN logging libraries (e.g., Log4j[1] and SLF4J[2]) usually support convenient ways to log the stack trace of an exception in a logging statement. For example, in the code snippet shown in Figure 7.1, the stack trace of an exception is logged by specifying the exception object as the last parameter of a logging statement. In this chapter, we name a logging statement in a catch block as an *exception logging statement*. An exception logging statement can either log the stack trace of an exception or not. Logging the stack traces of exceptions is very important to assist developers in diagnosing field failures (Han et al., 2012; Schroter et al., 2010; Chapter 3). However, stack traces can usually grow the log files very fast, lead to performance overhead, and easily frustrate the end users of a software system (Chapter 3).

As discussed in Chapter 3, developers usually have difficulties to balance the benefits and costs of logging stack traces and they raise conflicting concerns about whether to log the stack trace of an exception in a logging statement. For example, issue report HADOOP-10571[3] proposes to add stack traces to many exception logging statements across modules. However, other developers raise concerns that stack traces should be avoided for some of these exception logging statements. As a result, it takes significant efforts (e.g., as much as 10 patches) to resolve the conflicting concerns. Table 7.1 lists ten issue report examples that are concerned with whether to log the stack trace of an exception. Seven of the issue report examples request to add missing stack traces in exception logging statements, while the other three issue report examples request to remove existing stack traces from exception logging statements. These examples

[1]https://logging.apache.org/log4j/2.x/
[2]https://www.slf4j.org
[3]https://issues.apache.org/jira/browse/HADOOP-10571

```
try {
  Thread.sleep(detectionInterval);
} catch (InterruptedException e) {
  /* Logging the stack trace by specifying the exception (e) as the last
     parameter of a logging statement. */
  LOG.error("Disk Outlier Detection thread interrupted", e);
}
```

Figure 7.1: Example of logging the stack trace of an exception.

motivate us to develop automated approaches to help developers make appropriate decisions about whether to log the stack trace of an exception in the first place.

Prior work of “learning to log” learns statistical models from existing logging code to provide suggestions about *where to log* (Zhu et al., 2015; Jia et al., 2018; Chapter 4) and *which log level to use* (Chapter 6). However, prior work never provides suggestions about whether to log the stack trace of an exception. We believe that making appropriate decisions of logging stack traces is also very important to developers, because unnecessary stack traces can significantly increase the size of log files (i.e., a stack trace is usually much longer than a regular log message) and falsely alarm end users (i.e., a stack trace usually indicate a system problem).

In this work, we propose an automated approach to provide suggestions about whether to print the stack trace of an exception in an exception logging statement. We first use static program analysis to extract the contextual information of an exception logging statement (e.g. the exception type) and whether the exception logging statement logs the stack trace. Then, we construct Random Forest models to provide automated suggestions about whether an exception logging statement should log the stack trace of an exception. We perform a case study on four open source projects, namely *Hadodp*, *Directory Server*, *Hive*, and *Kafka*. Our experimental results show that

Table 7.1: Examples of issue reports of the *Hadoop* project that are concerned with whether to log the stack trace of an exception.

Issue ID[1]	**Issue report summary**	**Issue request**
HADOOP-14481	Print stack trace when native bzip2 library does not load	Add stack trace
HADOOP-13711	Supress CachingGetSpaceUsed from logging interrupted exception stacktrace	Remove stack trace
HADOOP-13682	Log missing stacktrace in catch block	Add stack trace
HADOOP-13458	LoadBalancingKMSClientProvider#doOp should log IOException stacktrace	Add stack trace
HADOOP-12840	UGI to log@ debug stack traces when failing to find groups for a user	Add stack trace
HADOOP-12795	KMS does not log detailed stack trace for unexpected errors	Add stack trace
HADOOP-11868	Invalid user logins trigger large backtraces in server log	Remove stack trace
HADOOP-10571	Use Log.*(Object, Throwable) overload to log exceptions	Add stack trace
HADOOP-10562	Namenode exits on exception without printing stack trace in AbstractDelegationTokenSecretManager	Add stack trace
HADOOP-8711	Provide an option for IPC server users to avoid printing stack information for certain exceptions	Remove stack trace

[1] For more details about each issue, one can refer to its web link which is "https://issues.apache.org/jira/browse/" followed by the issue ID. For example, the link for the first issue is "https://issues.apache.org/jira/browse/HADOOP-14481".

our automated approach can accurately suggest whether to log the stack trace of an exception in an exception logging statement.

Chapter organization. The remainder of the chapter is organized as follows. Section 7.2 describes our case study methodology, covering our subject projects, data extraction and data analysis approaches. Section 7.3 presents our experimental results. Finally, Section 7.4 draws conclusions based on our presented findings.

7.2 Methodology

7.2.1 Overview

Figure 7.2 shows an overview of our approach for providing automated suggestions about whether to print the stack trace of an exception in an exception logging statement. From the source code of each studied open source project, we use static program analysis to search for all the exception logging statements (i.e., the logging statements within a catch block). We implemented an IntelliJ plugin based on the IntelliJ Platform SDK [4] to perform our static program analysis. For each exception logging statement, we then use our IntelliJ plugin to extract a label (i.e., whether a stack trace is logged) and a set of code features that capture the contextual information of the logging statement. Based on the extracted label and code features for each exception logging statement, we construct a Random Forest model to suggest whether an exception logging statement should log the stack trace of an exception. Based on our model, we analyze the feature importance to understand the important factors that explain the likelihood of printing an exception stack trace in a logging statement.

7.2.2 Subject Projects

We perform a case study on four open source projects, including *Hadoop*, *Directory Server*, *Hive*, and *Kafka*. Table 7.2 lists the overview information about our subject projects. *Hadoop* is a distributed computing framework that supports distributed storage and processing of big data sets. *Directory Server* is an embeddable directory server. *Hive* is a data warehouse that supports accessing distributed data sets using

[4]https://www.jetbrains.org/intellij/sdk/docs/welcome.html

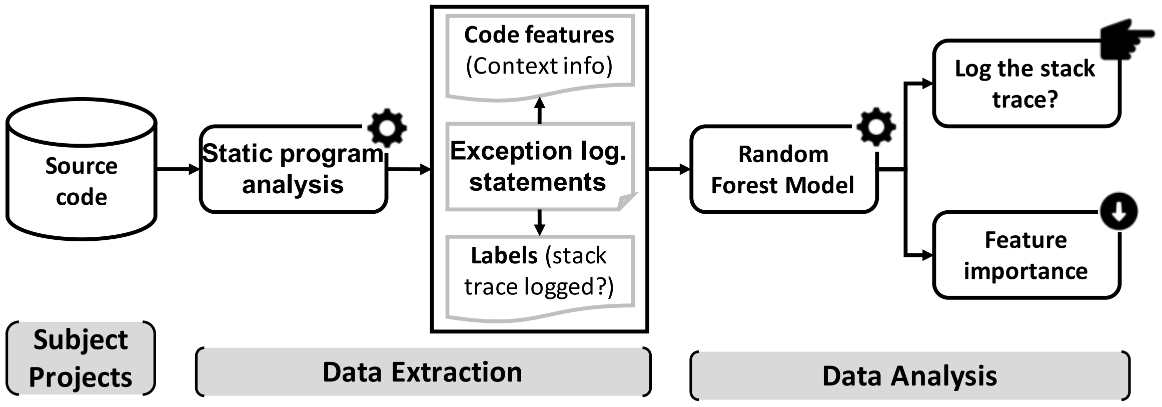


Figure 7.2: Overview of our approach for providing automated suggestions about whether to print the stack trace of an exception in an exception logging statement.

SQL. *Kafka* supports a streaming platform for messaging, storing and processing real-time records. Our case study projects are successful and mature projects with many years of development history. They represent different domains, which ensures that our findings are not limited to a particular domain.

All of these projects are primarily developed in Java. Table 7.2 shows the SLOC (Source Lines of Code) and the Java SLOC of the studied projects. In this chapter, we only consider the Java source code, and we exclude the testing code. The last column of Table 7.2 shows the number of exception logging statements in each studied project and the percentage of exception logging statement that print exception stack traces. *Hadoop* has the largest SLOC (i.e., 2,906 K) and largest number (i.e., 2,522) of exception logging statements. *Kafka* has the smallest number (i.e., 264) of exception logging statements but the largest percentage (i.e., 76%) of exception logging statements that print exception stack traces. *Directory Server* has the smallest SLOC (i.e., 245 K) and also the smallest percentage (i.e., 47%) of exception logging statements that print exception stack traces.

Table 7.2: Overview of the studied projects.

Project	**Studied release (Time of release)**	**SLOC**	**Primary Language (SLOC)**	**# Exception logging statements*** **(% stack traces logged)**
Hadoop	3.1.1 (2018.08)	2,906 K	Java (1,518 K)	2,522 (61%)
Directory Server	2.0.0.AM25 (2018.08)	245 K	Java (232 K)	391 (47%)
Hive	3.1.0 (2018.07)	1,721 K	Java (1,256 K)	1,615 (63%)
Kafka	2.0.0 (2018.07)	327 K	Java (214 K)	264 (76%)

* The number of exception logging statements for each project is calculated for the primary language.

7.2.3 Data Extraction

In order to explain the likelihood of logging an exception stack trace in an exception logging statement, we extract a label that captures whether a stack trace is printed in the logging statement and a set of code features that capture the contextual information of the logging statement. As shown in Figure 7.2, we use static program analysis (based on the IntelliJ Platform SDK) to search for all the exception logging statements in the source code, and then extract the label and code features for each exception logging statement. Below, we explain the label and the code features that we extract for each exception logging statement.

Label (logging an exception stack trace or not): A boolean variable indicating whether an exception logging statement prints an exception stack trace.

Code features. We extract a number of code features for each exception logging statement to explain the likelihood of logging a stack trace in the logging statement. Table 7.3 lists our code features for each exception logging statement and explains our rationale for choosing each of these features. These features fall into six dimensions:

- **Logging statement features** capture the characteristics of an exception logging statement and its local code context (e.g., whether the logging statement is inside a loop).
- **Exception features** capture the characteristics of the exception that is caught by the containing *catch* block of an exception logging statement. If there are multiple exceptions caught by the same *catch* block, we use the first one.
- ***Catch* block features** capture the characteristics of the containing *catch* block of an exception logging statement.
- ***Try* block features** capture the characteristics of the pairing *try* block of the containing *catch* block of an exception logging statement.
- **Exception-throwing method features** capture the characteristics of the method in the *try* block that can throw the caught exception (i.e., declared in the *throws* clause). If there are multiple methods that can throw the caught exception, we use the first one.
- **Containing code features** capture the characteristics of the code structures beyond the *try*-*catch* scope, such as the fan-in of the containing method.

7.2.4 Data Analysis

Model construction and evaluation. Using these code features as explanatory variables and the labels as a response variable, we train Random Forest models to suggest the likelihood of logging an exception stack trace in an exception logging statement. We use 10-fold cross-validation to estimate the efficacy of our models. The whole data set is randomly partitioned into 10 sets of roughly equal size. One subset is used as the

Table 7.3: Selected code features that are relevant to the likelihood of logging the stack trace of an exception in an exception logging statement.

Dimension	**feature**	**Definition (d) | Rationale (r)**
Logging statement features	Log level	d: The verbosity level of the logging statement. r: Lower log levels (e.g., debug) are more likely to print stack traces for debugging.
	Log in loop	d: Is the logging statement contained in a loop? r: Logging statements in loops are more likely to produce excessive logging.
	Log in branch	d: Is the logging statement contained in a branch inside the containing catch block? r: Logging statements in a branch are less likely to be related to the exception.
Exception features	Exception type	d: Type of the exception that is caught by the containing *catch* block. r: Indicates the severity of a problem. Severe problems are more likely to need stack traces.
	Exception category	d: Category of the caught exception (runtime exception, error, or checked exception). r: Indicates the severity of the problem at a higher level.
	Exception source	d: Source of the exception class (i.e., from the JDK, libraries, or the project). r: Problems related to different sources might have different level of severity.
	Exception package	d: The containing package of the exception class. r: The exceptions that are defined in the same package might indicate similar problems.
Catch block features	*Throw* in *catch*	d: Does the containing *catch* block contain a *throw* statement? r: An exception is less likely to be logged with the stack trace if it is re-thrown.
	Return in *catch*	d: Does the containing *catch* block contain a *return* statement? r: An early *return* might indicate a severe problem.
	LOC before logging	d: Number of lines of code in the containing catch block that are prior to the logging statement. r: Indicates how the logging statement is related to the caught exception.
	LOC after logging	d: Number of lines of code in the containing catch block that are after the logging statement. r: Indicates how well the exception is handled. Well-handled exceptions are less likely to need stack traces for debugging.
	Method calls before logging	d: Number of method calls in the containing catch block that are prior to the logging statement. r: Indicates how the logging statement is related to the caught exception.
	Method calls after logging	d: Number of method calls in the containing catch block that are after the logging statement. r: Indicates how well the exception is handled.
Try block features	*Throw* in *try*	d: Does the *try* block contain a *throw* statement? r: Indicates a problem that might need a stack trace to assist in debugging.
	Return in *try*	d: Does the *try* block contain a *return* statement? r: An exception before a *return* might indicate a severe problem.
	Method calls in *try*	d: Number of method calls in the *try* block excluding the logging statement. r: Indicates if the pairing *catch* block handles a specific problem or general problems.
Exception-throwing method features	Exception method	d: The method that throws the caught exception, or the containing method if the exception is thrown by the try block. r: The exceptions that are triggered by the same method call might be logged in similar ways.
	Exception method source	d: The source of the exception-throwing method (i.e., from the JDK, libraries, or the project). r: Problems related to different sources might have different level of severity.
	Exception method package	d: The containing package of the method that throws the exception. r: The exceptions that are triggered by methods from the same package might be logged in similar ways.
Containing code features	Containing method fan-in	d: Number of usages of the containing method in the entire project. r: Methods called at many code locations tend to log the stack traces for debugging.
	Containing file	d: The containing file of the logging statement. r: Logging statements in the same file might share similar logging patterns.
	Containing package	d: The containing package of the logging statement. r: Logging statements in the same package might share similar logging patterns.

testing set (i.e., the held-out set) and the other nine subsets are used as the training set. We train our models using the training set and evaluate the performance of our models on the held-out set. The process repeats 10 times until all subsets are used as testing set once. We repeat the 10-fold cross-validation 10 times (i.e., repeated 10-fold cross-validation), which means 100 different held-out sets would be used to estimate the efficacy of our models.

There are some nominal features (e.g., "exception type") which have many classes. Directly using these features as model variables would lead to highly sparse feature vectors. Instead, we use a target coding method to transfer these nominal features into numerical variables. For example, for each exception type, we calculate the ratio of the instances that print stack traces against all the instances with the same exception type. We only use the training set to calculate the ratios.

In each fold of cross-validation, we use precision, recall, and AUC to measure the performance of our models. Precision measures the ratio of logging statements that are both predicted and observed to log stack traces against all the logging statements that are predicted to log stack traces. Recall measures the ratio of logging statements that are both predicted and observed to log stack traces against all the logging statements that are observed to log stack traces. AUC measures our models' ability to discriminate the logging statements that log stack traces from the logging statements that do not.

Feature importance. The random forest model measures the importance of a feature by permuting the value of the feature while keeping the values of the other features unchanged in the testing data (i.e., the so-called "OOB" data) (Breiman, 2001). The importance score of a feature measures the impact of such a permutation of the feature on the classification error rate (Breiman, 2002; Liaw and Wiener, 2002). For each of the

100 folds in our repeated 10-fold cross-validation, we measure the importance score of each of our features. As a result, we get 100 importance scores for each feature.

Scott-Knott clustering. In order to understand the important factors that explain the likelihood of logging an exception stack trace in a logging statement, we compare the average importance of our features. However, the differences among the importance of some features might actually be due to random variability. Thus we use a Scott-Knott (SK) algorithm (Scott and Knott, 1974) to partition all the features into statistically homogeneous groups. The SK algorithm hierarchically cluster the features into groups and uses the likelihood ratio test to judge the significance of the difference of the importance scores among the feature groups. As a result, the SK algorithm generates statistically distinct groups of features, i.e., the importance scores of the features in two different groups are significantly different (i.e., p-value < 0.05), while the importance scores of the features within the same group are not significantly different (i.e, p-value ≥ 0.05).

7.3 Experimental Results

Our Random Forest models accurately suggest the likelihood of logging a stack trace in an exception logging statement, providing a precision of 0.81 to 0.87, a recall of 0.81 to 0.89, and an AUC of 0.85 to 0.94. Table 7.4 shows the performance of our Random Forest models, for the *Hadoop*, *Directory Server*, *Hive* and *Kafka* projects. A high precision value indicates that a high percent of exception logging statements that are suggested to print stack traces actually need to do so. A high recall value indicates that our models can detect a high percent of exception logging statements that need to print stack traces. A high AUC value indicates that our models are very likely to distinguish

Table 7.4: The performance of our Random Forest models for suggesting the likelihood of logging a stack trace in an exception logging statement.

Project	**Precision**	**Recall**	**AUC**
Hadoop	0.81	0.81	0.85
Directory Server	0.86	0.88	0.94
Hive	0.86	0.87	0.90
Kafka	0.87	0.89	0.86

exception logging statements that need to print stack traces and those do not.

Table 7.4 shows that our models perform better in the *Directory Server*, *Hive* and *Kafka* projects than in the *Hadoop* project. These results might be because the *Directory Server*, *Hive* and *Kafka* projects follow more consistent logging patterns. In fact, the *Hadoop* project implements quite heterogeneous functionalities, such as distributed storage, distributed computing, and resource management, which might have different logging needs.

Table 7.4 also shows that our models have a better precision and recall in the *Kafka* project than in the *Directory Server* and *Hive* projects. On the other hand, our models have a better AUC in the *Directory Server* and *Hive* projects than in the *Kafka* project. These results might look contradictory. In fact, the contradiction can be explained by the class imbalance of the response variable (i.e., "logging an exception stack trace or not"). As shown in Table 7.2, 76% of exception logging statements in the *Kafka* project log stack traces. In contrast, only 47% to 63% of the exception logging statements in the *Directory Server* and *Hive* projects log stack traces. In case of class imbalance, AUC is likely to produce more meaningful contrasts between models than precision and recall (Kuhn and Johnson, 2013).

Our automated approach can accurately suggest the likelihood of logging a stack trace in an exception logging statement, providing a precision of 0.81 to 0.87, a recall of 0.81 to 0.89, and an AUC of 0.85 to 0.94.

The containing code features play the most important roles in explaining the likelihood of logging exception stack traces in two out of the four studied projects. Table 7.5 lists the top 10 most important features for explaining the likelihood of printing a stack trace in an exception logging statement. The containing code features, including "containing file" and "containing package", play the most important roles (i.e., with an importance rank of the first and the second, respectively) for determining whether an exception logging statement in the *Hadoop* and *Hive* projects need to print a stack trace. The containing code features are also important (i.e., with an importance rank of the second and the sixth) for explaining the likelihood of logging stack traces in the *Directory Server* project. **These projects tend to follow consistent logging practices (i.e., in terms of logging stack traces) within the same files and packages.**

The exception features and exception-throwing method features are consistently important in explaining the likelihood of logging exception stack trace in all the studied projects. In particular, the "exception type" feature is the first, second, third, and fifth important feature in *Directory Server*, *Kafka*, *Hive*, and *Hadoop*, respectively. The "exception method package", which captures the containing package of an exception-throwing method, ranks as the third, third, fourth, and fifth important feature in *Hadoop*, *Directory Server*, *Hive*, and *Kafka*, respectively. Therefore, **the type of an exception and the method that throws the exception are import to determine the likelihood of logging the stack trace of the exception**.

Table 7.5: The mean importance of the top 10 features for explaining the likelihood of logging a stack trace in an exception logging statement. These features are divided into distinct homogeneous groups by Scott-Knott clustering of their importance scores.

Hadoop			Directory Server		
Group	**Feature**	**Mean Importance**	**Group**	**Feature**	**Mean Importance**
1	Containing file	0.067	1	Exception type	0.162
2	Containing package	0.053	2	Containing package	0.120
3	Exception method package	0.052	3	Exception method package	0.108
4	Log level	0.034	4	Exception package	0.076
5	Exception type	0.031	5	Exception method	0.046
6	Exception method	0.020	6	Containing file	0.035
7	Exception package	0.014		LOC after logging	0.035
8	LOC before logging	0.009	7	Exception source	0.026
9	Method calls in try	0.009	8	Exception category	0.022
10	LOC after logging	0.007	9	Log level	0.017
Hive			Kafka		
Group	**Feature**	**Mean Importance**	**Group**	**Feature**	**Mean Importance**
1	Containing file	0.085	1	Log level	0.059
2	Containing package	0.062	2	Exception type	0.044
3	Exception type	0.043	3	Exception package	0.033
4	Exception method package	0.040	4	LOC after logging	0.032
5	Log level	0.027	5	Exception method package	0.029
	Exception package	0.026	6	Exception category	0.027
6	Exception method	0.019	7	Return in catch	0.021
7	LOC after logging	0.012	8	Exception method	0.016
	Method calls in try	0.011	9	Containing package	0.010
8	Method fan-in	0.008		Containing file	0.010

The log level of a logging statement plays an important role in explaining the likelihood of logging a stack trace in the logging statement. In particular, the "log level" feature is the most important feature for explaining the likelihood of logging exception stack traces in the *Kafka* project. The "log level" feature ranks as the fourth and fifth important feature in *Hadoop* and *Hive*, respectively. As discussed in Chapter 3, **logging a stack trace at a high level is likely to bring many logging costs, such as excessive log information and misleading end users**.

The containing code features, exception features, and exception-throwing method features play the most important roles in explaining the likelihood of logging the stack trace of an exception.

7.4 Chapter Summary

Logging the stack trace of an exception can bring both benefits (e.g., assisting in failure diagnosis) and costs (e.g., growing log files very fast). In Chapter 3, we observe that developers have difficulties to decide whether to log the stack trace of an exception. In this chapter, we propose an approach to provide automated suggestions about whether to print the stack trace of an exception in a logging statement. We first use static program analysis to extract the code features that capture the contextual information of an exception logging statement. We then construct Random Forest models to suggest whether to print an exception stack trace in an exception logging statement. Finally, we analyze the resulting models to understand the important factors the explain the likelihood of printing an exception stack trace in a logging statement. Our experimental results on four open source projects show that our automated approach can accurately suggest whether to print the stack trace of an exception in a logging statement, with a precision of 0.81 to 0.87, a recall of 0.81 to 0.89, and an AUC of 0.85 to 0.94. We find that the features that capture the containing file and package of an exception catch block, and the features that capture the characteristics of an exception itself and its exception-throwing method play the most important roles in explaining the likelihood of logging the stack trace of an exception.

Making appropriate decisions of logging exception stack traces is critical for balancing the benefits and costs of logging (Chapter 3). Our automated approach can help developers make informed decisions about whether to print an exception stack trace in a logging statement. Our findings also provide developers and researcher insights into the important factors (e.g., exception type) that drive developers' decisions of logging exception stack traces.

CHAPTER 8

Conclusions and Future Work

LOG messages provide valuable information for software practitioners to understand system runtime behaviors and diagnose field failures. However, developers typically insert logging statements in an *ad hoc* manner, which usually results in insufficient logging in some code snippets and excessive logging in other code snippets. This Ph.D. thesis aims to understand and support software logging practices through mining development knowledge (i.e., source code, code change history, and issue reports). We believe that development knowledge contains valuable information that explains developers' rationale behind their logging practices, which can help us better understand existing logging practices and develop helpful tools to assist developers in their logging practices. To evaluate our hypothesis, we mine different aspects of development knowledge: 1) mining issue reports to understand developers' logging concerns; 2) mining source code to understand how developers distribute their logging statements in the source code; 3) mining code change history to learn how developers develop and maintain their logging code, and how they choose log levels for their logging statements. Based on our empirical findings, we also propose automated approaches to support developers' logging decisions. Our findings and approaches make valuable contributions towards improving current logging practices and the qualify of logging code.

8.1 Thesis Contributions

Below, we highlight the main contributions of the thesis.

- **Understanding Developers' Logging Concerns**. We perform a qualitative study on 533 logging-related issue reports from three large and successful open source projects. We conceptualize developers' logging concerns (and how they address their concerns) into easy-to-perceive categories. We derive best logging practices and general logging advice along with our qualitative study, which can help developers improve their logging code and be aware of some logging traps. Besides, logging library providers can learn from our advice to improve their logging libraries. Our empirical findings also shed lights on future research opportunities for improving software logging.
- **Understanding Software Logging Using Topic Models**. We use LDA to extract the underlying topics from the source code, and study the relationship between

the logging decisions and the recovered topics. We find that a small number of topics, in particular, the topics that can be generalized to communication between machines or interaction between threads, are much more likely to be logged than other topics. We also find that topics can provide additional explanatory power over the structural information of a code snippet for explaining the likelihood of a code snippet being logged. Our findings suggest that future work on logging recommendation tools should consider topic information in order to help software practitioners make more informed logging decisions. Our findings also encourage future work to develop topic-influenced logging guidelines (e.g., which topics need further logging).

- **Automated Suggestions for Log Changes**. We empirically study why developers make log changes and propose an automated approach to provide developers with log change suggestions as soon as they commit a code change (i.e., just-in-time suggestions). Through a case study on four open source projects, we find that the reasons for log changes can be grouped along four categories: block change, log improvement, dependence-driven change, and logging issue. Our experimental results show that our automated approach can accurately provide just-in-time log change suggestions using a within and across projects evaluation. Our findings also demonstrate that developers can leverage machine learning models to guide their log changing practices.

- **Automated Suggestions for Choosing Log Levels**. We analyze the development history of four open source projects to study how developers assign log levels to their logging statements. We find that the distribution of log levels varies across different types of blocks, while the log levels in the same type of block show similar distributions across different projects. We also propose an automated approach based on ordinal regression models to help developers determine the most appropriate log level when they add a new logging statement. Our automated approach can accurately suggest the appropriate log level of a newly-added logging statement, outperforming the performance of a naive model based on the log level distribution and a random guessing model. Developers can leverage our approach to receive automatic suggestions on the choices of log levels or to receive warnings on inappropriate usages of log levels.

- **Automated Suggestions for Logging Stack Traces**. We observe that logging exception stack traces appropriately is crucial for balancing the benefits and costs of logging. We propose an automated approach that can accurately suggest whether to print the stack trace of an exception in a logging statement. Our automated approach can help developers make informed decisions regarding logging exception stack traces. Our findings based on our automated approach also provide developers and researchers insights into the important factors (e.g., exception types) that explain the likelihood of logging an exception stack trace.

8.2 Limitations

The first limitation is concerned with the generalization of the findings in this thesis. In order to understand and support software logging practices, we perform case studies on several open source projects. Although our findings are general among the studied projects, our findings may not generalize to other open source projects and closed source projects. For example, the log-intensive topics that we derive in Chapter 4 might vary in other software projects from different domains. Findings from additional case studies on other software projects can benefit our studies. However, the methodologies proposed in this thesis can apply to other software projects. For example, developers can apply our proposed approach in Chapter 4 to leverage the specific topics in their own projects to understand and guide their logging decisions.

The second limitation is that some findings in this thesis may seem obvious. For example, in Chapter 3, we find that developers are concerned about the logging cost of *excessive log information*, which might seem obvious to some software practitioners and researchers. However, as it is discussed in this thesis, software practitioners till face challenges to solve such “obvious” problems. Our work uses rigorous methods to help software practitioners and researchers better understand such problems. Our work also encourages future work to tackle these “obvious” problems that still concern software practitioners.

8.3 Future Research

This thesis highlights the need for standard logging guidelines and automated tooling support for software logging practices. Below, we propose some potential research opportunities that may benefit logging practices in the future.

8.3.1 Studying human aspects of software logging

Existing studies about software logging focus their analysis on the source code. However, the human aspect (e.g., developers) plays an important role in making logging decisions. For example, in Chapter 3, we find that different developers usually raise conflicting concerns about the same logging code. Therefore, it is very interesting to study how human aspects impact logging practices and to develop automated logging tooling that considers such human aspect.

8.3.2 Automated generation of logging code

Existing studies provide automated suggestions about *where to log*. However, there is no work yet that can automatically generate logging statements for developers (i.e., *what to log*). It will be a promising avenue to investigate approaches that can automatically generate logging statements based on the context code and the specified logging purposes.

8.3.3 Categorizing logging by their purposes

As discussed in Chapter 3, developers insert logging statements for different purposes (or benefits) (e.g., assisting in debugging, exposing runtime problems, and bookkeeping). For many usage scenarios, only one or two logging purposes need to be fulfilled. Therefore, it is valuable to develop approaches that can automatically categorize logging statements and log messages by their purposes, in order to help practitioners filter out unneeded log messages.

8.3.4 Detecting performance-critical logging statements

Performance overhead is a major cost for software logging. As discussed in Chapter 3, developers sometimes are not aware of such a cost and insert logging statements that significantly slow down their systems. Future studies are needed to help developers identify the logging code that brings too much performance overhead, through static analysis or dynamic analysis.

Bibliography

Aguinis, H. (2004). *Regression analysis for categorical moderators*. Guilford Press.

Apache-Commons (2016). Apache commons logging user guide - best practices. http://commons.apache.org/proper/commons-logging/guide.html. Accessed 25 July 2018.

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*, ICSE '10, pages 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*, OOPSLA '08, pages 543–562.

Bavota, G., Oliveto, R., Gethers, M., Poshyvanyk, D., and Lucia, A. D. (2014). Methodbook: Recommending move method refactorings via relational topic models. *IEEE Transactions on Software Engineering*, 40(7):671–694.

Binkley, D., Heinz, D., Lawrie, D., and Overfelt, J. (2014). Understanding LDA in source code analysis. In *Proceedings of the 22nd International Conference on Program Comprehension*, ICPC '14, pages 26–36.

Bitincka, L., Ganapathi, A., Sorkin, S., and Zhang, S. (2010). Optimizing data analysis with a semi-structured time series database. In *Proceedings of the 2010 Workshop on Managing Systems via Log Analysis and Machine Learning Techniques*, SLAML'10, pages 7–7.

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

Breiman, L. (2001). Random forests. *Machine learning*, 45(1):5–32.

Breiman, L. (2002). Manual on setting up, using, and understanding random forests v3.1. `https://www.stat.berkeley.edu/~breiman/Using_random_forests_V3.1.pdf`. Accessed 25 July 2018.

Brier, G. W. (1950). Verification of forecasts expressed in terms of probability. *Monthly weather review*, 78(1):1–3.

Bring, J. (1994). How to standardize regression coefficients. *The American Statistician*, 48(3):209–213.

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*, 18:467–479.

Chang, J., Gerrish, S., Wang, C., Boyd-graber, J. L., and Blei, D. M. (2009). Reading tea leaves: How humans interpret topic models. In *Advances in Neural Information Processing Systems 22*, pages 288–296.

Chen, B. and Jiang, Z. M. (2017a). Characterizing and detecting anti-patterns in the logging code. In *Proceedings of the 39th International Conference on Software Engineering*, ICSE '17, pages 71–81.

Chen, B. and Jiang, Z. M. J. (2017b). Characterizing logging practices in Java-based open source software projects – a replication study in apache software foundation. *Empirical Software Engineering*, 22(1):330–374.

Chen, T.-H., Shang, W., Hassan, A. E., Nasser, M., and Flora, P. (2016a). Cacheoptimizer: Helping developers configure caching frameworks for hibernate-based database-centric web applications. In *Proceedings of the 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, FSE '16, pages 666–677.

Chen, T.-H., Shang, W., Nagappan, M., Hassan, A. E., and Thomas, S. W. (2017a). Topic-based software defect explanation. *Journal of Systems and Software*, 129:79–106.

Chen, T.-H., Syer, M. D., Shang, W., Jiang, Z. M., Hassan, A. E., Nasser, M., and Flora, P. (2017b). Analytics-driven load testing: An industrial experience report on load testing of large-scale systems. In *Proceedings of the 39th International Conference on Software Engineering: Software Engineering in Practice Track*, ICSE-SEIP '17, pages 243–252.

Chen, T.-H., Thomas, S. W., and Hassan, A. E. (2016b). A survey on the use of topic models when mining software repositories. *Empirical Software Engineering*, 21(5):1843–1919.

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*, MSR '12, pages 189–198.

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*, 14(1):93–130.

Cohen, I., Goldszmidt, M., Kelly, T., Symons, J., and Chase, J. S. (2004). Correlating instrumentation data to system states: A building block for automated diagnosis and control. In *Proceedings of the 6th Conference on Symposium on Opearting Systems Design & Implementation - Volume 6*, OSDI' 04, pages 16–16.

Cohen, J., Cohen, P., West, S. G., and Aiken, L. S. (2013). *Applied multiple regression/-correlation analysis for the behavioral sciences*. Routledge.

Cullmann, A. D. (2015). *HandTill2001: Multiple Class Area under ROC Curve*. R package version 0.2-10.

D'Ambros, M., Lanza, M., and Robbes, R. (2012). Evaluating defect prediction approaches: a benchmark and an extensive comparison. *Empirical Software Engineering*, 17(4-5):531–577.

De Lucia, A., Di Penta, M., Oliveto, R., Panichella, A., and Panichella, S. (2012). Using

IR methods for labeling source code artifacts: Is it worthwhile? In *Proceedings of the 20th International Conference on Program Comprehension*, ICPC '12, pages 193–202.

De Lucia, A., Di Penta, M., Oliveto, R., Panichella, A., and Panichella, S. (2014). Labeling source code with information retrieval methods: an empirical study. *Empirical Software Engineering*, pages 1–38.

Eberhardt, C. (2014). The art of logging. http://www.codeproject.com/Articles/42354/The-Art-of-Logging. Accessed 25 July 2018.

Efron, B. (1979). Bootstrap methods: another look at the jackknife. *The Annals of Statistics*, 7(1):1–26.

Efron, B. (1986). How biased is the apparent error rate of a prediction rule? *Journal of the American Statistical Association*, 81(394):461–470.

Friedman, J., Hastie, T., and Tibshirani, R. (2010). Regularization paths for generalized linear models via coordinate descent. *Journal of Statistical Software*, 33(1):1–22.

Fu, Q., Lou, J.-G., Lin, Q., Ding, R., Zhang, D., and Xie, T. (2013). Contextual analysis of program logs for understanding system behaviors. In *Proceedings of the 10th Working Conference on Mining Software Repositories*, MSR '13, pages 397–400.

Fu, Q., Lou, J.-G., Wang, Y., and Li, J. (2009). Execution anomaly detection in distributed systems through unstructured log analysis. In *Proceedings of the 9th IEEE International Conference on Data Mining*, ICDM '09, pages 149–158.

Fu, Q., Zhu, J., Hu, W., Lou, J.-G., Ding, R., Lin, Q., Zhang, D., and Xie, T. (2014). Where do developers log? An empirical study on logging practices in industry. In *Companion Proceedings of the 36th International Conference on Software Engineering*, ICSE Companion ’14, pages 24–33.

Fukushima, T., Kamei, Y., McIntosh, S., Yamashita, K., and Ubayashi, N. (2014). An empirical study of just-in-time defect prediction using cross-project models. In *Proceedings of the 11th Working Conference on Mining Software Repositories*, MSR 2014, pages 172–181.

Ghotra, B., McIntosh, S., and Hassan, A. E. (2015). Revisiting the impact of classification techniques on the performance of defect prediction models. In *Proceedings of the 37th International Conference on Software Engineering - Volume 1*, ICSE ’15, pages 789–800.

Glerum, K., Kinshumann, K., Greenberg, S., Aul, G., Orgovan, V., Nichols, G., Grant, D., Loihle, G., and Hunt, G. (2009). Debugging in the (very) large: Ten years of implementation and experience. In *Proceedings of the ACM SIGOPS 22Nd Symposium on Operating Systems Principles*, SOSP ’09, pages 103–116.

Goshtasby, A. A. (2012). Similarity and dissimilarity measures. In *Image Registration: Principles, Tools and Methods*, pages 7–66. Springer London, London.

Graves, T. L., Karr, A. F., Marron, J. S., and Siy, H. (2000). Predicting fault incidence using software change history. *IEEE Transactions on Software Engineering*, 26(7):653–661.

Groeneveld, R. A. and Meeden, G. (1984). Measuring Skewness and Kurtosis. *Journal of the Royal Statistical Society. Series D (The Statistician)*, 33(4).

Gülcü, C. and Stark, S. (2003). *The complete log4j manual*. Quality Open Software, Lausanne, Switzerland.

Hall, D., Jurafsky, D., and Manning, C. D. (2008). Studying the history of ideas using topic models. In *Proceedings of the 2008 conference on empirical methods in natural language processing*, EMNLP '08, pages 363–371.

Han, S., Dang, Y., Ge, S., Zhang, D., and Xie, T. (2012). Performance debugging in the large via mining millions of stack traces. In *Proceedings of the 34th International Conference on Software Engineering*, ICSE '12, pages 145–155.

Hand, D. J. and Till, R. J. (2001). A simple generalisation of the area under the ROC curve for multiple class classification problems. *Machine learning*, 45(2):171–186.

Harrell, Jr., F. E. (2015a). *Regression modeling strategies: with applications to linear models, logistic and ordinal regression, and survival analysis*. Springer.

Harrell, Jr., F. E. (2015b). *rms: Regression Modeling Strategies*. R package version 4.4-1.

Harrell, Jr., F. E., with contributions from Charles Dupont, and many others. (2014). *Hmisc: Harrell Miscellaneous*. R package version 3.14-5.

Hassan, A. E. (2008). The road ahead for mining software repositories. In *Frontiers of Software Maintenance*, FoSM '08, pages 48–57.

Hassan, A. E. (2009). Predicting faults using the complexity of code changes. In *Proceedings of the 31st International Conference on Software Engineering*, ICSE '09, pages 78–88.

Hassan, A. E. and Holt, R. C. (2004). Predicting change propagation in software systems. In *Proceedings of the 20th IEEE International Conference on Software Maintenance*, ICSM '04, pages 284–293.

Hassan, A. E., Martin, D. J., Flora, P., Mansfield, P., and Dietz, D. (2008). An industrial case study of customizing operational profiles using log compression. In *Proceedings of the 30th International Conference on Software Engineering*, ICSE '08, pages 713–723.

Hindle, A., Bird, C., Zimmermann, T., and Nagappan, N. (2015). Do topics make sense to managers and developers? *Empirical Software Engineering*, 20(2):479–515.

Hu, J., Sun, X., Lo, D., and Li, B. (2015). Modeling the evolution of development topics using dynamic topic models. In *Proceedings of the 22nd IEEE International Conference on Software Analysis, Evolution, and Reengineering*, SANER' 15, pages 3–12.

Jelihovschi, E. G., Faria, J. C., and Allaman, I. B. (2014). Scottknott: A package for performing the scott-knott clustering algorithm in R. *Trends in Applied and Computational Mathematics*, 15(1):3–17.

Jia, Z., Li, S., Liu, X., Liao, X., and Liu, Y. (2018). Smartlog: Place error log statement by deep understanding of log intention. In *Proceedings of the 25th IEEE International Conference on Software Analysis, Evolution and Reengineering*, SANER '18, pages 61–71.

Jiang, Z. M., Hassan, A. E., Hamann, G., and Flora, P. (2008). Automatic identification of load testing problems. In *Proceedings of the 2008 IEEE International Conference on Software Maintenance*, ICSM '08, pages 307–316.

Kabacoff, R. (2011). *R in Action*. Manning Publications Co.

Kabinna, S., Bezemer, C.-P., Shang, W., and Hassan, A. E. (2016a). Logging library migrations: A case study for the Apache Software Foundation projects. In *Proceedings of the 13th International Conference on Mining Software Repositories*, MSR '16, pages 154–164.

Kabinna, S., Bezemer, C.-P., Shang, W., Syer, M. D., and Hassan, A. E. (2018). Examining the stability of logging statements. *Empirical Software Engineering*, 23(1):290–333.

Kabinna, S., Shang, W., Bezemer, C. P., and Hassan, A. E. (2016b). Examining the stability of logging statements. In *Proceedings of the 23rd IEEE International Conference on Software Analysis, Evolution, and Reengineering*, SANER '16, pages 326–337.

Kamei, Y., Fukushima, T., McIntosh, S., Yamashita, K., Ubayashi, N., and Hassan, A. E. (2016). Studying just-in-time defect prediction using cross-project models. *Empirical Software Engineering*, 21(5):2072–2106.

Kamei, Y., Shihab, E., Adams, B., Hassan, A. E., Mockus, A., Sinha, A., and Ubayashi, N. (2013). A large-scale empirical study of just-in-time quality assurance. *IEEE Transactions on Software Engineering*, 39(6):757–773.

Kavulya, S., Tan, J., Gandhi, R., and Narasimhan, P. (2010). An analysis of traces from a production mapreduce cluster. In *Proceedings of the 10th IEEE/ACM International Conference on Cluster, Cloud and Grid Computing*, CCGRID '10, pages 94–103.

King, J., Pandita, R., and Williams, L. (2015). Enabling forensics by proposing heuristics to identify mandatory log events. In *Proceedings of the 2015 Symposium and Bootcamp on the Science of Security*, HotSoS '15, pages 6:1–6:11.

King, J., Stallings, J., Riaz, M., and Williams, L. (2017). To log, or not to log: using heuristics to identify mandatory log events – a controlled experiment. *Empirical Software Engineering*, 22(5):2684–2717.

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

Kuhn, M. and Johnson, K. (2013). *Applied predictive modeling*. Springer.

Lal, S. and Sureka, A. (2016). Logopt: Static feature extraction from source code for automated catch block logging prediction. In *Proceedings of the 9th India Software Engineering Conference*, ISEC '16, pages 151–155.

Lawless, J. and Singhal, K. (1978). Efficient screening of nonnormal regression models. *Biometrics*, 34(2):318–327.

Levenshtein, V. I. (1966). Binary codes capable of correcting deletions, insertions, and reversals. *Soviet physics doklady*, 10(8):707–710.

Li, H., Chen, T.-H. P., Shang, W., and Hassan, A. E. (2018). Studying software logging using topic models. *Empirical Software Engineering*.

Li, H., Shang, W., and Hassan, A. E. (2017a). Which log level should developers choose for a new logging statement? *Empirical Software Engineering*, 22(4):1684–1716.

Li, H., Shang, W., Zou, Y., and E. Hassan, A. (2017b). Towards just-in-time suggestions for log changes. *Empirical Software Engineering*, 22(4):1831–1865.

Liaw, A. and Wiener, M. (2002). Classification and regression by randomforest. *R news*, 2(3):18–22.

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*, ICMLA '12, pages 813–818.

Liu, Y., Poshyvanyk, D., Ferenc, R., Gyimothy, T., and Chrisochoides, N. (2009a). Modeling class cohesion as mixtures of latent topics. In *Proceedings of the 25th International Conference on Software Maintenance*, ICSE '09, pages 233 –242.

Liu, Y., Poshyvanyk, D., Ferenc, R., Gyimothy, T., and Chrisochoides, N. (2009b). Modeling class cohesion as mixtures of latent topics. In *Proceedings of the 25th IEEE International Conference on Software Maintenance*, ICSM '09, pages 233–242.

Macbeth, G., Razumiejczyk, E., and Ledesma, R. D. (2011). Cliff's delta calculator: A non-parametric effect size program for two groups of observations. *Universitas Psychologica*, 10(2):545–555.

Mant, J., Doust, J., Roalfe, A., Barton, P., Cowie, M. R., Glasziou, P., Mant, D., McManus, R., Holder, R., Deeks, J., et al. (2009). Systematic review and individual patient data meta-analysis of diagnosis of heart failure, with modelling of implications of different diagnostic strategies in primary care. *Health Technol Assess*, 13(32):1–207.

Mantel, N. (1970). Why stepdown procedures in variable selection. *Technometrics*, 12(3):621–625.

Mariani, L. and Pastore, F. (2008). Automated identification of failure causes in system logs. In *Proceedings of the 19th International Symposium on Software Reliability Engineering*, ISSRE '08, pages 117–126.

Martin, T. M., Harten, P., Young, D. M., Muratov, E. N., Golbraikh, A., Zhu, H., and Tropsha, A. (2012). Does rational selection of training and test sets improve the outcome of qsar modeling? *Journal of chemical information and modeling*, 52(10):2570–2578.

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*, pages 113–120.

McCabe, T. J. (1976). A complexity measure. *IEEE Transactions on software Engineering*, (4):308–320.

McCallum, A. K. (2002). Mallet: A machine learning for language toolkit.

McCullagh, P. (1980). Regression models for ordinal data. *Journal of the Royal Statistical Society. Series B (Methodological)*, 42(2):109–142.

McIntosh, S., Kamei, Y., Adams, B., and Hassan, A. E. (2014). The impact of code review coverage and code review participation on software quality: A case study of the qt, vtk, and itk projects. In *Proceedings of the 11th Working Conference on Mining Software Repositories*, MSR '14, pages 192–201.

McIntosh, S., Kamei, Y., Adams, B., and Hassan, A. E. (2016). An empirical study of the impact of modern code review practices on software quality. *Empirical Software Engineering*, 21(5):2146–2189.

McKelvey, R. D. and Zavoina, W. (1975). A statistical model for the analysis of ordinal level dependent variables. *Journal of Mathematical Sociology*, 4(1):103–120.

Microsoft-MSDN (2016). Logging an exception. https://msdn.microsoft.com/en-us/library/ff664711(v=pandp.50).aspx. Accessed 25 July 2018.

Misra, H., Cappé, O., and Yvon, F. (2008). Using lda to detect semantically incoherent documents. In *Proceedings of the 12th Conference on Computational Natural Language Learning*, CoNLL '08, pages 41–48.

Nagappan, N. and Ball, T. (2007). Using software dependencies and churn metrics to predict field failures: An empirical case study. In *Proceedings of the First International Symposium on Empirical Software Engineering and Measurement*, ESEM '07, pages 364–373.

Nagappan, N., Ball, T., and Zeller, A. (2006). Mining metrics to predict component failures. In *Proceedings of the 28th International Conference on Software Engineering*, ICSE '06, pages 452–461.

Nagaraj, K., Killian, C., and Neville, J. (2012). Structured comparative analysis of systems logs to diagnose performance problems. In *Proceedings of the 9th USENIX Conference on Networked Systems Design and Implementation*, NSDI'12, pages 26–26, Berkeley, CA, USA. USENIX Association.

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*, ICSE '11, pages 932–935.

Oliner, A., Ganapathi, A., and Xu, W. (2012). Advances and challenges in log analysis. *Communications of the ACM*, 55(2):55–61.

Panichella, A., Dit, B., Oliveto, R., Di Penta, M., Poshyvanyk, D., and De Lucia, A. (2013). How to effectively use topic models for software engineering tasks? an approach based on genetic algorithms. In *Proceedings of the 2013 International Conference on Software Engineering*, ICSE '13, pages 522–531.

Panichella, A., Dit, B., Oliveto, R., Di Penta, M., Poshyvanyk, D., and De Lucia, A. (2016). Parameterizing and assembling ir-based solutions for se tasks using genetic algorithms. In *Proceedings of the 23rd IEEE International Conference on Software Analysis, Evolution, and Reengineering*, SANER '16, pages 314–325.

Pecchia, A., Cinque, M., Carrozza, G., and Cotroneo, D. (2015). Industry practices and event logging: Assessment of a critical software development process. In *Proceedings of the 37th International Conference on Software Engineering*, ICSE '15, pages 169–178.

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 Trans. on Software Engineering*, pages 420–432.

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*, MSR '11, pages 43–52.

Romano, J., Kromrey, J. D., Coraggio, J., and Skowronek, J. (2006). Appropriate statistics for ordinal level data: Should we really be using t-test and cohensd for evaluating group differences on the nsse and other surveys. In *annual meeting of the Florida Association of Institutional Research*, pages 1–33.

Rugg, G. and McGeorge, P. (2005). The sorting techniques: a tutorial paper on card sorts, picture sorts and item sorts. *Expert Systems*, 22(3):94–107.

Sarbanes, P. (2002). Sarbanes-Oxley Act of 2002. In *The Public Company Accounting Reform and Investor Protection Act*.

Schroter, A., Schrter, A., Bettenburg, N., and Premraj, R. (2010). Do stack traces help developers fix bugs? In *Proceedings of the 7th IEEE Working Conference on Mining Software Repositories*, MSR '10, pages 118–121.

Scott, A. and Knott, M. (1974). A cluster analysis method for grouping means in the analysis of variance. *Biometrics*, 30(3):507–512.

Shang, W., Jiang, Z. M., Adams, B., Hassan, A. E., Godfrey, M. W., Nasser, M., and Flora, P. (2011). An exploratory study of the evolution of communicated information about the execution of large software systems. In *2011 18th Working Conference on Reverse Engineering*, WCRE '11, pages 335–344.

Shang, W., Jiang, Z. M., Adams, B., Hassan, A. E., Godfrey, M. W., Nasser, M., and Flora, P. (2014a). An exploratory study of the evolution of communicated information about the execution of large software systems. *Journal of Software: Evolution and Process*, 26(1):3–26.

Shang, W., Jiang, Z. M., Hemmati, H., Adams, B., Hassan, A. E., and Martin, P. (2013). Assisting developers of big data analytics applications when deploying on hadoop clouds. In *Proceedings of the 2013 International Conference on Software Engineering*, ICSE '13, pages 402–411.

Shang, W., Nagappan, M., and Hassan, A. E. (2015). Studying the relationship between logging characteristics and the code quality of platform software. *Empirical Software Engineering*, 20(1):1–27.

Shang, W., Nagappan, M., Hassan, A. E., and Jiang, Z. M. (2014b). Understanding log lines using development knowledge. In *Proceedings of the 30th IEEE International Conference on Software Maintenance and Evolution*, ICSME '14, pages 21–30.

Sharma, B., Chudnovsky, V., Hellerstein, J. L., Rifaat, R., and Das, C. R. (2011). Modeling and synthesizing task placement constraints in google compute clusters. In *Proceedings of the 2nd ACM Symposium on Cloud Computing*, SOCC '11, pages 3:1–3:14.

Shihab, E., Jiang, Z. M., Ibrahim, W. M., Adams, B., and Hassan, A. E. (2010). Understanding the impact of code and process metrics on post-release defects: A case study on the Eclipse project. In *Proceedings of the 4th ACM-IEEE International Symposium on Empirical Software Engineering and Measurement*, ESEM '10, pages 4:1–4:10.

Simon, N., Friedman, J., Hastie, T., and Tibshirani, R. (2011). Regularization paths for cox's proportional hazards model via coordinate descent. *Journal of Statistical Software*, 39(5):1–13.

Sommer, S. and Huggins, R. M. (1996). Variables selection using the Wald test and a robust CP. *Applied statistics*, 45(1):15–29.

Spencer, D. (2009). *Card sorting: Designing usable categories*. Rosenfeld Media.

Steyvers, M. and Griffiths, T. (2007). Probabilistic topic models. *Handbook of latent semantic analysis*, 427(7):424–440.

Sun, X., Li, B., Leung, H., Li, B., and Li, Y. (2015a). Msr4sm: Using topic models to effectively mining software repositories for software maintenance tasks. *Information and Software Technology*, 66:1–12.

Sun, X., Li, B., Li, Y., and Chen, Y. (2015b). What information in software historical repositories do we need to support software maintenance tasks? an approach based on topic model. In *Computer and Information Science*, pages 27–37. Springer International Publishing, Cham.

Sun, X., Liu, X., Li, B., Duan, Y., Yang, H., and Hu, J. (2016). Exploring topic models in software engineering data analysis: A survey. In *Proceedings of the 17th IEEE/ACIS International Conference on Software Engineering, Artificial Intelligence, Networking and Parallel/Distributed Computing*, SNPD' 16, pages 357–362.

Swinscow, T. D. V., Campbell, M. J., et al. (2002). *Statistics at Square One*. BMJ, London.

Syer, M. D., Jiang, Z. M., Nagappan, M., Hassan, A. E., Nasser, M., and Flora, P. (2013). Leveraging performance counters and execution logs to diagnose memory-related performance issues. In *Proceedings of the 29th IEEE International Conference on Software Maintenance*, ICSM '13, pages 110–119.

Tantithamthavorn, C., McIntosh, S., Hassan, A. E., and Matsumoto, K. (2017). An empirical comparison of model validation techniques for defect prediction models. *IEEE Transactions on Software Engineering*, 43(1):1–18.

Thomas, S., Adams, B., Hassan, A. E., 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*, SCAM '10, pages 55–64.

Thomas, S. W. (2012). A lightweight source code preprocesser. https://github.com/doofuslarge/lscp. Accessed 25 July 2018.

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*, pages 173–182.

Thomas, S. W., Adams, B., Hassan, A. E., and Blostein, D. (2014). Studying software evolution using topic models. *Science of Computer Programming*, 80:457–479.

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*, MSR '09, pages 163–166.

Tibshirani, R. (1996). Regression shrinkage and selection via the lasso. *Journal of the Royal Statistical Society. Series B (Methodological)*, pages 267–288.

Tourani, P. and Adams, B. (2016). The impact of human discussions on just-in-time quality assurance: An empirical study on openstack and eclipse. In *Proceedings of the 23rd International Conference on Software Analysis, Evolution, and Reengineering*, SANER '16, pages 189–200.

Wallach, H. M., Mimno, D. M., and McCallum, A. (2009). Rethinking lda: Why priors matter. In *Advances in neural information processing systems*, NIPS '09, pages 1973–1981.

Wilks, D. S. (2011). *Statistical methods in the atmospheric sciences*, volume 100. Academic press.

Witten, I. H. and Frank, E. (2005). *Data Mining: Practical machine learning tools and techniques*. Morgan Kaufmann.

Xu, W., Huang, L., Fox, A., Patterson, D., and Jordan, M. (2008). Mining console logs for large-scale system problem detection. In *Proceedings of the Third Conference on Tackling Computer Systems Problems with Machine Learning Techniques*, SysML'08, pages 4–4.

Xu, W., Huang, L., Fox, A., Patterson, D., and Jordan, M. (2009a). Online system problem detection by mining patterns of console logs. In *Proceedings of the 2009 Ninth IEEE International Conference on Data Mining*, ICDM '09, pages 588–597.

Xu, W., Huang, L., Fox, A., Patterson, D., and Jordan, M. I. (2009b). Detecting large-scale system problems by mining console logs. In *Proceedings of the ACM SIGOPS 22Nd Symposium on Operating Systems Principles*, SOSP '09, pages 117–132.

Yuan, D., Luo, Y., Zhuang, X., Rodrigues, G. R., Zhao, X., Zhang, Y., Jain, P. U., and Stumm, M. (2014). Simple testing can prevent most critical failures: An analysis of production failures in distributed data-intensive systems. In *Proceedings of the 11th USENIX Conference on Operating Systems Design and Implementation*, OSDI'14, pages 249–265.

Yuan, D., Mai, H., Xiong, W., Tan, L., Zhou, Y., and Pasupathy, S. (2010). Sherlog: Error diagnosis by connecting clues from run-time logs. In *Proceedings of the 15th International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '10, pages 143–154.

Yuan, D., Park, S., Huang, P., Liu, Y., Lee, M. M., Tang, X., Zhou, Y., and Savage, S. (2012a). Be conservative: Enhancing failure diagnosis with proactive logging. In *Proceedings of the 10th USENIX Conference on Operating Systems Design and Implementation*, OSDI'12, pages 293–306.

Yuan, D., Park, S., and Zhou, Y. (2012b). Characterizing logging practices in open-source software. In *Proceedings of the 34th International Conference on Software Engineering*, ICSE '12, pages 102–112.

Yuan, D., Zheng, J., Park, S., Zhou, Y., and Savage, S. (2011). Improving software diagnosability via log enhancement. In *Proceedings of the 16th International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '11, pages 3–14.

Zeng, L., Xiao, Y., and Chen, H. (2015). Linux auditing: Overhead and adaptation. In *Proceedings of 2015 IEEE International Conference on Communications*, ICC '15, pages 7168–7173.

Zhang, C., Guo, Z., Wu, M., Lu, L., Fan, Y., Zhao, J., and Zhang, Z. (2011). Autolog: Facing log redundancy and insufficiency. In *Proceedings of the Second Asia-Pacific Workshop on Systems*, APSys '11, pages 10:1–10:5.

Zhang, S., Cohen, I., Symons, J., and Fox, A. (2005). Ensembles of models for automated diagnosis of system performance problems. In *Proceedings of the 2005 International Conference on Dependable Systems and Networks*, DSN '05, pages 644–653.

Zhao, X., Rodrigues, K., Luo, Y., Stumm, M., Yuan, D., and Zhou, Y. (2017a). The game of twenty questions: Do you know where to log? In *Proceedings of the 16th Workshop on Hot Topics in Operating Systems*, HotOS '17, pages 125–131.

Zhao, X., Rodrigues, K., Luo, Y., Stumm, M., Yuan, D., and Zhou, Y. (2017b). Log20: Fully automated optimal placement of log printing statements under specified overhead threshold. In *Proceedings of the 26th Symposium on Operating Systems Principles*, SOSP '17, pages 565–581.

Zhu, J., He, P., Fu, Q., Zhang, H., Lyu, M. R., and Zhang, D. (2015). Learning to log: Helping developers make informed logging decisions. In *Proceedings of the 37th International Conference on Software Engineering - Volume 1*, ICSE '15, pages 415–425.

Zimmermann, T. (2016). Card-sorting: From text to themes. In Tim Menzies, L. W. and Zimmermann, T., editors, *Perspectives on Data Science for Software Engineering*, pages 137–141. Morgan Kaufmann, Burlington, Massachusetts.

Zimmermann, T., Weisgerber, P., Diehl, S., and Zeller, A. (2004). Mining version histories to guide software changes. In *Proceedings of the 26th International Conference on Software Engineering*, ICSE '04, pages 563–572.