Noname manuscript No.
(will be inserted by the editor)

Which Log Level Should Developers Choose for a New Logging Statement?

Heng Li · Weiyi Shang · Ahmed E. Hassan

Received: date / Accepted: date

Abstract Logging statements are used to record valuable runtime information about applications. Each logging statement is assigned a log level such that users can disable some verbose log messages while allowing the printing of other important ones. However, prior research finds that developers often have difficulties when determining the appropriate level for their logging statements. In this paper, 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 (Hadoop, Directory Server, Hama, and Qpid), 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.

Heng Li, Ahmed E. Hassan
Software Analysis and Intelligence Lab (SAIL)
Queen's University
Kingston, Ontario, Canada
E-mail: {hengli, ahmed}@cs.queensu.ca

Weiyi Shang
Department of Computer Science and Software Engineering
Concordia University
Montreal, Quebec, Canada
E-mail: shang@encs.concordia.ca

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 (Fu *et al.*, 2014; Gülcü and Stark, 2003; Yuan *et al.*, 2012b).

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). 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.*, 2012a,c). 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). Too many logs may contain noisy information that becomes a burden for developers during failure diagnosis (Yuan *et al.*, 2014).

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, 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.

Prior research finds that developers often have difficulties when estimating the cost and benefit of each log level, and spend much effort on adjusting the levels of logs (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

[1] http://logging.apache.org/log4j/2.x
[2] http://commons.apache.org/proper/commons-logging
[3] http://www.slf4j.org
[4] https://issues.apache.org/jira/browse/HADOOP-10274
[5] https://issues.apache.org/jira/browse/HADOOP-10015

patch in Listing 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 Listing 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 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.

Listing 1 Patch for JIRA issue HADOOP-10274 (svn commit number: 1561934).

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

Listing 2 Patch for JIRA issue HADOOP-10015 (svn commit number: 1580977).

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

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 paper, 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 paper, 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.

Paper organization. The remainder of the paper is organized as follows. Section 2 describes the studied software projects and our experimental setup. Section 3 performs an empirical study on the log level distribution in the studied projects. Section 4 explains the approaches that we used to answer the research questions and presents the results of our case study. Section 5 discusses the topics about cross-project evaluation and the log level changes. Section 6 discusses threats to the validity of our findings. Section 7 surveys recent work on software logs that has been done in the recent years. Finally, section 8 draws conclusions.

2 Case Study Setup

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

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) 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 1 presents the size of these projects in terms of source lines of code (SLOC), 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.

2.2 Data Extraction

Figure 1 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

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

Table 1 Overview of the studied projects.

Project	**SLOC**	**Studied develop. 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

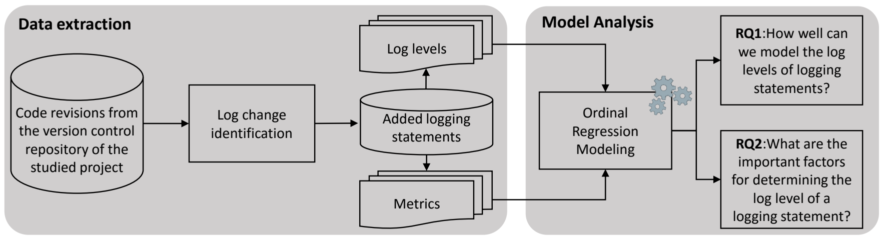
Data extraction
Code revisions from the version control repository of the studied project
Log change identification
Log levels
Added logging statements
Metrics
Model Analysis
Ordinal Regression Modeling
RQ1:How well can we model the log levels of logging statements?
RQ2:What are the important factors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

Fig. 1 An overview of our data extraction and analysis approaches.

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.

3 Preliminary Study

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

3.1 Log level distribution in the studied projects

No single log level dominates all other log levels. As shown in Figure 2, 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

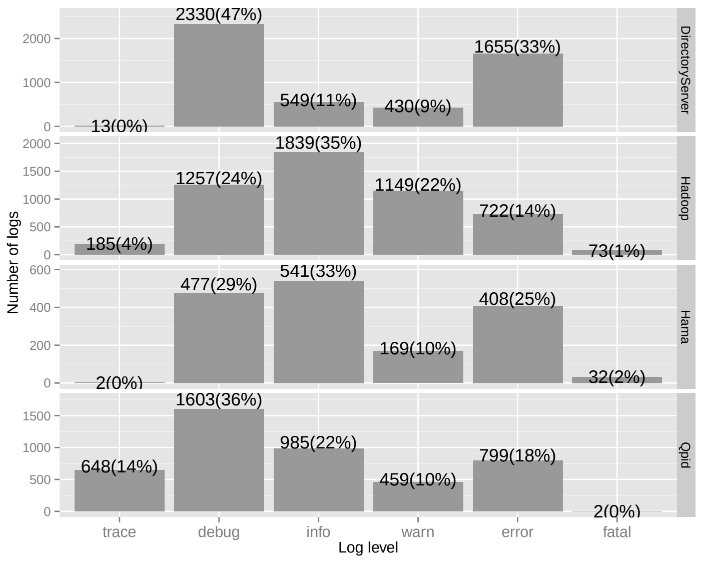

Fig. 2 Log level distribution in the added logging statements.

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 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.

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

3.2 Log level distribution in different blocks

Logging statements have different distribution of log levels across the different containing blocks. In this paper, 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 3 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 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

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

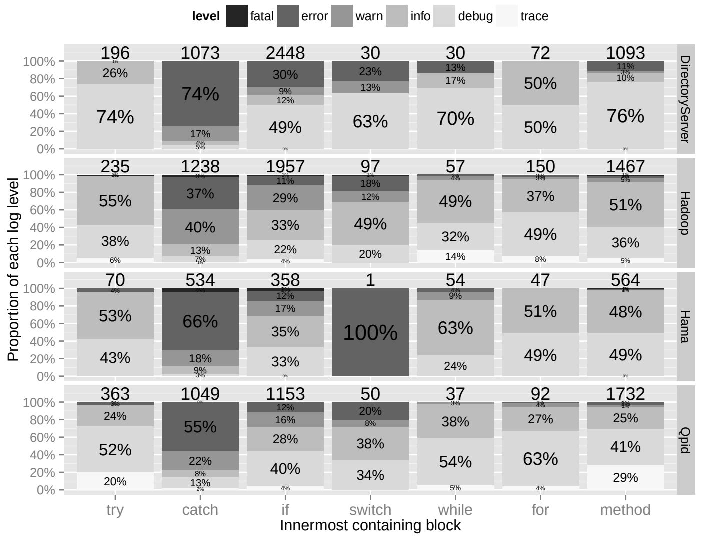

Fig. 3 Log level distribution in the added logging statements in different types of blocks.

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 record some expected runtime events, such as startup or shutdown, thus they usually use more verbose log levels such as *info* or *debug*. Figure 3 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.

3.3 Log level distribution in catch blocks

A common best practice for exception-handling is to log the information associated with the exception (MSDN, 2011). 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 4 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" (Eberhardt, 2014; Zhu *et al.*, 2015). For most types of exceptions (10 out of 12 as shown in Figure 4), 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.

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.

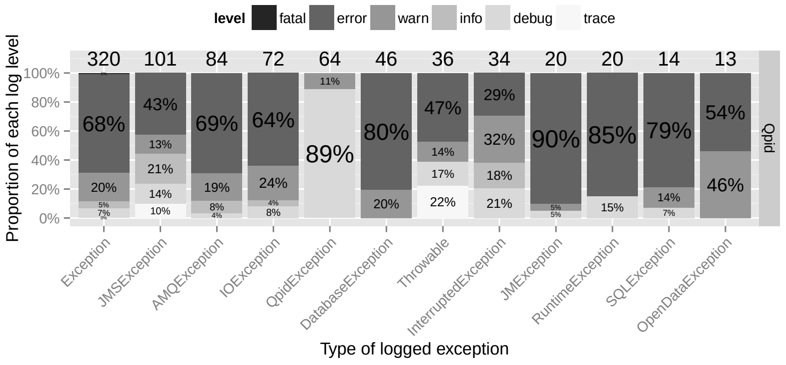

Fig. 4 Log level distribution in the added logging statements in different types of exception-catching blocks (Qpid).

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 2 presents a list of all the metrics that we collected along the five dimensions. Table 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 paper 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 3) (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*, *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

Table 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 (D'Ambros *et al.*, 2012; Shihab *et al.*, 2010), 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 (D'Ambros *et al.*, 2012; Shihab *et al.*, 2010), 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 (D'Ambros *et al.*, 2012; Graves *et al.*, 2000; Hassan, 2009). 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 (D'Ambros *et al.*, 2012; Graves *et al.*, 2000; Hassan, 2009). 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.

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 5 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 5, 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 (McCullagh, 1980; McKelvey and Zavoina, 1975), 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). 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.

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

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

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

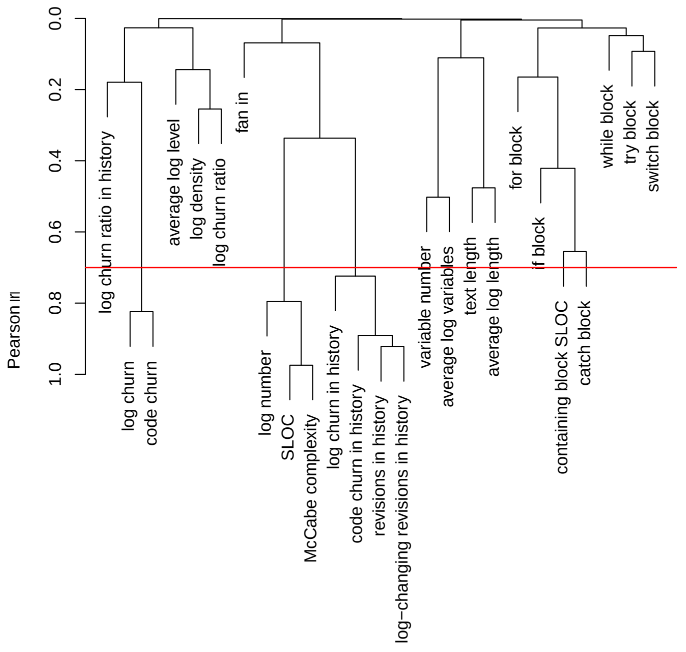

Fig. 5 Correlation analysis using Spearman hierarchical clustering (for Directory Server). The red line indicates the threshold (0.7) that is used to remove collinear metrics.

– $P[level = error] = P[level \geq error] - P[level \geq fatal]$
– $P[level = fatal] = P[level \geq fatal]$

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 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

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

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 paper, 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 2 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.*, 2015). 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.*, 2015):

– 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 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 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 3, for the *Hadoop* project, the most-frequently used log level in each type of block only covers 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

Table 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	AUC	Brier Score	AUC	Brier Score	AUC
D. Server	0.44 (0.43)[1]	0.78(0.78)[2]	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.

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.

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.*, 2015; 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 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 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.

Table 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 $p \geq 0.05$; * $p < 0.05$; ** $p < 0.01$; *** $p < 0.001$

Table 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 $p \geq 0.05$; * $p < 0.05$; ** $p < 0.01$; *** $p < 0.001$

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 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 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 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 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 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 contribute 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 2). 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 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.

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 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 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.

Table 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

Comparing Table 6 and Table 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 2. 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 1, there are 491 added logging statements that experience a later log level change in the four studied projects. Table 7 summarizes the patterns 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*.

Table 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.

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 paper we study the log levels of logging statements generally; however, future study should explore the confusion and distinction between adjacent log levels.

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 paper 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 paper 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 paper 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 paper 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.

7 Related Work

In this section, we discuss the prior research with regards to leveraging logs, improving logs, and empirical studies of logs.

Leveraging logs. A large amount of log-related research work focuses on postmortem diagnosis of logs (Mariani and Pastore, 2008; Mariani *et al.*, 2009; Oliner *et al.*, 2012; Xu *et al.*, 2009; Yuan *et al.*, 2010). Since console logs that are generated in large-scale data centers often consist of voluminous messages and it is difficult for operators to detect noteworthy logs, Xu *et al.* (2009) propose a method to mine the rich source of log information in order to automatically detect system runtime problems within a short time. As field logs and source code are usually the only resources for developers to diagnose a production failure, Yuan *et al.* (2010) propose a tool named SherLog, which leverages run-time log information and source code to infer the execution path during a failed production run. The tool needs neither reproducing the failure nor expert knowledge of the product. Mariani and Pastore (2008) propose a technique, which automatically analyzes log files and retrieves valuable information, to assist developers in identifying failure causes. The wide usage of logs highlights the importance of proper logging practices, and motivates our work to provide automated suggestions for the appropriate log level of a newly-added logging statement.

Improving logs. Several prior research efforts focus on improving logging practices. In order to address the issue of lacking of log messages for failure diagnosis, *Errlog* (Yuan *et al.*, 2012a) analyzes the source code to detect unlogged exceptions (abnormal or unusual conditions) and automatically insert the missing logging statements. *LogEnhancer*(Yuan *et al.*, 2012c), on the other hand, automatically adds *causally-related* information on existing logging statements to aid in future failure diagnosis. A recent tool named *LogAdvisor* (Zhu *et al.*, 2015) aims to provide developers with suggestions on where to log. *LogAdvisor* extracts contextual features (such as textual features) of a code snippet (exception snippet or return-value-check snippet) and leverages the features to suggest whether a logging statement should be added to a code snippet. These tools try to improve logs by adding additional logged information or suggesting where to log. In this paper, we attempt to support logging practices by suggesting the appropriate log level of newly-added logging statements.

Empirical studies of logs. Researchers have conducted several empirical studies on logging practices. Fu *et al.* (2014) study the logging practices in two industrial software projects. They investigate what kinds of code snippet contain logs in order to provide a guideline on where to log. A recent empirical study (Kabinna *et al.*, 2016b) highlights that developers should carefully estimate the benefits and effort for migrating from an old logging library to a new logging library. Another recent work (Kabinna *et al.*, 2016a) studies how likely a logging statement is going to be changed after the initial commit. In contrast, we aim to provide developers on the choice of log level when they initially add a logging statement. A recent study on industry logging practices (Pecchia *et al.*, 2015) highlights that the logging behavior is strongly developer dependent, and there is a need to establish a standard logging policy in a company. Such prior findings motivate our study towards improving current logging practices. Prior research shows that logs are often changed by developers without considering the needs of other stakeholders (Shang *et al.*, 2011, 2014), and the changes to logs often break the functionality of log processing applications that are highly depending on the format of logs. Similarly, Yuan *et al.* (2012b) studies the logging practices in four open source projects. They study the modification of logging statements over time. They examine the amount of effort that developers spend on modifying logging statements. Yuan et al. build a simple log-level checker to detect problematic log levels. Their checker is based on the assumption that if two logging statements within similar code snippets have inconsistent log levels, then at least one of them is likely to be incorrect. Our study differs from their work in two dimensions: 1) our model predicts actual log levels while the log-level checker only detects inconsistent levels; 2) the checker is based on identifying similar source code but our model considers a large number of software metrics that are relevant to log levels.

8 Conclusions

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 firstly 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.

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.

References

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

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

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.

Eberhardt, C. (2014). The art of logging. `http://www.codeproject.com/Articles/42354/The-Art-of-Logging`. Accessed 12 May 2016.

Efron, B. (1979). Bootstrap methods: another look at the jackknife. *The annals of Statistics*, pages 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.

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, New York, NY, USA. ACM.

Graves, T. L., Karr, A. F., Marron, J. S., and Siy, H. (2000). Predicting fault incidence using software change history. *IEEE Trans. Softw. Eng.*, **26**(7), 653–661.

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

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. (2009). Predicting faults using the complexity of code changes. In *Proceedings of the 31st International Conference on Software Engineering*, ICSE '09, pages 78–88, Washington, DC, USA. IEEE Computer Society.

Kabinna, S., Bezemer, C.-P., Hassan, A. E., and Shang, W. (2016a). Examining the stability of logging statements. In *Proceedings of the 23rd IEEE International Conference on Software Analysis, Evolution, and Reengineering*, SANER '16.

Kabinna, S., Bezemer, C.-P., Shang, W., and Hassan, A. E. (2016b). 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.

Kuhn, M. and Johnson, K. (2013). *Applied predictive modeling*. Springer.

Lawless, J. and Singhal, K. (1978). Efficient screening of nonnormal regression models. *Biometrics*, pages 318–327.

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.

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 2008 19th International Symposium on Software Reliability Engineering*, ISSRE '08, pages 117–126, Washington, DC, USA. IEEE Computer Society.

Mariani, L., Pastore, F., and Pezze, M. (2009). A toolset for automated failure analysis. In *Proceedings of the 31st International Conference on Software Engineering*, ICSE '09, pages 563–566, Washington, DC, USA. IEEE Computer Society.

McCullagh, P. (1980). Regression models for ordinal data. *Journal of the royal statistical society. Series B (Methodological)*, pages 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 2014, pages 192–201, New York, NY, USA. ACM.

McIntosh, S., Kamei, Y., Adams, B., and Hassan, A. E. (2015). An empirical study of the impact of modern code review practices on software quality. *Empirical Software Engineering*, pages 1–44.

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.

MSDN (2011). Logging an exception. `https://msdn.microsoft.com/en-us/library/ff664711(v=pandp.50).aspx`. Accessed 12 May 2016.

Oliner, A., Ganapathi, A., and Xu, W. (2012). Advances and challenges in log analysis. *Communications of the ACM*, **55**(2), 55–61.

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 - Volume 2*, ICSE '15, pages 169–178, Piscataway, NJ, USA.

IEEE Press.

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 *Proceedings of the 18th Working Conference on Reverse Engineering*, WCRE '11, pages 335–344, Washington, DC, USA. IEEE Computer Society.

Shang, W., Jiang, Z. M., Adams, B., Hassan, A. E., Godfrey, M. W., Nasser, M., and Flora, P. (2014). 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.

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 2010 ACM-IEEE International Symposium on Empirical Software Engineering and Measurement*, ESEM '10, pages 4:1–4:10, New York, NY, USA. ACM.

Sommer, S. and Huggins, R. M. (1996). Variables selection using the Wald test and a robust CP. *Applied statistics*, pages 15–29.

Wilks, D. S. (2011). *Statistical methods in the atmospheric sciences*, volume 100. Academic press.

Xu, W., Huang, L., Fox, A., Patterson, D., and Jordan, M. I. (2009). 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. ACM.

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 *ACM SIGARCH Computer Architecture News*, volume 38, pages 143–154. ACM.

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*, volume 12 of *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. IEEE Press.

Yuan, D., Zheng, J., Park, S., Zhou, Y., and Savage, S. (2012c). Improving software diagnosability via log enhancement. *ACM Transactions on Computer Systems*, **30**(1), 4.

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, Berkeley, CA, USA. USENIX Association.

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, Piscataway, NJ, USA. IEEE Press.

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, Washington, DC, USA. IEEE Computer Society.