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

Towards Just-in-time Suggestions for Log Changes

Heng Li · Weiyi Shang · Ying Zou ·
Ahmed E. Hassan

Received: date / Accepted: date

Abstract Software developers typically insert logging statements in their source code to record runtime information. However, providing proper logging statements remains a challenging task. Prior approaches automatically enhance logging statements, as a post-implementation process. Such automatic approaches do not take into account developers' domain knowledge; nevertheless, developers usually need to carefully design the logging statements since logs are a rich source about the field operation of a software system. The goals of this paper include: i) understanding the reasons for log changes; and ii) proposing an approach that can provide developers with log change suggestions as soon as they commit a code change, which we refer to as "just-in-time" suggestions for log changes. In particular, we derive a set of measures based on manually examining the reasons for log changes and our experiences. We use these measures as explanatory variables in random forest classifiers to model whether a code commit requires log changes. These classifiers can provide just-in-time suggestions for log changes. We perform a case study on four open source projects: Hadoop, Directory Server, Commons HttpClient, and

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

Ying Zou
Department of Electrical and Computer Engineering
Queens University
Kingston, ON, Canada
E-mail: ying.zou@queensu.ca

Qpid. We find that: (i) The reasons for log changes can be grouped along four categories: block change, log improvement, dependence-driven change, and logging issue; (ii) our random forest classifiers can effectively suggest whether a log change is needed: the classifiers that are trained from within-project data achieve a balanced accuracy of 0.76 to 0.82, and the classifiers that are trained from cross-project data achieve a balanced accuracy of 0.76 to 0.80; (iii) the characteristics of code changes in a particular commit and the current snapshot of the source code are the most influential factors for determining the likelihood of a log change in a commit.

1 Introduction

Logs are generated at runtime from logging statements in the source code. Logs record valuable run-time information. A logging statement, as shown below, typically specifies a verbosity level (e.g., debug/info/warn/error/fatal), a static text and one or more variables [10, 14, 42].

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

Logs help software practitioners better understand the behaviors of large scale software systems and assist in improving the quality of the systems [9, 33]. Software operators leverage the rich information in logs to guide capacity planning efforts [19, 34], to monitor system health [2], and to identify abnormal behaviors [8, 35, 38]. Besides, software developers rely on logs for debugging field failures [13, 39, 41]. In recent years, the broad usage of logs led to the emergence a new market of *Log Processing Applications* (LPAs) (e.g., Splunk[1] [2], XpoLog[2], and Logstash[3]), which support the collection, storage, search, and analysis of large amounts of log data.

However, appropriate logging is difficult to reach in practice. Both logging too little and logging too much is undesirable [10]. Logging too little may result in the lack of runtime information that is crucial for understanding and diagnosing software systems [40, 41]; while logging too much may lead to runtime overhead and costly log maintenance efforts [10]. A recent study shows that developers spend a large amount of effort for maintaining logging statements, and 33% of the log changes are introduced as after-thoughts (i.e., as follow-up changes instead of being done when the actual surrounding code is being changed) [42].

Listing 1 shows a code snippet that is taken from Hadoop revision 1240413. We use "svn blame"[4] to show the contributing commit of each code line. The code snippet shows that the commit 1190122 added a *try-catch* statement, and an *if* statement in the *catch* clause. In a later commit 1240413, which contributed to a bug fix, developer added an error logging statement to

[1] Splunk. http://www.splunk.com/
[2] XpoLog. http://www.xpolog.com/
[3] Logstash. http://logstash.net/
[4] svn blame. http://svnbook.red-bean.com/en/1.7/svn.ref.svn.c.blame.html

record important runtime information, in order to help fix bug MAPREDUCE-3711[5]. The information that is recorded by the logging statement even help understand the execution of the system when fixing bugs in the future (e.g., MAPREDUCE-5501[6] and MAPREDUCE-6317[7]). Suppose that the developer was suggested to add the logging statement in the earlier commit (1190122) with the try-catch statement, the logged information would help developers and operators understand the system's behavior when they are fixing bug MAPREDUCE-3711. In this paper, we propose an approach that can automatically provide developers with such suggestions for log changes when they commit new code changes.

Listing 1 An "svn blame" example showing that a later commit (1240413) added a logging statement that was missing in an earlier commit (1190122)

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

Log enhancement approaches, such as *Errlog* [41] and *LogEnhancer* [40], aim to improve software failure diagnosis by automatically adding more logged information to the existing code, as a post-implementation process. However, these automatic log enhancement approaches never take into account developers' knowledge about where and how to add logging statements. In practice, developers need to carefully design logging statements since logs contain valuable information for both software developers and operators [42].

Recent studies investigate where developers insert logging statements [10] and automatically suggest locations in need of logging statements [44]. In particular, the authors conducted source code analysis to investigate the types of code snippets (e.g., catch block) in which developers often insert logging statements. The study provides post-coding guidelines for inserting logging

[5] https://issues.apache.org/jira/browse/MAPREDUCE-3711
[6] https://issues.apache.org/jira/browse/MAPREDUCE-5501
[7] https://issues.apache.org/jira/browse/MAPREDUCE-6317

statements into the source code. However to the best of our knowledge, there exists no studies to guide developers during coding, i.e., providing guidance about whether to change (add, delete or modify) logging statements when developers are committing code changes.

In this paper, we propose an approach that can provide just-in-time suggestions as to whether a log change is needed when a code change occurs. The term "just-in-time" is based on prior research by Kamei et al. [17] that advocates the benefits of providing suggestions to developers at commit time. Follow-up studies [11, 18, 37] also use the term "just-in-time" to describe commit-time suggestions or alerts. In this paper, we leverage prior commits to build classifiers in order to suggest whether log changes are needed for a new commit. We perform a case study on four open source systems (*Hadoop*, *Directory Server*, *Commons HttpClient*, and *Qpid*), to answer the following three research questions:

RQ1: ***What are the reasons for changing logging statements?***
Through a manual analysis of a statistically representative sample of logging statements, we find that the reasons for log changes can be grouped along four categories: block change, log improvement, dependence-driven change, and logging issue.

RQ2: ***How well can we provide just-in-time log change suggestions?***
We build random forest classifiers using software measures that are derived from our manual study in RQ1, and from our experience, in order to model the drivers for log changes in a code commit. We evaluate our classifiers in both a within-project and a cross-project evaluation. For our within-project evaluation, we build a random forest classifier for every code commit using all previous code commits as training data, in order to suggest whether a log change is needed for the current commit. The random forest classifiers that are built from historical data from the same project achieve a balanced accuracy of 0.76 to 0.82. For our cross-project evaluation, we build random forest classifiers that are trained from three out of four studied projects and suggest log changes in the remaining project. We repeat the process for each of the studied projects. The classifiers reach a balanced accuracy of 0.76 to 0.80 and an AUC of 0.84 to 0.88.

RQ3: ***What are the influential factors that explain log changes?***
Factors which capture characteristics about the changes to the non-logging code in a commit (i.e., change measures, such as the number of changed control flow statements) and factors that capture characteristics of the current snapshot of the source code (i.e., product measures, such as the number of existing logging statements) are the most influential factors for explaining log changes in a commit. In particular, change measures are the most influential explanatory factors for log additions, while product measures are the most influential explanatory factors for log modifications.

Paper organization. The remainder of the paper is organized as follows. Section 2 surveys related work on software logs. Section 3 describes the studied systems and our experimental setup. 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 characteristics of commits that only change logging statements without changing the non-logging code. Section 6 discusses the threats of validity. Finally, Section 7 draws conclusions based on our presented findings.

2 Related Work

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

2.1 Leveraging logs

A large amount of log related research focuses on postmortem diagnosis of logs [22, 23, 28, 38, 39]. Since console logs, generated in large-scale data centers, often consist of voluminous messages and it is difficult for operators to detect noteworthy logs, Xu et al. [38] propose a method to mine logs to automatically detect system runtime problems within a short time. As field logs and source code are usually the only resource for developers to diagnose a production failure, Yuan et al. [39] propose a tool named SherLog, which leverages run-time log information and source code to infer the execution path of a failed production run. The tool requires no expert knowledge of the product. Mariani's work [22] also proposes a technique, which automatically analyzes log files and retrieves valuable information, to assist developers in identifying failure causes. The widespread usage of logs highlights the importance of proper logging practices, and motivates our work to propose an approach that can provide effective suggestions for log changes.

2.2 Improving logs

Prior research proposed approaches to improve logging statements. In order to address the lack of log messages for failure diagnosis, *Errlog* [41] analyzes the source code to detect unlogged exceptions (abnormal or unusual conditions) and automatically inserts the missing logging statements. *LogEnhancer*[40], on the other hand, automatically adds additional *causally-related* information to existing logging statements to aid in future failure diagnosis. A recent tool named *LogAdvisor* [44] 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 as a post-implementation process. In this paper, in comparison, we propose an proactive approach that can provide just-in-time log change suggestions to developers. We provide guidance to developers by suggesting the need for a log change when developers commit code changes.

2.3 Empirical studies of logging practices

Prior research has empirically studied logging practices in both open source and industrial software projects. Yuan et al. [42] perform a study of logging practices in four open source projects. They study the current practices of modifying logging code. Their work notes several observations on the amount of efforts that developers often spend on modifying log messages, as well as observations on where developers spend most of their efforts modifying the log messages. Similarly, prior research also shows that logs are often changed by developers without considering the needs of other stakeholders [30, 32], and that changes to logs often break the functionality of log processing applications which are highly dependent on the format of logs. Another study [41] investigates the efficacy of logs for failure diagnosis across five large software systems, and finds that more than half of the failures could not be diagnosed using existing log information. Fu et al. [10] study the logging practices in two industrial software projects. They investigate in what kinds of code snippets do developers add logs and provide guidelines on where to log. In this paper, we focus on understanding what causes developers to change logging statements instead of what kind of code needs to be logged.

3 Case Study Setup

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

3.1 Subject Systems

This paper studies the reasons for log changes and explores the feasibility of providing accurate just-in-time suggestions for log changes through a case study on four open source projects: *Hadoop*, *Directory Server*, *Commons HttpClient*, and *Qpid*. All the four projects are mature Java projects with years of development history and from different domains. Table 1 shows the studied development history for each project. We use the "svn log"[8] command to retrieve the development history for each project (i.e., the svn commit records). We analyze the development history of the main branch (trunk) of each project, and focus on Java source code (excluding Java test code). Some commits import a large number of atomic commits from a branch into the trunk (a.k.a. merge commits), which usually contain a large amount of code changes and log

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

Table 1 Overview of the studied systems.

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

changes. Such merge commits would introduce noise in our study [45] of log changes in a commit. We unroll each merge commit into the various commits of which it is composed (using the "use-merge-history" option of the "svn log" command).

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

3.2 Data Extraction

Figure 1 presents an overview of our data extraction and data analysis approaches. From the version control repositories of each subject system, we analyze the code changes in each commit and identify the commits that contain changes to logging statements. As a result, we are able to create a log-change database (i.e., a collection of log changes) and label each commit as to whether it contains log changes or not. The log-change database is used in our manual analysis (RQ1), and the labeled commit data is employed in our modeling analysis (RQ2 and RQ3).

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

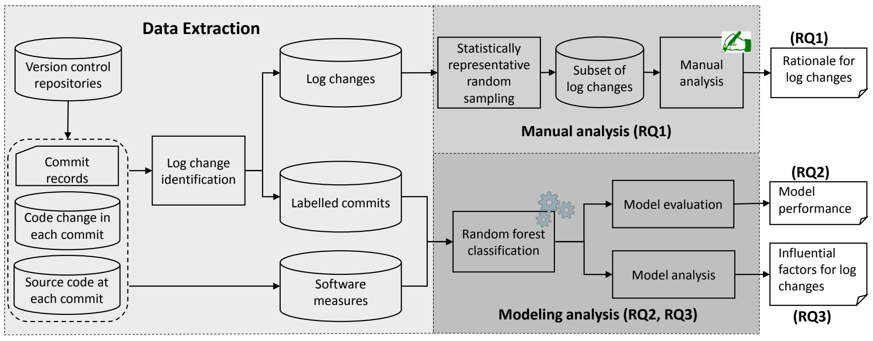
Data Extraction
Version control repositories
Commit records
Code change in each commit
Source code at each commit
Log change identification
Log changes
Labelled commits
Software measures
Statistically representative random sampling
Subset of log changes
Manual analysis
Manual analysis (RQ1)
(RQ1)
Rationale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

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

Within the commits that change logging statements, there are only 1.2% to 4.2% of them that do not change other source code (i.e., log-changing-only commits). Since our models aim to provide developers with just-in-time suggestions for log changes when they are changing other source code, we exclude these log-changing-only commits in our modeling analysis. We revisit the characteristics of the log-changing-only commits at the end of the paper, in Section 5.

3.3 Log Change Identification

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

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

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

We use regular expressions to identify the added and deleted logging statements across commits (see on-line replication package for the used regular

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

expressions[13]). If a pair of an added logging statement and a deleted one are within the same code snippet and they are textually similar to each other, the pair of logging statements are considered as a log modification. Otherwise they are considered as one log addition and one log deletion. We measure the textual similarity between two logging statements by calculating the Levenshtein distance ratio [20] between their concatenation of static text and variable names. Two logging statements are considered similar if the Levenshtein distance ratio between them is larger than a specified threshold for which we choose 0.5 in this paper (see Section 6 for a sensitivity analysis of the impact of this threshold on the identification of log modifications).

4 Case Study Result

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.

RQ1: What are the reasons for changing logging statements?

Motivation

Before proposing an approach that can provide just-in-time suggestions for log changes, we first conduct a manual study in order to investigate the reasons for changing logging statements. Our manual observation will assist us in defining appropriate measures that we can use later on to build models to provide just-in-time suggestions for log changes when developers commit code changes.

Approach

There is a total of 32,480 logging statement changes in the studied commits of the four studied projects (9,503 for Hadoop, 11,883 for DirectoryServer, 2,333 for HttpClient and 8,761 for Qpid). Each commit may contain multiple logging statement changes. We randomly selected a statistically representative sample (95% confidence level with a $\pm 5\%$ confidence interval) of 380 log changes. Among the 380 log changes, there are 204 log additions, 91 log modifications, and 85 log deletions. We manually examine the possible reasons for these log changes. For each log change, we check the log change itself, the co-changed code, the commit message, and the associated issue report if an issue id is noted in the commit message. Certain log change reasons (e.g., a typo) can be detected by only looking at the log change itself. Examining the co-changed code can help us determine the log change reasons such as "a logging statement is changed because the logged variables are changed". The commit message

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

and the issue report directly communicate the intention of the developer and the issue owner for a log change. The first and second author of this paper work together by manually examining all log changes from the random sample. We examine the log change, code change, commit message and the associated issue report to understand the reason of a log change. If the reason is new, we add it to the list of identified reasons. If there is a disagreement during the process, the two authors discuss and reach a consensus.

Table 2 Log-change reasons and the distribution: manual analysis result.

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

Results

We find 20 reasons for log changes across four categories: changing context code, improving logging, dependency-driven changes and fixing logging issues. Table 2 summarizes the log change reasons. We present below the four categories of reasons for log changes.

Block change. Logging statements are added (or deleted) as a result of the change of the surrounding code blocks. According to our manual analysis, logging statements are added (or deleted) when developers are adding (or deleting) try-catch blocks, adding (or deleting) methods, adding (or deleting) branches (if branches and switch branches), adding (or deleting) if-null branches (if branches checking an abnormal condition), and adding (or deleting) loops (for loops and while loops). For example, the following code snippet indicates that a logging statement is added to record the error information as part of the newly added try-catch block. (Note: the plus sign (+) or minus sign (-) leading a code line indicates that the code line is added or deleted in that particular commit.)

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

Log improvement. Logging statements are added, deleted or modified to achieve a better logging practice. Developers change logging statements (e.g., by adding a logging statement which tracks the value of a variable) to improve the debugging capability of the logged information. They also change a logging statement to improve the readability of the logged information; for example, they rephrase a logging statement such that the log message would be easier to understand. Some logging statements are changed to leverage log message translation method (i.e., using predefined code such as "I18n.ERR_115" to represent a log message). Developers also change logging statements, for example, by adding a logging statement to record the occurrence of an event, to improve the logged runtime information. Removing redundant log information is another way to improve the logging of a system; the redundant log information includes duplicated log information and unnecessary log information. Developers sometimes improve their logging by migrating from an old logging style (e.g., "System.out") to a more advanced logging library (e.g., Log4j) (i.e., log library migration [16]). Finally, we also find that a logging statement is removed because of a security issue that is mentioned in the associated issue report. The following code listing shows that a logging statement is added to a method in order to enhance the debugging capability. The commit message states that the developer added "some extra debug log entries for the authentication process".

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

Dependence-driven change. Logging statements are changed because they depend on other code elements (e.g., variables) that are changed by developers. A log change might be driven by the change of a logger (i.e., an class object that is used to invoke a logging method), a variable, a method or a class. We also find that a logging statement is changed to remove its dependence to a different module to remove the coupling between modules. The following examples shows that a logging statement is modified because the method ("DatanodeID:getName") that it depended on has been replaced by a new method ("toString"). The reason of the log changes is recorded in the commit message and the associated issue report[14].

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

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

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

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

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

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

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

Motivation

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

Approach

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

In order to model the drivers for log changes, we extract and calculate a set of measures from three dimensions: *change measures*, *historical measures*, and *product measures.* Table 3 presents a list of measures that we collect for each dimension. Table 3 also describes our proposed measures and explains our motivation behind each measure. We build classifiers at the granularity of a code commit, thus we calculate all of our proposed measures for very commit during the studied development history. We describe below each dimension of measures:

- **Change measures** capture the changes in the commit itself, represented by the changes of control flow statements (e.g., *try statement*, *if statement*), and the type of a commit (*commit type*, Bug/Improvement/New Feature/Task/Subtask/Test). We choose the *change measures* according to our manual analysis results. As shown in the results of RQ1, most log changes are accompanied with contextual code changes (i.e., *block change* and *dependence-drive change*). For example, *adding/deleting try-catch block* is one of the most frequent reasons for a log change. The *commit type* captures the context or purpose of the code change; we use the "type" field of the JIRA issue report that is linked to the commit.
- **Historical measures** capture the code changes throughout the development history (before the considered commit) of the changed files. Based on our intuition, source code files undergoing frequent log changes in the past may have log changes in the future. Besides, prior research shows that files with high churn rate are more defect-prone [26, 27], and developers are likely to add more logs in defect-prone source code files [31].
- **Product measures** capture the current snapshot of the source code, represented by the status of logging statements and other source code, of the software system just before the considered commit. For example, *log number* and *log density* capture the log-intensiveness of the changed code. Our manual analysis in RQ1 shows that many log changes are committed to improve existing logging (i.e., the *log improvement* reasons) or fix logging issues (i.e., the *logging issue* reasons). Thus the code changes on log-intensive code are more likely to involve log changes. In addition, based on our intuition, the appropriateness of logging statements should be related to their contextual source code. Therefore, we also calculate several product measures that capture the contextual source code of logging statements (i.e., *SLOC*, *McCabe complexity* and *fan-in*).

For the *if statement* measure from the dimension of *change measures*, we do not consider *if* statements that act as logging guards. An example of such *if* statements is the one listed below:

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

Correlation analysis. Prior to constructing the classifiers for log changes, we check the pairwise correlation between our proposed measures using the Spearman rank correlation test (ρ). Specifically, we use the "varclus" function in the "Hmisc" R package to cluster measures based on their Spearman rank correlation. We choose the Spearman rank correlation method because it is robust to non-normally distributed data [24]. In this work, we choose the correlation value 0.8 as the threshold to remove collinearity. In other words, if the correlation between a pair of measures is greater than 0.8 ($|\rho| > 0.8$),

Table 3 Software measures used to model the drivers for log changes, measured per each commit.

Dimension	**Measures**	**Definition (d) — Rationale (r)**
Change measures	class declaration	d: Number of changed class declarations in the commit.
		r: Developers might add logging statements in a new class so that they can better observe the behavior of the class.
	method declaration	d: Number of changed method declarations in the commit.
		r: Developers might add logging statements in a new method so that they can better observe the behavior of the method.
	try statement	d: Number of changed *try* statements in the commit.
		r: Logging statements often reside inside *try* blocks; hence logging statements are likely to co-change with *try* statements.
	catch clause	d: Number of changed *catch* clauses in the commit.
		r: Exception catching code is often logged [10, 42, 44]; hence logging statements are likely to co-change with *catch* clauses.
	throw statement	d: Number of changed *throw* statements in the commit.
		r: A logging statement is often inserted right before a *throw* statement [10]; hence developers changing a *throw* statement are likely to change the corresponding logging statement.
	throws clause	d: Number of method definitions with *throws* clauses (which declare that a method can throw exceptions) changed in the commit.
		r: Methods that throw exceptions are likely to have logging statements; thus logging statements might co-change with *throws* clauses.
	if statement	d: Number of changed *if* statements in the commit.
		r: Logging statements are usually inside *if* branches [10, 44]; thus logging statements are likely to co-change with *if* statements.
	if-null statement	d: Number of changed *if-null* statements (if statements with null condition, e.g., “if (outcome == NULL)”) in the commit.
		r: *if-null* branches are usually corner-case execution paths which are likely to be logged [10, 44]; thus logging statements might co-change with *if-null* blocks.
	else clause	d: Number of changed *else* clauses in the commit.
		r: Logging statements are usually inside *if-else* branches [10, 44]; thus logging statements are likely to co-change with *else* clauses.
	for statement	d: Number of changed *for* statements in the commit.
		r: Logging statements inside *for* loops usually record the execution path or status of the *for* loops; hence these logging statements are likely to co-change with the *for* statements.
	while statement	d: Number of changed *while* statements in the commit.
		r: Logging statements inside *while* loops usually record the execution path or status of the *while* loops; hence these logging statements are likely to co-change with the *while* statements.
	commit type	d: Change type of the commit: Bug/Improvement/New Feature/Task/Subtask/Test.
		r: Change type characterized the context of a code change, thus it might affect developers’ logging behavior.
Historical measures	log churn in history	d: Number of changed logs in the development history of the involved files.
		r: Files experiencing frequent log changes in the past might expect frequent log changes in the future.
	log churn ratio in history	d: Ratio of the number of changed logging statements to the number of changed lines of code in the development history of the involved files.
		r: Files experiencing frequent log changes in the past are likely to exhibit frequent log changes in the future.
	log-changing commits in history	d: Number of commits involving log changes in the development history of the involved files.
		r: Files experiencing frequent log changes in the past are likely to exhibit frequent log changes in the future.
	code churn in history	d: Number of changed lines of code in the development history of the involved files.
		r: Frequently changed code are problem-prone thus are more likely to be logged.
	commits in history	d: Number of commits in the development history of the involved files.
		r: Frequently changed code are problem-prone thus are more likely to be logged.

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

we keep one of the two measures in the classifier. We find that the measures that are listed in Table 3 present similar patterns of correlation across all four studied projects, thus we drop (i.e., do not consider in the classifier) the same measures for all the studied projects. Dropping the same set of measures for the projects enables us to build cross-project classifiers as discussed in the "Cross-project Evaluation" part that follows. We combine the data of the four projects together and perform correlation analysis on the combo data. Figure 2 shows the result of the correlation analysis on the combo data, where the horizontal bridge between each pair of measures indicates the correlation, and the red dotted line represents the threshold value (0.8 in this case). The results of our correlation analysis on each individual project can be downloaded at a public link[15]. To ease the interpretation of the classifier, we try to keep the one that is easier to understand and calculate from each pair of highly-correlated measures. For example, the *SLOC* and *McCabe complexity* measures have a correlation higher than 0.8, we keep the *SLOC* measure and drop the *McCabe complexity* measure. Based on the result shown in Figure 2, we drop the following measures: *log churn in history*, *log-changing commits in*

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

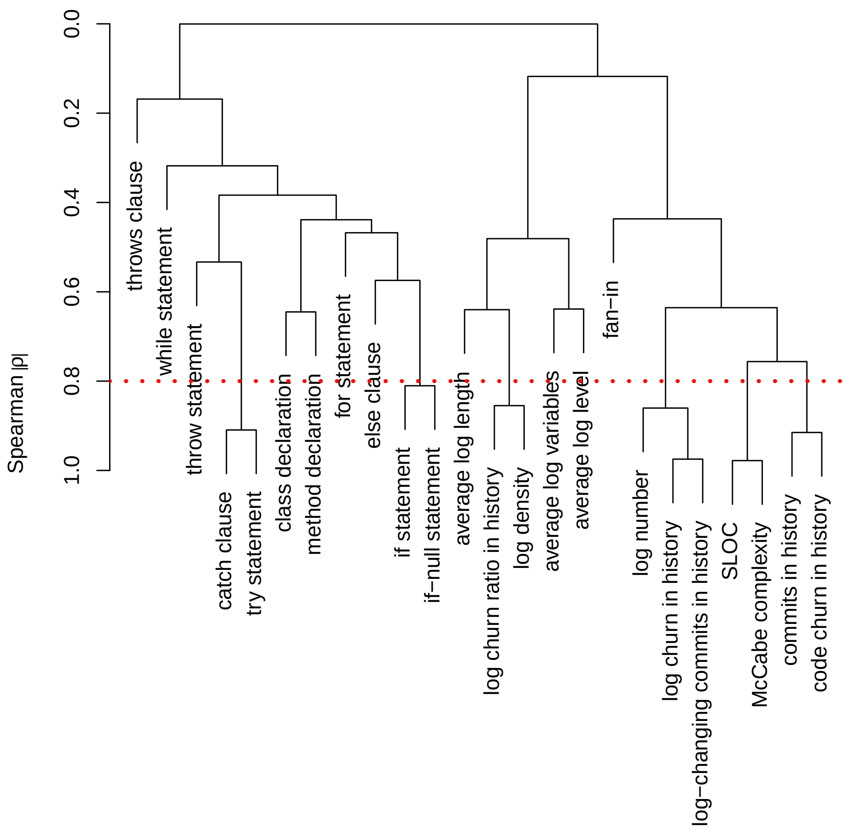

Fig. 2 Correlation analysis using Spearman hierarchical clustering (for the combo data).

history, *McCabe complexity*, *commits in history*, *log churn ratio in history*, *try statement* and *if-null statement*, as they are highly correlated with other measures.

Modeling technique. We build random forest classifiers to model the drivers for log changes. A random forest is a classifier consisting of a collection of decision tree classifiers and each tree casts a vote for the most popular class for a given input [3]. Random forests construct each tree using a different bootstrap sample (i.e., if the number of instances in the training set is N, randomly sample N instances with replacement) of the input data as the training set. The random forest classifier uses a bootstrap approach internally to get an unbiased evaluation of the performance of a classifier [3]. In addition, unlike standard decision trees where each decision node is split using the best split among all variables, random forests split each node using the best among a randomly chosen subset of variables from each of the constructed trees [21].

Random forests are naturally robust against overfitting, and they perform very well in terms of accuracy [3]. Random forest provides us a way to do sensitivity analysis on the measures so that we can understand the most influential factors in our classifiers [4, 21]. Besides, a recent study [12] compares 31 classifiers in software defects prediction and suggests that Random Forest outperforms other classifiers.

Within-project Evaluation. We build a random forest classifier to determine the likelihood of a log change for each commit based on the development history prior to that particular commit. Specifically, for each commit, we build a random forest classifier using all the prior commits as training data, and use the classifier to determine whether a log change is needed for that particular commit. A classification result can be "true" (i.e., the likelihood of a log change is higher than 0.5) or "false" (i.e., the likelihood of a log change is lower than 0.5). A "true" classification result suggests the need of log changes in that code commit, while a "false" classification result suggests that no log change is needed for that commit. Then, we update the classifier with the new commit and use the updated classifier to determine the likelihood of a log change for the following commit, and so on. Evaluating the classification result for a commit can have one of four outcomes: **TP** - true positive, **FP** - false positive, **FN** - false negative, and **TN** - true negative. The outcomes are illustrated in the confusion matrix that is shown in Table 4.

We use *balanced accuracy* (**BA**) as prior research [44] to evaluate the performance of our within-project evaluation. BA averages the probability of correctly identifying a log-changing commit and the probability of correctly identifying a non-log-changing commit. BA is widely used to evaluate the modeling results on imbalanced data [5, 43, 44], because it avoids over-optimism on imbalanced data. BA is calculated by Equation (1):

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

We determine the likelihood of a log change for each commit throughout the lifetime of a project, using a random forest classifier that is trained from all the prior commits of the same project, and get an outcome that is represented by one of TP, FP, FN and TN. We train the first classifier for each project when there are 50 commits in the development history; in other words, we evaluate our first classifier on the 51st commit. For each project, we sum up the TP, FP, FN and TN for all commits (except for the first 50 commits) and apply Equation 1 to calculate the overall performance of our just-in-time suggestions that is represented by a BA. Moreover, in order to observe the evolution of our classifier's performance over the lifetime of each project, we use a "sliding window" technique to calculate the BA for each commit. Specifically, the "sliding window" of a particular commit contains 101 consecutive commits, including 50 preceding commits, the commit itself, and 50 following commits. In order to get the BA for the particular commit, we sum up the TP, FN, TN and FP in the "sliding window" and then calculate the averaged BA using Equation (1). Each time we move the sliding window forward by one commit to calculate the BA for the next commit. The BA for each commit that is calculated from the "sliding window" enables us to examine the stability of the performance of our just-in-time suggestions, and whether the suggestions are accurate when there are only a small number of commits available to train a classifier at the start of a project.

Table 4 Confusion matrix for the classification results of a commit.

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

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

We evaluate the BA of the cross-project classifiers. For each testing project, we sum up the TP, FN, TN and FP that are computed from determining the likelihood of a log change of all the commits of the project, and apply Equation (1) to calculate the BA.

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

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

Results

Our random forest classifiers can effectively provide just-in-time suggestions for log changes using historical data from the same

project. The overall BA values when considering all commits in Hadoop, DirectoryServer, HttpClient, and Qpid are 0.76, 0.83, 0.77, 0.77, respectively (see Table 5). Figure 3, Figure 4, Figure 5 and Figure 6 illustrate the within-project classification results using the "sliding window" technique for Hadoop, DirectoryServer, HttpClient, and Qpid, respectively. For each figure, the horizontal axis denotes the commit index while the vertical axis shows the BA value. The black solid curve plots the BA value at each commit, and the red dashed line indicates an average BA over all the commits of the project. These figures show that our random forest classifiers achieve an average BA of 0.76 to 0.82. In other words, given a commit, our classifiers can tell whether this commit should change logging statements or not, with an average accuracy of 0.76 to 0.82. Table 6 presents the detailed TP, FN, TN, and FP numbers that are used to calculate the overall BAs using Equation 1. Taking the Hadoop project for example, among the 1,541 actual logging commits (TP + FN), 76% (1,166) of them are correctly classified as logging commits, and 24% (375) of them are incorrectly classified as non-logging commits; among the 3,718 actual non-logging commits (TN + FP), 76% (2,822) of them are correctly classified as non-logging commits, and 24% (896) of them are incorrectly classified as logging commits. On average, training such a random forest classifier for a large system like Hadoop on a workstation (Intel i7 CPU, 8G RAM) takes about 2 seconds, and classifying the log changes for a particular commit takes about 0.02 seconds. For each commit, we would only need to perform the classification step in real-time, while the training can be done offline. These results indicate that our within-project classifiers can effectively provide just-in-time suggestions for log changes.

Our classifier achieve the average balanced accuracy with a small number of commits as training data. We measure when a project accumulates sufficient commit data to train a classifier that reaches the average performance in terms of BA. In Figure 3, Figure 4, Figure 5 and Figure 6, we've marked the commit number where each classifier reaches its average BA for the first time. We find that the within-project classifiers for Hadoop, DirectoryServer, HttpClient, and Qpid reach their average BAs when the projects got 209, 193, 211 and 155 commits, respectively. The results indicate that the classifiers can learn an average classification power after a relatively small number of commits.

The performance of our classifiers fluctuates over time. As shown in Figure 3 through 6, the performance (in terms of the BA) of the within-project classifiers fluctuates over time, and the fluctuation does not follow any clear trend. These results might be explained by the assumption that developers follow different logging practices at different development stages. For example, developers might be less focused on logging at the beginning of a release cycle and might pay more attention on logging when a product is approaching its release (and final testing is being performed on it). Table 5 shows the average BA, 5 percentile BA and 95 percentile BA for the within-project classifiers over time. Only 5% of the commits get a BA smaller than 0.68, 0.71, 0.66 and 0.68 for Hadoop, DirectoryServer, HttpClient, and Qpid,

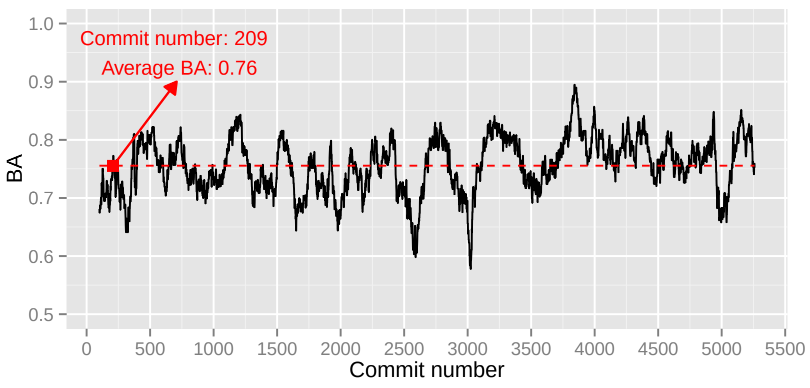

Fig. 3 The balanced accuracy of the within-project classifiers for Hadoop.

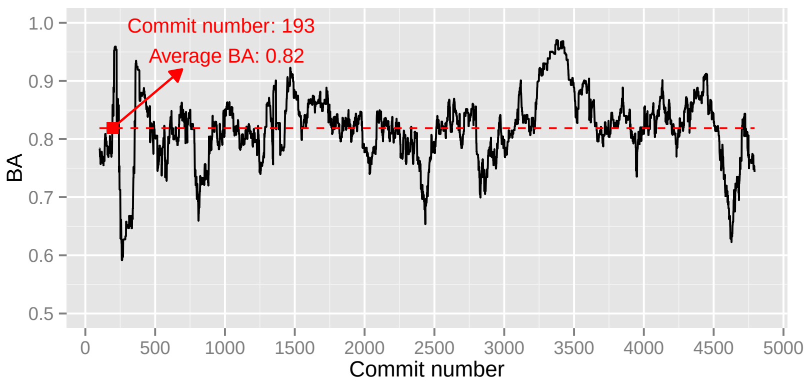

Fig. 4 The balanced accuracy of the within-project classifiers for DirectoryServer.

respectively. And 5% commits get a BA bigger than 0.82, 0.92, 0.83 and 0.88, respectively.

Table 5 The BA results for the within-project evaluation.

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

Our random forest classifiers can effectively provide just-in-time suggestions for log changes using the development history of other projects. Table 7 lists the performance of the cross-project classifiers, ex-

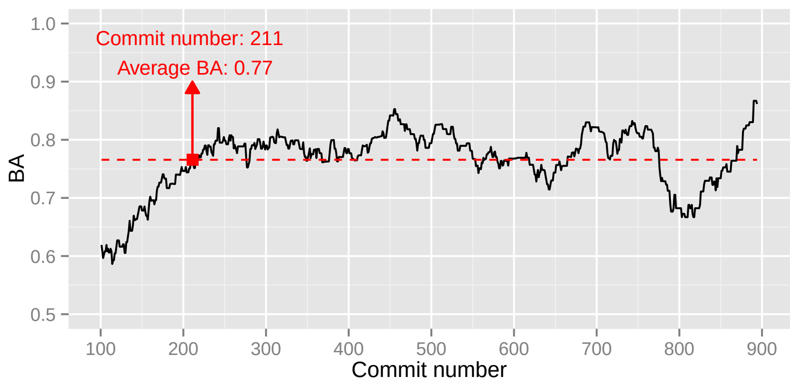

Fig. 5 The balanced accuracy of the within-project classifiers for HttpClient.

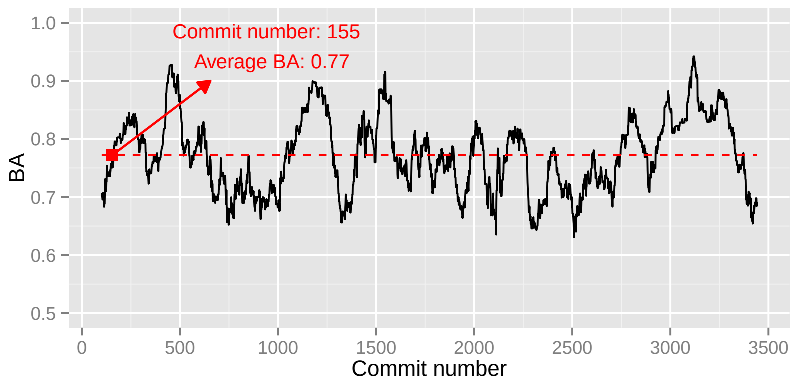

Fig. 6 The balanced accuracy of the within-project classifiers for Qpid.

Table 6 The TP, FN, TN, FP results for the within-project evaluation.

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

pressed by the BA and the AUC measures. Each row of the table shows the performance of the classifier that uses the specified project as testing data and all the other projects as training data. The cross-project classifiers reach a BA of 0.76 to 0.80, indicating that the cross-project classifiers can effectively determine the likelihood of a log change for each commit of a project with little development history.

Our random forest classifiers can effectively discriminate log-changing commits and non-log-changing commits. As shown in Table 7, the cross-project classifiers achieve an AUC of 0.84 to 0.88. The high AUC values indicate that the classifiers perform much better than random guessing in discriminating the log-changing commits and non-log-changing commits.

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

Table 7 The BA and AUC results for the cross-project evaluation.

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

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

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

Motivation

In order to quantitatively understand the reasons for log changes, we analyze the random forest classifiers to find out the most influential factors that are associated with log changes. There are three types of log changes: log additions, log deletions, and log modifications. The influential factors for log changes, log additions, log deletions and log modifications may be different. Therefore, we analyze the influence of factors for log changes, log additions, log deletions, and log modifications, separately. From our manual analysis in RQ1, we find that the occurrences of deleting a logging statement is usually associated with deleting the enclosing code block (see Table 2). Therefore, we do not analyze the factors that influence log deletions.

Approach

Bootstrap analysis. In order to study the influential factors that affect log changes, we use the bootstrap method to repeatedly sample training data and build a large number of random forest classifiers, so as to statistically analyze the influence of the factors. Bootstrap [7] is a general approach to infer the relationship between a sample data and the overall population, by resampling the sample data with replacement and analyzing the relationship between the resample data and the sample data. The bootstrap analysis is implemented in the following steps:

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

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

Variable influence in Random Forest. The random forest classifier evaluates the influence of each factor by permuting the values of the corresponding measure of that factor while keeping the values of the other factors unchanged in the testing data (the so-called "OOB" data) [3]. The classifier measures the impact of such a permutation on the classification error rate [4, 21]. In each round of the 1,000 bootstraps, we use the "importance" function in the R package "randomForest" to evaluate the influence of the factors.

Scott-Knott clustering. In this step, we compare the average influence of the factors from the 1,000 bootstrap iterations. However, the differences among the influence of some factors might actually be due to random variability. Thus we need to partition all the factors into statistically homogeneous groups so that the influence means of the factors within the same group are not significantly different (i.e, $p\,value \geq 0.05$). The Scott-Knott (SK) algorithm is a hierarchical clustering approach that can partition the results into distinct homogeneous groups by comparing their means [15, 29]. The SK algorithm hierarchically divides the factors into groups and uses the likelihood ratio test to judge the significance of difference among the groups. The SK method generates statistically distinct groups of factors, i.e., each two groups of factors have a $p\,value < 0.05$ in a likelihood ratio test of their influence values.

In this work, we use an enhanced SK approach [36], which considers the effect size in addition to the statistical significance of the difference between groups, to divide the factors into distinct groups according to their influence in the random forest classifiers.

We repeat our approach (bootstrap analysis, variable influence and Scott-Knott clustering) for the log addition classifier and the log modification classi-

Table 8 The influence mean of the top 10 factors (measures) for the log change classifier, divided into distinct homogeneous groups by Scott-Knott clustering.

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

fier to study the most influential factors for log additions and log modifications respectively.

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

Results

***Change measures* and *product measures* are the most influential factors for log changes.** Table 8, Table 9 and Table 10 present the influence values of the 10 most influential predictor variables for the log change classifier, the log addition classifier and the log modification classifier, respectively. For each studied project, we measure the mean value of each factor's influence. The factors are divided into statistically distinct groups by a Scott-Knott test on the influence values. Overall, the most influential factors in these classifiers include the *if statement* (with a median group ranking of 2), *catch clause* (with a median group ranking of 3) and *method declaration* (with a median group ranking of 5) measures from the dimension of *change measures*, as well as the *log density* (with a median group ranking of 2), *log number* (with a median group ranking of 3.5), *average log variables* (with a median group ranking of 6), *average log length* (with a median group ranking of 6.5) and *average log level* (with a median group ranking of 6.5) measures from the dimension of *product measures.*

The strong influence of the measures from the *change measures* dimension indicates that log changes are highly associated with other code changes, and

Table 9 The influence mean of the top 10 factors (measures) for the log addition classifier, divided into distinct homogeneous groups by Scott-Knott clustering.

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

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

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

this is in accordance with the fact that the *block change* reasons are the most frequent reasons that we observed in our manual analysis. In particular, the influence of the *catch clause* measure suggests a strong association between exception handling and logging, and this matches with the *adding/deleting try-catch block* reason that is observed in our manual analysis. This result also quantitatively supports best practices recommendation that exception-handling code should log the information that is associated with the exception being handled [1, 25]. The strong influence of the *if statement* and *if-null statement* measures (*if-null statement* has a high correlation with *if statement*) shows that developers tend to change logging statements while changing conditional branches, and this corresponds to the *adding/deleting branch* and *adding/deleting if-null branch* reasons that we observed in RQ1. As another influential factor from the *change measures* family, the *method declaration* measure suggest that the change of a method declaration is strong indicator for log changes. Again, this result consents with manually detected reason *adding/deleting method.*

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

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

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

of the commits for the *HttpClient* project contain changes to *catch clauses*, while that proportions are 21.0%, 20.3% and 27.5% for the *Hadoop*, *Directory Server*, and *Qpid* projects, respectively.

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

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

5 Discussion

Discussion regarding log-changing-only commits. Our random forest classifiers aim to provide developers with just-in-time log change suggestions when they commit code changes. However, developers sometimes also change logging statements without affecting other source code (we call such commits as log-changing-only commits). In such cases, our approach cannot provide just-in-time log change suggestions. As shown in Table 11, the log-changing-only commits take up 1.2% to 4.2% of all the commits that change logging statements. By manually examining all these log-changing-only commits, we find that a log-changing-only commit occurs when either 1) the developers have not correctly implemented the needed logging statements in the first place, or 2) the requirements of logging have changed afterwards. The following example shows that a logging statement was missing in an exception handling code, which increased the difficulty of finding the root cause of a reported bug (i.e., QPID-1352[16]). A log-changing-only commit (704187) added a logging statement in the exception handling block so that "hopefully the next time it shows up we have a bit more info".

```
/* Project: Qpid; Commit:704187
 * File: /incubator/qpid/trunk/qpid/java/client/src/main/java/
     org/apache/qpid/client/AMQConnection.java
 * Commit message: "log the original exception so we don't lose the stack
     trace"
```

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

Table 11 Number of log-changing-only commits.

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

```
 * JIRA issue report: "I've committed a change to log the original error. I
      can't reproduce this on my system, so hopefully the next time it shows
      up we have a bit more info."
 */
  catch (AMQException e)
  {
+   _logger.error("error:", e);
    JMSException jmse = new JMSException("Error closing connection: " + e);
    jmse.setLinkedException(e);
    throw jmse;
  }
```

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

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

Table 11 presents the number of log-changing-only commits that add, delete and modify logging statements, respectively. 88 out of 129 (68.2%) log-changing-only commits modify existing logging statements, while 38 (29.5%) of them add new logging statements. Developers are least likely to delete a logging statement without changing other source code: only 24 (18.6%) of the log-changing-only commits delete logging statements. If developers forget to add, delete or modify a logging statement, leading to a log-changing-only commits afterwards, our approach would help developers avoid forgetting to change logs in the first place. If current logging statements need to be fixed or improved, our approach cannot provide such suggestion without the context of other code changes. In the studied projects, however, the log-changing-only commits represent 1.2% to 4.2% of the commits that change logging statements.

6 Threats to Validity

6.1 External Validity

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

6.2 Internal Validity

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

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

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

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

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

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

6.3 Construct Validity

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

We identify a log modification by calculating the Levenshtein distance ratio between a pair of added and deleted logging statements. Two logging statements are considered similar if the Levenshtein distance ratio between them is larger than a specified threshold (L-threshold). In this paper, we choose an L-threshold value of 0.5 to identify a log modification. However, this approach is not guaranteed to identify all log modifications. In order to address this threat, we perform a sensitivity analysis to measure the impact of the L-threshold value on the identification of log modifications for all the log changes in the Hadoop project. Specifically, we change the L-threshold to 0.4 and 0.6 and examine the difference of the identification results for these two thresholds. Table 12 shows summary statistics of the identification results using the 0.4, 0.5 and 0.6 L-thresholds. We identify 1,861 log modifications using the 0.5 threshold; while we identify 1,932 (+3.8%) and 1,788 (−3.9%) modifications for the 0.4 and 0.6 L-thresholds, respectively. The results show that using a different L-threshold does not lead to a large change in the number of identified log modifications.

We examine the amount of added logging statements that are classified differently (i.e., classified as a log addition or a part of a log modification) when using different L-thresholds. We find that there are only 248 out of 7,215 (3.4%) added logging statements that are classified differently using different L-thresholds. We manually examine these 248 added logging statements that are classified differently and identify which L-threshold can accurately classify the added logging statements. We find that using 0.5 as a threshold has the highest accuracy. In particular, 106 out of the 248 (42.7%) added logging statements that are classified differently are correctly classified by using the 0.4 threshold; 162 (65.3%) of the added logging statements are correctly classified using the 0.5 threshold; and 149 (60.1%) of the added logging statements are correctly classified using the 0.6 threshold.

Table 12 L-threshold's impact on the identification of log modifications (for the Hadoop project). The values in the brackets are the percentages of difference when compared with the 0.5 L-threshold.

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

In this paper we choose 25 software measures (as listed in Table 3) based on our manual analysis results and our intuition. However, there may be other measures, such as OO measures [6], that can assist in suggesting log changes. Some feature learning techniques may also help improve the performance of our classifiers. As a first step of suggesting proper log changing practices, we aim to show the benefit of leveraging historical information for providing log change suggestions in the future. We expect more measures or feature learning techniques to be proposed in follow-up research in order to provide more accurate classifiers to suggest log changes.

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

7 Conclusions

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

– Log change reasons can be grouped along four categories: block change, log improvement, dependence-driven change, and logging issue.

- Our random forest classifiers can provide accurate just-in-time suggestions for log changes. The classifiers trained from historical data of the same project achieve a balanced accuracy of 0.76 to 0.82; the classifiers trained from other projects reach a balanced accuracy of 0.76 to 0.80 and an AUC of 0.84 to 0.88.
- Code changes in a commit and the current snapshot of the source code are the most influential factors for determining the likelihood of a log change in a commit.

Our work provides insights about the reasons why developers change (add, modify or delete) logging statements in their code. Inexperienced developers may learn from these reasons to understand the current logging practice (e.g., the need to add logging statements when changing catch blocks). Our findings also show that developer can leverage machine learning models to guide their log changing practices.

References

1. Apache-Commons (2016) Best practices - logging exceptions. `http://commons.apache.org/proper/commons-logging/guide.html`
2. Bitincka L, Ganapathi A, Sorkin S, Zhang S (2010) Optimizing data analysis with a semi-structured time series database. In: Proceedings of the 2010 Workshop on Managing Systems via Log Analysis and Machine Learning Techniques, USENIX Association, Berkeley, CA, USA, SLAML'10, pp 7–7
3. Breiman L (2001) Random forests. Machine learning 45(1):5–32
4. Breiman L (2002) Manual on setting up, using, and understanding random forests v3. 1. Statistics Department University of California Berkeley, CA, USA
5. Cohen I, Goldszmidt M, Kelly T, Symons J, Chase JS (2004) Correlating instrumentation data to system states: A building block for automated diagnosis and control. In: Proceedings of the 6th Conference on Symposium on Opearting Systems Design & Implementation - Volume 6, USENIX Association, Berkeley, CA, USA, OSDI' 04, pp 16–16
6. DAmbros M, Lanza M, Robbes R (2012) Evaluating defect prediction approaches: a benchmark and an extensive comparison. Empirical Software Engineering 17(4-5):531–577
7. Efron B (1979) Bootstrap methods: another look at the jackknife. The annals of Statistics pp 1–26
8. Fu Q, Lou JG, Wang Y, Li J (2009) Execution anomaly detection in distributed systems through unstructured log analysis. In: Proceedings of the 2009 Ninth IEEE International Conference on Data Mining, IEEE Computer Society, Washington, DC, USA, ICDM '09, pp 149–158
9. Fu Q, Lou JG, Lin Q, Ding R, Zhang D, Xie T (2013) Contextual analysis of program logs for understanding system behaviors. In: Proceedings of the 10th Working Conference on Mining Software Repositories, IEEE Press, Piscataway, NJ, USA, MSR '13, pp 397–400
10. Fu Q, Zhu J, Hu W, Lou JG, Ding R, Lin Q, Zhang D, 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, ACM, New York, NY, USA, ICSE Companion '14, pp 24–33
11. Fukushima T, Kamei Y, McIntosh S, Yamashita K, Ubayashi N (2014) An empirical study of just-in-time defect prediction using cross-project models. In: Proceedings of the 11th Working Conference on Mining Software Repositories, ACM, New York, NY, USA, MSR 2014, pp 172–181
12. Ghotra B, McIntosh S, Hassan AE (2015) Revisiting the impact of classification techniques on the performance of defect prediction models. In: Proceedings of the 37th International Conference on Software Engineering - Volume 1, IEEE Press, Piscataway, NJ, USA, ICSE '15, pp 789–800
13. Glerum K, Kinshumann K, Greenberg S, Aul G, Orgovan V, Nichols G, Grant D, Loihle G, Hunt G (2009) Debugging in the (very) large: Ten years of implementation and experience. In: Proceedings of the ACM SIGOPS 22Nd Symposium on Operating Systems Principles, ACM, New York, NY,

USA, SOSP '09, pp 103–116

14. Gülcü C, Stark S (2003) The complete log4j manual. QOS.CH, Lausanne, Switzerland
15. Jelihovschi EG, Faria JC, Allaman IB (2014) Scottknott: A package for performing the scott-knott clustering algorithm in R. Trends in Applied and Computational Mathematics 15(1):3–17
16. Kabinna S, Bezemer CP, Shang W, Hassan AE (2016) 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 (Accepted)
17. Kamei Y, Shihab E, Adams B, Hassan AE, Mockus A, Sinha A, Ubayashi N (2013) A large-scale empirical study of just-in-time quality assurance. IEEE Transactions on Software Engineering 39(6):757–773
18. Kamei Y, Fukushima T, McIntosh S, Yamashita K, Ubayashi N, Hassan AE (2015) Studying just-in-time defect prediction using cross-project models. Empirical Software Engineering pp 1–35
19. Kavulya S, Tan J, Gandhi R, Narasimhan P (2010) An analysis of traces from a production mapreduce cluster. In: Proceedings of the 2010 10th IEEE/ACM International Conference on Cluster, Cloud and Grid Computing, IEEE Computer Society, Washington, DC, USA, CCGRID '10, pp 94–103
20. Levenshtein VI (1966) Binary codes capable of correcting deletions, insertions, and reversals. In: Soviet physics doklady, vol 10, pp 707–710
21. Liaw A, Wiener M (2002) Classification and regression by randomforest. R news 2(3):18–22
22. Mariani L, Pastore F (2008) Automated identification of failure causes in system logs. In: Proceedings of the 2008 19th International Symposium on Software Reliability Engineering, IEEE Computer Society, Washington, DC, USA, ISSRE '08, pp 117–126
23. Mariani L, Pastore F, Pezze M (2009) A toolset for automated failure analysis. In: Proceedings of the 31st International Conference on Software Engineering, IEEE Computer Society, Washington, DC, USA, ICSE '09, pp 563–566
24. McIntosh S, Kamei Y, Adams B, Hassan AE (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, ACM, New York, NY, USA, MSR '14, pp 192–201
25. Microsoft-MSDN (2016) Logging an exception. `https://msdn.microsoft.com/en-us/library/ff664711(v=pandp.50).aspx`
26. Nagappan N, Ball T (2007) Using software dependencies and churn metrics to predict field failures: An empirical case study. In: Proceedings of the First International Symposium on Empirical Software Engineering and Measurement, IEEE Computer Society, Washington, DC, USA, ESEM '07, pp 364–373

27. Nagappan N, Ball T, Zeller A (2006) Mining metrics to predict component failures. In: Proceedings of the 28th International Conference on Software Engineering, ACM, New York, NY, USA, ICSE '06, pp 452–461
28. Oliner A, Ganapathi A, Xu W (2012) Advances and challenges in log analysis. Communications of the ACM 55(2):55–61
29. Scott A, Knott M (1974) A cluster analysis method for grouping means in the analysis of variance. Biometrics pp 507–512
30. Shang W, Jiang ZM, Adams B, Hassan AE, Godfrey MW, Nasser M, Flora P (2011) An exploratory study of the evolution of communicated information about the execution of large software systems. In: Proceedings of the 2011 18th Working Conference on Reverse Engineering, IEEE Computer Society, Washington, DC, USA, WCRE '11, pp 335–344
31. Shang W, Nagappan M, Hassan AE (2013) Studying the relationship between logging characteristics and the code quality of platform software. Empirical Software Engineering pp 1–27
32. Shang W, Jiang ZM, Adams B, Hassan AE, Godfrey MW, Nasser M, 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
33. Shang W, Nagappan M, Hassan AE, Jiang ZM (2014) Understanding log lines using development knowledge. In: ICSME '14: Proceedings of the 30th IEEE International Conference on Software Maintenance and Evolution, IEEE, pp 21–30
34. Sharma B, Chudnovsky V, Hellerstein JL, Rifaat R, Das CR (2011) Modeling and synthesizing task placement constraints in google compute clusters. In: Proceedings of the 2Nd ACM Symposium on Cloud Computing, ACM, New York, NY, USA, SOCC '11, pp 3:1–3:14
35. Syer MD, Jiang ZM, Nagappan M, Hassan AE, Nasser M, Flora P (2013) Leveraging performance counters and execution logs to diagnose memory-related performance issues. In: Proceedings of the 29th IEEE International Conference on Software Maintenance, IEEE, ICSM '13:, pp 110–119
36. Tantithamthavorn C, McIntosh S, Hassan AE, Matsumoto K (2015) An Empirical Comparison of Model Validation Techniques for Defect Prediction Model. Under Review at IEEE Transactions on Software Engineering
37. Tourani P, Adams B (2016) The impact of human discussions on just-in-time quality assurance: An empirical study on openstack and eclipse. In: Proceedings of the 23rd International Conference on Software Analysis, Evolution, and Reengineering, SANER '16, pp 189–200
38. Xu W, Huang L, Fox A, Patterson D, Jordan MI (2009) Detecting large-scale system problems by mining console logs. In: Proceedings of the ACM SIGOPS 22Nd Symposium on Operating Systems Principles, ACM, New York, NY, USA, SOSP '09, pp 117–132
39. Yuan D, Mai H, Xiong W, Tan L, Zhou Y, Pasupathy S (2010) Sherlog: Error diagnosis by connecting clues from run-time logs. SIGARCH Comput Archit News 38(1):143–154

40. Yuan D, Zheng J, Park S, Zhou Y, Savage S (2011) Improving software diagnosability via log enhancement. In: Proceedings of the Sixteenth International Conference on Architectural Support for Programming Languages and Operating Systems, ACM, New York, NY, USA, ASPLOS XVI, pp 3–14
41. Yuan D, Park S, Huang P, Liu Y, Lee MM, Tang X, Zhou Y, Savage S (2012) Be conservative: Enhancing failure diagnosis with proactive logging. In: Proceedings of the 10th USENIX Conference on Operating Systems Design and Implementation, USENIX Association, Berkeley, CA, USA, OSDI'12, pp 293–306
42. Yuan D, Park S, Zhou Y (2012) Characterizing logging practices in open-source software. In: Proceedings of the 34th International Conference on Software Engineering, IEEE Press, Piscataway, NJ, USA, ICSE '12, pp 102–112
43. Zhang S, Cohen I, Symons J, Fox A (2005) Ensembles of models for automated diagnosis of system performance problems. In: Proceedings of the 2005 International Conference on Dependable Systems and Networks, IEEE Computer Society, Washington, DC, USA, DSN '05, pp 644–653
44. Zhu J, He P, Fu Q, Zhang H, Lyu MR, Zhang D (2015) Learning to log: Helping developers make informed logging decisions. In: Proceedings of the 37th International Conference on Software Engineering - Volume 1, IEEE Press, Piscataway, NJ, USA, ICSE '15, pp 415–425
45. Zimmermann T, Weisgerber P, Diehl S, Zeller A (2004) Mining version histories to guide software changes. In: Proceedings of the 26th International Conference on Software Engineering, IEEE Computer Society, Washington, DC, USA, ICSE '04, pp 563–572