Optimizing Contemporary Application and Processes in Open Source Software

Mehdi Khosrow-Pour
Information Resources Management Association, USA

A volume in the Advances in Systems Analysis, Software Engineering, and High Performance Computing (ASASEHPC) Book Series
Chapter 3
Logging Analysis and Prediction in Open Source Java Project

Sangeeta Lal
Jaypee Institute of Information Technology, India

Neetu Sardana
Jaypee Institute of Information Technology, India

Ashish Sureka
Ashoka University, India

ABSTRACT
Log statements present in source code provide important information to the software developers because they are useful in various software development activities such as debugging, anomaly detection, and remote issue resolution. Most of the previous studies on logging analysis and prediction provide insights and results after analyzing only a few code constructs. In this chapter, the authors perform an in-depth, focused, and large-scale analysis of logging code constructs at two levels: the file level and catch-blocks level. They answer several research questions related to statistical and content analysis. Statistical and content analysis reveals the presence of differentiating properties among logged and nonlogged code constructs. Based on these findings, the authors propose a machine-learning-based model for catch-blocks logging prediction. The machine-learning-based model is found to be effective in catch-blocks logging prediction.

INTRODUCTION
Logging is an important software development practice that is used to record important program execution points in the source code. The recorded log generated from program execution provides important information to the software developers at the time of debugging. Fu et al. (2014) conducted a survey of Microsoft developers, asking them their opinion on source code logging. Results of the survey showed that 96 percent of the developers consider logging statements the primary source of information for problem diagnosis. In many scenarios, logging is the only information available to the software developers.

DOI: 10.4018/978-1-5225-5314-4.ch003
for debugging because the same execution environment is unavailable (which makes bug regeneration difficult) or the same user input is unavailable (because of security and privacy concerns) (Yuan et al., 2012). Yuan et al. (2012) showed in their characterization study that the bug reports consisting of logging statements get fixed 2.2 times faster compared to the bug reports not consisting of any logging statements. Logging statements are not only useful in debugging, but they are also useful in many other applications, such as anomaly detection (Fu et al., 2009), performance problem diagnosis (Nagaraj et al., 2012), and workload modeling (Sharma et al., 2011).

Logging statements are important, but they have an inherent cost and benefit tradeoff (Fu et al., 2014). A large number of logging statements can affect system performance because logging is an I/O-intensive activity. An experiment by Ding et al. (2015) and Sigelman et al. (2010) reveal that in the case of search engines, logging can increase average execution time of requests by of 16.3%. Similar to excess logging, less logging is also problematic. An insufficient number of logging statements can miss important debugging information and can lessen the benefits of logging. Hence, developers need to avoid both excessive and insufficient logging. However, previous research and studies show that developers often face difficulty in optimal logging, that is, identifying which code construct to log in the source code (Fu et al., 2014; Zhu et al., 2015). It happens because of lack of training and the domain experience required for optimal logging. For example, Shang et al. (2015) reported an incident of a user from a Hadoop project complaining about less logging of catch-blocks. Recently the software engineering research community has conducted studies to understand the logging practices of software developers in order to build tools and techniques to help with automated logging. The current studies provide limited characterization study or conduct analysis on fewer code constructs. There are gaps in previous studies, as they do not analyze all the code constructs in detail, which this study aims to fill.

The work presented in this chapter is the first large-scale, in-depth, and focused study of logged and nonlogged code constructs at multiple levels. High-level (source code files) and low-level (catch-blocks) analysis were conducted to identify relationships between code constructs and logging characteristics. Based on the finding of this multilevel analysis authors proposed a machine learning based model for log statement prediction for catch-blocks. A case study was performed on three large, open-source Java projects: Apache Tomcat (Apache Tomcat, n.d.), CloudStack (Apache CloudStack, n.d.), and Hadoop (Page, n.d.). Empirical analysis reveals several interesting insights about logged and nonlogged code constructs at both the levels. The machine learning based model give encouraging results for catch-blocks logging prediction on Java projects.

RELATED WORK

This section presents the closely related work and the novel research contributions of the study presented in this chapter in context to existing work. The authors categorize the related work in three dimensions: 1) improving source code logging, 2) uses of logging statements in other applications, and 3) applications of LDA in topic identification.

Improving Source Code Logging

Yuan et al. (2012) analyze source code and propose ErrorLog tool that logs all the generic exception patterns. However, logging all the generic exception can cause excess of log statements in the source
code. Fu et al. (2014) empirically analyzed logging practices of software developers on two industrial systems. They addressed three research questions in their study: first, finding code snippets that were logged frequently; second, identifying the distinguishing characteristics of logged and nonlogged code constructs; and third, building a tool for logging prediction. They analyzed 100 randomly chosen logging statements and identified the most frequently logged code construct types. They performed detailed analysis of return value check and exception snippets. They computed the logging ratio of each unique exception type and reported that the majority of the exception types falls in the range of a medium logging ratio (i.e., 10 percent to 90 percent). They analyzed 70 nonlogged catch-blocks and identified the main reasons for not inserting a logging statement in the catch-block. They reported the correlations among the presence of some specific keywords that affect the logging decision such as “delete,” “remove,” “get,” etc. The machine learning–based tool proposed by Fu et al., which used contextual information from the code, gave an F-score of 80 percent to 90 percent. This shows that contextual information can be an important factor when making logging decisions. This study extends the characterization study performed by Fu et al. on many dimensions. First, the study performed by Fu et al. presents results on the basis of manual analysis of only a few code constructs, whereas in this work the authors present their analysis using much larger code constructs. Second, the authors extended their study by answering many more research questions at two levels. Third, the authors analyzed open-source Java project, whereas they analyzed closed-source C# projects. Hence, the results in this chapter can be reproduced by the software engineering research community. Zhu et al. (2015) extended the study performed by Fu et al. by using more features for building the logging prediction model. However, their study also lack comprehensive analysis of the features used for building the logging prediction model.

Yuan et al.’s (2012b) work involved empirically analyzing modifications to log messages. They reported many interesting findings from their empirical analysis performed on four large open-source projects. Yuan et al. (2012b) reported that 18 percent of all the committed revisions modify logging code, and 26 percent of the time developers modify the verbosity level of the logging code as an afterthought. Forty-five percent and 27 percent of the time developers modify the text and variable of the log messages, respectively, to incorporate changes in the execution information. Based on these findings, they proposed a simple code clone-based technique to find inconsistent verbosity levels in the source code. In another study, Yuan et al. (2012c) propose model for enhancing the content of log statements. Chen et al. (2016) replicated the study performed by Yuan et al. (2012b) for Java projects and reported several differences in results as compared to the results reported by Yuan et al. (2012b). Kabinna et al.’s (2016) work on predicting the stability of logging statements using features from three different domains: context, developer, and content. In another study, Kabinna et al.’s (2016b) work on empirically analyzing migration of log libraries in Java projects. Li et al. (2016) worked on predicting verbosity level of log statement. In another study, Li et al. (2016b) worked on predicting just in time log changes. In contrast to these studies, our work focuses on finding distinguishing features of logged and non-logged code constructs and to predict logged code constructs. This book chapter is based on our previous published work (Lal et al., 2015, Lal & Sureka, 2016; Lal et al., 2016b). This work is found to be useful and has been extended for if-blocks logging prediction (Lal et al., 2016) and cross-project catch-blocks and if-blocks logging prediction (Lal et al., 2017a; Lal et al., 2017b).
Uses of Logging Statements in Other Applications

Logging statements have been found useful in various software development tasks (Mariani & Pastore, 2008; Nagaraj et al., 2012; Shang et al., 2015; Xu et al., 2009; Yuan et al., 2010). Shang et al. (2015) used logging statements present in a file to predict defects. Shang et al. (2015) proposed various product and process metrics using logging statements to predict post-release defects in software. Nagaraj et al. (2012) used good and bad logs of the system to detect performance issues. Nagaraj et al. (2012) also developed a tool, DISTALYZER, that helps developers find components responsible for poor system performance. Xu et al.’s (2009) work involved mining console logs from distributed systems at Google. They used logging information to find anomalies in the system. The authors verified anomalies were detected at the time when the system raised performance-related issues. They reported that performance issues are raised at the same time when anomalies are detected in system. Yuan et al. (2010) proposed a technique for finding the root cause of the failures by using logging information. They developed a tool, SherLog, that can use logs to find information about failed runs. SherLog can find important information about failures without requiring any re-execution of the code. All these studies focused on using log information in other applications such as finding root causes and performance issue detection. In contrast to these studies, the work described in this chapter focuses on the comparison between logged and nonlogged code constructs at two levels.

LDA Applications in Topic Identification

LDA is a popular topic modeling technique (Blei et al., 2003). It has been utilized widely in various software engineering applications to discover meaningful topics (Barua et al., 2012; Pagano & Maalej, 2013; Thomas et al., 2014; Tian et al., 2009). Tian et al. (2009) used LDA for software categorization. They proposed a system that can learn topic models from the identifier and comments present in the source code and can categorize software into one of the 43 programming languages such as C, C++, Java, PHP, Perl, etc. Thomas et al. (2014) used LDA topic models for software evolution analysis. Results reported by them show that topic models are effective in discovering actual code changes. Pagano et al. (2013) used LDA to study blogging behavior of committers and noncommitters. Results showed that committers’ blogs consist of topics related to features and domain concepts, and 15 percent of the time blogs consist of topics related to source code. In contrast, blogs of noncommitters consist of topics related to conferences, events, configuration and deployment. Barua et al. (2012) used LDA on a StackOverflow questions and answers dataset in order to discover the most popular topics among the developer community. Results showed that a wide variety of topics are present in the developer discussions. They also showed that topics related to Web and mobile application development are gaining popularity compared to other topics. All these previous studies show the effectiveness of LDA topic models in the software engineering applications, and hence the authors of this chapter choose LDA for topic analysis. However, in contrast to these studies, the authors used LDA for topic identification in logging and nonlogging code constructs. To the best of their knowledge, LDA has never been used for topic modeling in this context.
Table 1 shows four main research dimensions (RDs) and respective research questions (RQs) considered in this work. Following is a brief description of each RD and respective RQs:

- **RD1 - Statistical Analysis of Source Code Files**: In RD1, the authors answer three main research questions related to the statistical properties of logged and nonlogged files. Statistical analysis is important because it provides insights about the logged and nonlogged code constructs without looking at the semantics of the code. The first and second RQs compute the percentage of logged files and their average SLOCs. The third RQ computes the correlation between file SLOC and respective logging count.

- **RD2 - Statistical Analysis of Catch-Blocks**: In RD2, the authors answer five research questions related to the statistical properties of logged and nonlogged catch-blocks. The fourth research question compares the complexities of the try-blocks associated with logged and nonlogged catch-blocks to investigate whether complexities of try-blocks have any effect on the corresponding catch-block logging decision or not. The fifth and seventh RQs compute the logging ratio of all the exception types and the top 20 exception types in all three projects. The sixth RQ computes the contribution of an exception type in total catch-blocks and total logged exception types. The eighth RQ investigates whether logged and nonlogged catch-blocks can co-exist.

- **RD3 - Content-Based Analysis of Catch-Blocks**: In RD3, the authors use an LDA-based topic modeling technique on the contextual information present in the try-blocks associated with logged and nonlogged catch-blocks. They hypothesize that the contextual information present in the try-blocks can reveal important information for the corresponding catch-block logging.

- **RD4 - Logging Prediction Model for Catch-Blocks**: In RD4, the authors use finding of this empirical study and propose machine learning based model for logged catch-blocks prediction.
RESEARCH METHOD AND EXPERIMENTAL DATASET

This section presents the research methodology and experimental dataset details (refer to Figure 1). The research method consists of two phases: dataset selection and dataset preparation.

Dataset Selection Phase

In this phase, the authors selected open-source projects on which to conduct their experiments. Following are the list of properties and essential criteria which were taken into account while selecting the three open-source projects for the analysis:

1. **Type: Open Source**: The authors conducted their study on open-source software projects so that the work can be replicated and used for benchmarking and comparison.
2. **Programming Language: Java**: The authors selected a Java-based project for the study because Java is one of the most used programming languages (Kim, 2016; Krill, 2016).
3. **Logging Framework: Log4J**: The authors used Java projects utilizing the Log4J (Goers, n.d.) framework for logging. They targeted projects using the Log4J framework only because this is one of the widely used frameworks for Java logging.
4. **Number of Java Files: More Than 1,000**: The authors set this threshold so that they can draw statistically significant conclusions.
5. **Number of Catch-Blocks: More Than 1,000**: The authors set this threshold so that they can draw statistically significant conclusions.
Experimental Dataset Details

The authors selected three projects for their empirical study based on the criteria defined for the dataset selection phase: Apache Tomcat, CloudStack, and Hadoop. All three projects are long-lived Java projects with a development history of ≈7 to 17 years. Table 2 shows the SLOC of all three projects. SLOC are computed using the LocMetrics tool (LocMetrics, n.d.). Apache Tomcat, CloudStack and Hadoop have been previously used by the research community for logging and other studies (Kabinna et al., 2016; Lal & Sureka, 2016; Lal et al., 2016; Shang et al., 2015; Zimmermann et al., 2009). Following are the details of each project.

- **Apache Tomcat**: Apache Tomcat is open-source software developed under the umbrella of the Apache Software Foundation (Apache Tomcat, n.d). It is a Web server that implements many Java EE specifications like Java Servlet, Java EL, Java Sever Pages, and WebSocket. Logging is important in Apache Tomcat Web server; it has its own LogManager implementation; and it also supports private per-application logging configurations (Crossley, n.d.; Team, n.d.).

- **CloudStack**: CloudStack is open-source software developed by the Apache Software Foundation (Apache CloudStack, n.d). It provides public, private, and hybrid cloud solutions. It also provides a highly available and scalable Infrastructure as a Service (IaaS) cloud computing platform for deployment and management of networks of virtual machines. It provides support for many hypervisors such as VMware, KVM and Xen Cloud Platform (XCP). CloudStack provides large amounts of log entries, and for a CloudStack administrator investigating errors in the logs is an inevitable task (Kosinski, 2013).

- **Hadoop**: Hadoop is also developed by the Apache Software Foundation (Page, n.d). It is a framework that enables distributed processing of large datasets. It is scalable from a single server to multiple machines. The Apache Hadoop library is designed to detect and handle application-layer failures. Hadoop is one of the most widely used software platforms, and various tools have been developed to monitor the status of the Hadoop using generated logs (Shang et al., 2015; Rabkin & Katz, 2010).

Dataset Preparation Phase

In this step, the authors extract logging statements and target code constructs from the source code. Following are the details of the data preparation.

- **Files**: The authors extracted all the high-level (source code files) code constructs from the source code. They focused only on Java files in this work and removed other types of files such as CSS and XML. Table 2 shows statistics on the number of Java files extracted from each of the projects. For example, the Apache Tomcat project consists of 2,037 Java files, whereas the CloudStack project consists of 5,351 Java files. The authors extracted logging statements from each file (refer to Table 3). They marked a file as “logged” if it consisted of at least one logging statement; otherwise, it was marked as “non-logged.”
• **Catch-Blocks:** Next the authors extract all the catch-blocks from the source code. They extracted all the catch-blocks from the Java files using the Eclipse Java source code parsing library (Beaton, n.d). However, a single try-block can have multiple catch-blocks. In such cases the authors considered all catch-blocks belonging to a single try-block as a separate instance. Figure 2 shows an illustrative example of separate instance creation. The authors marked a catch-block as “logged” if it consisted of at least one logging statement. Table 2 shows that the experimental dataset consists of 3,325, 12,591, and 7,947 catch-blocks in Apache Tomcat, CloudStack, and Hadoop, respectively. It also shows that 27 percent, 26.15 percent, and 22 percent of the catch-blocks are logged in Apache Tomcat, Hadoop, and CloudStack, respectively.

• **Logging Lines:** All three projects used in the empirical study are Java and Log4J based projects. However, the authors observed several inconsistencies in logging statement formats and hence created 26 regular expressions to extract all the logging statements. The authors observed two semantically different types of logging: first, in which the logging level is explicitly mentioned (for example, Type 1 and Type 2 logging statements in Listing 1) and second, in which the logging level is not mentioned explicitly (for example, Type 4 and Type 5 in Listing 1). The authors also observed several inconsistencies in the uses of the log levels. For example, Listing 1 shows three different ways in which the log level “warn” is used in different datasets (refer to Type 1, Type 2, and Type 3 in Listing 1).

### STATISTICAL ANALYSIS ON HIGH-LEVEL CODE CONSTRUCTS

The following subsections present the work on characterizing high-level code constructs (source code files). The authors answer research questions related to the distribution and complexity of logged files. They also analyze correlations between the logging count of a file and it’s SLOC.

*Figure 2. Catch-block instance creation from try-blocks*
Logging Analysis and Prediction in Open Source Java Project

Listing 1. Example of logging statements taken from the dataset

```java
/*------------------Type 1: (Taken from Hadoop)-----------------*/
LOG.warn(AuthenticationToken ignored: + ex.getMessage());
/*------------------Type 2: (Taken from Hadoop)-----------------*/
logWarningWhenAuxServiceThrowExceptions(service, AuxServicesEvent.Type.APPLICATION_INIT, th);
/*------------------Type 3: (Taken from Apache Tomcat)--------*/
Logger.getLogger(getLoggerName(getHost(),url)).log(Level.WARNING,"Unable to determine web application context.xml " + docBase,);
/*------------------Type 4: (Taken from Apache Tomcat)--------*/
log("Error closing redirector: " + ioe.getMessage(),Project.MSG_ERR);
/*------------------Type 5: (Taken from Apache Tomcat)--------*/
project.log(wrong object reference + refId + - + pref.getClass());
```

Table 2. Experimental dataset details

<table>
<thead>
<tr>
<th>Project</th>
<th>Apache Tomcat</th>
<th>CloudStack</th>
<th>Hadoop</th>
</tr>
</thead>
<tbody>
<tr>
<td>Version</td>
<td>8.0.9</td>
<td>4.3.0</td>
<td>2.7.1</td>
</tr>
<tr>
<td>Logging Library</td>
<td>Log4J</td>
<td>Log4J</td>
<td>Log4J</td>
</tr>
<tr>
<td>Java File</td>
<td>2,037</td>
<td>5,351</td>
<td>6,332</td>
</tr>
<tr>
<td>SLOC</td>
<td>276,209</td>
<td>1,142,970</td>
<td>951,629</td>
</tr>
<tr>
<td>Log Line Count</td>
<td>2,703</td>
<td>10,428</td>
<td>10,108</td>
</tr>
<tr>
<td>Total Catch Blocks</td>
<td>3,325</td>
<td>12,591</td>
<td>7,947</td>
</tr>
<tr>
<td>Logged Catch Blocks</td>
<td>887 (27%)</td>
<td>2,790 (22.16%)</td>
<td>2,078 (26.15%)</td>
</tr>
<tr>
<td>Distinct Exception Types</td>
<td>120</td>
<td>163</td>
<td>265</td>
</tr>
</tbody>
</table>

**RQ 1:** Is the distribution of the logged files skewed?

The authors counted the number of files that consisted of at least one logging statement. Table 3 shows that only 17.9 percent, 14.9 percent, and 22.3 percent of files consisted of logging statements in Apache Tomcat, CloudStack, and Hadoop, respectively. This result shows that distribution of files containing logging statements is highly skewed, that is, less than 23 percent of files consist of logging statements. The authors believe that understanding the characteristics of source code for files that do not contain any logging statements can provide useful insights for logging prediction tools, as the tool does not need to predict logging in the files, given that there is no history of logging statements.

The distribution of files containing and not containing log statements is skewed as only ≈14 percent to 22.3 percent of files contain logging statements.

**RQ 2:** Do logged files have greater complexity compared to nonlogged files?

This subsection presents a comparison of the complexity of the logged and nonlogged files. The authors measured the complexity of a file using its SLOC. To compute SLOC, they removed all the blank lines, package statements, import statements, and comments from the file. They also removed lines containing only `'{' or '}'`. Table 3 shows the values of average SLOC of logged and nonlogged files for all three projects. The table also shows that for the Apache Tomcat project, the average SLOC value
Logging Analysis and Prediction in Open Source Java Project

Table 3. The count (%) of logged files in the total files. It also shows the average SLOC of logged and nonlogged files. LFC: Logged File Count; AS: Average SLOC; LF: Logged Files; NLF: Nonlogged Files

<table>
<thead>
<tr>
<th>Project</th>
<th>Total Files</th>
<th>LFC (%)</th>
<th>AS</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>LF</td>
<td>NLF</td>
</tr>
<tr>
<td>Apache Tomcat</td>
<td>2,037</td>
<td>365 (17.9%)</td>
<td>260.04 69.37</td>
</tr>
<tr>
<td>CloudStack</td>
<td>5,351</td>
<td>798 (14.9%)</td>
<td>290.81 159.87</td>
</tr>
</tbody>
</table>
| Hadoop    | 6,332       | 1414 (22.3%) | 254    75.51  

of logged and nonlogged files is 260.04 and 69.37. Results show a similar trend for other two projects, that is, the average SLOC of logged files is higher than that of nonlogged files. Average values provide useful statistics, but they lack significant details about the actual distribution. Hence the authors drew a box-and-whisker plot for the SLOC of logged and nonlogged files. The graph in Figure 3 shows that the median SLOC values of logged files is higher than that of nonlogged files for all three projects. For example, the median SLOC value for logged files in the CloudStack project is 114.5, whereas the median SLOC value for nonlogged files in the CloudStack project is 46.0. Figure 3 also shows a higher interquartile range for logged files, which shows a higher spread of SLOC values in logged files compared to that of nonlogged files. The results presented in this subsection lead to many more questions regarding the analysis of more complex metrics (such as object-oriented metrics; Thwin & Quah 2005) of files to get a deeper understanding about the relation between file complexity and logging.

Logged files have greater complexity and spread (measured using SLOC of a file) as compared to that of nonlogged files.

RQ 3: Is there a positive correlation between file complexity and log statement count?

The box-and-whiskers plot of the previous subsection shows that files with higher SLOC (i.e., higher complexity) are more likely to contain logging statements. Hence the authors hypothesize that there exists a positive correlation between file SLOC and its log statement count, that is, the higher the SLOC, the higher the log statement count of the file. To test this hypothesis the authors created a scatter plot

Figure 3. SLOC comparison of logged and nonlogged files
between file SLOC and the respective log statement count. Scatter plots are one of the simplest yet powerful methods to visualize correlations between two variables. The authors created two scatter plots: the first scatter plot was between the SLOC of all the files in the database and respective log statement count, and the second scatter plot was between the SLOC only of logged files and respective log statement counts. The authors also computed the Pearson correlation between file SLOC and log statement count (Welcome to Statistics, n.d.). They obtained correlation values of 0.58, 0.76, and 0.67 for Apache Tomcat, CloudStack, and Hadoop, respectively, which shows that a positive correlation exists between SLOC and log statement count of a file (refer to Figure 5). However, it is interesting to observe the correlation value between file SLOC and logging count decreases after the addition of nonlogged files (refer to Figure 4). The authors observed the presence of three (one in each project) very large, nonlogged files in all three projects. Figure 4 shows these three files, marked using a red circle. Manual analysis reveals that these three files are tool-generated files and hence do not consist of any log statements. Table 4 gives details about the files and the tool used to generate these files. The experimental results presented in this subsection show a positive correlation between file SLOC and log statement count. The authors believe that these findings can be utilized by logging prediction tools to predict logging in the files if they exceed some project-specific threshold of file SLOC.

A positive correlation exists between the SLOC of logged files and the logging count.

Figure 4. Scatter plot showing correlation between SLOC of the files and respective logging counts

Figure 5. Scatter plot showing correlation between SLOC of only logged files and respective logging counts
The following subsections work on characterizing low-level code constructs (catch-blocks). The authors answer research questions related to complexity, logging ratio distribution, and whether logged and nonlogged catch-blocks can exist together.

**RQ4:** Is the complexity of try-blocks associated with logged catch-blocks greater than that of nonlogged catch-blocks?

The authors compared the complexity of the try-blocks associated with logged and nonlogged catch-blocks. They wanted to analyze whether the complexity of a try-block acts a parameter when deciding to log corresponding catch-blocks or not. In this work, they considered three parameters to measure the complexity of a try-block: size of the try-block (SLOC count), operator count of the try-block and method call count.

### Comparing SLOC of Try-Blocks Associated With Logged and Nonlogged Catch-Blocks

The authors computed SLOC of the corresponding try-blocks associated with logged and nonlogged catch-blocks. They computed SLOC using the same method described in a previous section. Listing 2 shows an example of a try-block from the Apache Tomcat project. The SLOC value of the try-block shown in Listing 2 is 2. Figure 6 shows box-and-whisker plots revealing the dispersion and skewness in SLOC for try-blocks associated with logged and nonlogged catch-block across three projects. The graph in Figure 6 reveals that the median and the third-quartile values for logged catch-blocks are more than the corresponding values for nonlogged catch-blocks in Apache Tomcat and Hadoop. For example, the third quartile and median for logged catch-blocks in the Apache Tomcat project is 7.0 and 2.0, respectively, whereas the third quartile and median for nonlogged catch-blocks in the Apache Tomcat project is 2.0 and 1.0, respectively. However, for the CloudStack project, the authors observed that the third quartile for logged catch-blocks is higher than the third quartile for the nonlogged catch-blocks but the median value is smaller. The box plots in Figure 6 also reveal that the interquartile range (width of the box: Q3 – Q1) for logged catch-blocks is higher than those of nonlogged try-blocks, indicating a higher spread.
Logging Analysis and Prediction in Open Source Java Project

Listing 2. Example of a try-block taken from the Apache Tomcat project

```java
try{
    lc=new LoginContext(getLoginConfigName());
    lc.login();
} catch(LoginException e) {
    log.error(sm.getString(spnegoAuthenticator.serviceLoginFail),e);
    response.sendRedirect(HttpServletResponse.SC_INTERNAL_SERVER_ERROR);
    return false;
}
```

Try-LOC: 2
Operator Count: 7 ((()()() =)
Method Call Count: 2 (getLoginConfigName, login)
Catch Exception: LoginException

Figure 6. Comparison of SLOC of try-blocks associated with logged and nonlogged catch-blocks

Figure 7. Comparison of operator count of try-blocks associated with logged and nonlogged catch-blocks

Comparing Operator Count of Try-Blocks Associated With Logged and Nonlogged Catch-Blocks

Counting the total number of operators in a program has been widely used as a metric to measure the complexity of given source code. The Halstead metric for computing program complexity is based on counting the total and distinct numbers of operators and operands in the source code (Virtual Machinery, n.d.). The authors created a list of 19 arithmetic operators (=, *, +, −, %, !, (,), [], &, ?, :, >, <, |, ...
Logging Analysis and Prediction in Open Source Java Project

They counted the number of operators (from the list of 19) in the try-block linked to logged and nonlogged catch-blocks. The box plots in Figure 7 reveal that the third-quartile values for logged try-blocks (28, 47, and 18) are greater than the corresponding values (9, 45, and 12) for nonlogged try-blocks in Apache Tomcat, CloudStack, and Hadoop. The median values for Apache Tomcat indicate that logged try-blocks have greater complexity in terms of operator count. The authors observed that the median value for logged try-blocks and nonlogged try-blocks for the Hadoop project is the same. They believe that the lines-of-code metric is correlated to the number-of-operators metric, and hence they observe similar trends for both measures.

Comparing the Method Call Count of Try-Blocks Associated With Logged and Nonlogged Catch-Blocks

The Halstead complexity measure computes program complexity based on several factors, such as the number of distinct operators and operands, as well as the total number of operators and operands. Predefined library function and user-defined function calls are considered operators according to the Halstead complexity metric. A large number of methods (equivalent to operators) within a try-block increases both cognitive complexity and testing complexity. Listing 2 shows an example of a try-block with two executable statements, both of which are function calls (getLoginConfigName() and login()).

There can be two try-blocks with the same number of executable statements but a different number of function calls, and hence the complexity measure based on method call count is different from the complexity measure based on lines of code, as well as the complexity measure based on total number of operators. The authors computed the number of function calls for every try-block in the source code dataset. Figure 8 shows the box plots for the number of methods for the three projects in the experimental dataset. It reveals that the third-quartile value for the logged try-block is higher than the corresponding values for the nonlogged try-block for the Apache Tomcat and Hadoop projects. For example, the median and third-quartile value for Apache Tomcat is 2.0 and 6.0, respectively, for the logged try-block, which is higher than the median and third-quartile value of 1.0 and 2.0, respectively, for the nonlogged try-block. The authors observed that for the CloudStack project, the third-quartile value is the same for both logged and nonlogged try-blocks. Results give an indication that complexity of try-blocks can be use as a parameters for catch-blocks logging prediction.
Try-blocks associated with logged catch-blocks have greater complexity than that of nonlogged catch-blocks for the Apache Tomcat and Hadoop projects.

RQ 5: What is the logging ratio trend of the various exception types?

This subsection statistically analyzes the logging ratio (LRi) of distinct exception types in all three projects. The logging ratio of each exception type is computed using Equation 1 (refer to Table 5 for details about the acronyms used in this equation). The logging ratio metric is defined and used earlier by Fu et al. (2014) for analysis of exception types on C# projects. The logging ratio of an exception type shows the percentage of its logged catch-blocks (TLCBi) to its total number of catch-blocks (TCBi). For example, the “ChannelException” exception type in the Tomcat dataset has 25 catch-blocks, out of which 15 are logged. Hence, the logging ratio of ChannelException exception type i.e., LRi_ChannelException, is 60 percent. Figure 9 shows the histogram of the logging ratio of distinct exception types for all three projects. In Figure 9, the x-axis shows the range of the logging ratio (with an interval of 10 percent) and the y-axis shows the percentage of the distinct exception types falling in that range. On top of each bar of the histogram, the distinct exception types falling in that logging ratio range are plotted and counted. For example, Figure 9a shows that 47 exception types in the Apache Tomcat project have a logging ratio between 0 percent and 10 percent. Fu et al. (2014) reported in their study that the majority of exception types belong to either a very high (>=90%) or low (<=10%) logging ratio range. Although they computed the results on C# projects and results of a Java project can differ, the authors observed results similar to Fu et al. (2014) with Java projects.

\[
LR_i = \frac{|TLCB_i| * 100}{|TCB_i|}
\]

(1)

The majority of the exception types in the Java project belong to either a very high (>=90%) or very low (<=10%) logging ratio.

RQ 6: Is the exception type contribution the same in total catch-blocks and in total logged catch-blocks?

Figure 9. Logging ratio of all three projects
This subsection measures the contribution of each exception type in total catch-blocks as well as in total logged catch-blocks. The authors define two metrics: Exception Type Ratio (Catch Count) [ERCC] and Exception Type Ratio (Log Count) [ERLC] for the same; refer to Equation 2 and Equation 3 for details (refer to Table 5 to get details on the acronyms used in these equations). ERCC defines the percentage of contribution of a particular exception type in total catch-blocks whereas ERLC defines it for logged catch blocks. ERCC computes the percentage of total catch-blocks of an exception type (TCB) to total catch-blocks in the dataset (TCBDT), whereas the ERLC metric computes the percentage of total logged catch-blocks of an exception type (TLCB) to total logged catch-blocks in the dataset (TLCBDT). For example, for the Apache Tomcat project we have TCBDT = 3325 and TLCBDT = 887. Now for ‘ChannelException’ exception type we have TCBChannelException = 25 and TLCBChannelException = 15. Hence, the ERCC and ERLC values can be calculated.

Table 5. Various acronyms used in Equation 1, Equation 2, and Equation 3

<table>
<thead>
<tr>
<th>Variable Name</th>
<th>Acronym</th>
</tr>
</thead>
<tbody>
<tr>
<td>Total Catch-Blocks in the Dataset</td>
<td>TCB\text{\textsubscript{DT}}</td>
</tr>
<tr>
<td>Total Logged Catch-Blocks in the Dataset</td>
<td>TLCB\text{\textsubscript{DT}}</td>
</tr>
<tr>
<td>Total Catch-Block of \textit{i}\textsuperscript{th} Exception Type</td>
<td>TCB\textsuperscript{i}</td>
</tr>
<tr>
<td>Total Logged Catch-Blocks of \textit{i}\textsuperscript{th} Exception Type</td>
<td>TLCB\textsuperscript{i}</td>
</tr>
<tr>
<td>Exception Type Ratio (Catch Count) of \textit{i}\textsuperscript{th} Exception Type</td>
<td>ERCC\textsuperscript{i}</td>
</tr>
<tr>
<td>Exception Type Ratio (Log Count) of \textit{i}\textsuperscript{th} Exception Type</td>
<td>ERLC\textsuperscript{i}</td>
</tr>
<tr>
<td>Logging Ratio of \textit{i}\textsuperscript{th} Exception Type</td>
<td>LR\textsuperscript{i}</td>
</tr>
</tbody>
</table>

Figure 10 shows the histogram of ERCC metrics for all three projects. In Figure 10, the x-axis shows the ERCC range and the y-axis shows the sum of ERCC values of all the exception types falling in that ERCC range, that is, all the exception types falling in a particular group give an ERCC range and their ERCC value can be summed. On top of each bar is a count of unique exception types falling in that ERCC range. For example, Figure 10a shows that for the Apache Tomcat project 116 exception types have an ERCC metric value between 0 and 5 and sum of their ERCC values is 35, that is, 116 exception types together constitute 35 percent of total catch-blocks. Figure 10a also shows that two exception types in the Apache Tomcat project have ERCC values between 20 and 25 and together they constitute 42 percent of total catch-blocks. Figure 11 shows the histogram for ERLC metrics for all three projects. In Figure 11, the x-axis shows the ERCC range and the y-axis shows the sum of ERLC values of all the exception types falling in that ERCC range. The x-axis is the same in both graphs so the contribution of the same exception type in total catch-blocks (using the ERCC metric) and in total logged catch-blocks (using the ERLC metric) can be compared. Figure 11a shows that 116 exception types together constitute 38 percent of all logged catch-blocks, whereas these same 116 exception types constitute only 35 percent of total catch-blocks (refer to Figure 10a).
Figure 10b and Figure 11b show an interesting finding about four exception types (marked by arrows in the Figure) from the CloudStack project. These four exception types—ADBException, AxisFault, IllegalArgumentException, and XMLStreamException—have very large ERCC values (i.e., very large contribution in total catch-blocks), but very low ERLC values (i.e., much smaller contribution in the total logged catch-blocks). For example, the exception type “XMLStream” has 1,952 occurrences in the CloudStack project, but none of these occurrences are logged. Detection of such exception types can be beneficial for logging prediction tools, as the tool can learn the exception types that have such drastic differences in ERCC and ERLC values.

\[
\text{ERCC}_i = \frac{|TCB_i| \times 100}{|TCB_{DT}|} \tag{2}
\]

\[
\text{ERLC}_i = \frac{|TLCB_i| \times 100}{|TLCB_{DT}|} \tag{3}
\]

Some exception types in the CloudStack project have very high ERCC values (i.e., very large contribution in total catch-blocks) but very low ERLC values (i.e., much less contribution in the total logged catch-blocks).

**RQ 7:** Are the top 20 exception types and their respective logging ratios the same in all three projects?

This section presents an analysis of the logging ratio of the top 20 most frequent exception types in all three projects. The authors plotted a pie chart showing the contribution of the top 20 most frequent exception types in total catch-blocks. Figure 12 shows that the top 20 most frequent exception types contribute to ≈80 percent to 88 percent of the total catch-blocks for all three projects. Hence, analyzing the top 20 exception types can be crucial for the logging prediction tools, as ≈80 percent of the time the tool will be making a prediction for one of these top 20 exception types. In addition to this, if a similar trend exists regarding the logging ratio of the top 20 exception types across the projects, then it can be beneficial for cross-project logging prediction.

**Figure 10. ERCC metric value for all three projects**
The authors wanted to answer two interesting research questions about the top 20 exception types: Are these top 20 exception types the same across all the projects, and do the top 20 exception types show common trends for logging ratios in all three projects? To answer these research questions, the authors computed the top 20 exception types, as well as their respective logging ratios (using Equation 1) for all three projects. The answer to the first question is “No.” Results show that only 6 exception types are common among the three projects in the top 20 exception type list. Table 7 shows details of these six common exception types (Exception, IQException, Throwable, InterruptedException, IllegalStateException, and IllegalArgumentException) in all the three projects. The Throwable class is the superclass of all errors and exceptions in the Java language. The Exception class and its subclasses are a form of Throwable that indicates conditions that a reasonable application might want to catch. Throwable and Exception are higher-level classes (Exception extends Throwable, which extends the root of the class hierarchy Object), with several subclasses defining specific exception types; hence, they are common. The authors believe that classes like InterruptedException are common because Apache Tomcat, CloudStack, and Hadoop extensively use multithreading, and InterruptedException is thrown when a thread is waiting, sleeping, or otherwise occupied and the thread is interrupted. The authors compared logging ratios of six common exceptions in all three projects. Table 7 shows no specific trend for logging ratios across the three projects. For example, the exception type “Exception” has a low logging ratio for the Apache Tomcat and Hadoop projects (i.e., 37.25 percent and 27.72 percent), whereas for the CloudStack project, it has a high logging ratio (i.e., 66.81 percent). The authors observed similar trend for other exception types. Thus, the answer to the second research question is also “No.” This indicates that the logging ratio of an exception type is project specific, and hence a cross-project defect-prediction technique might need more sophisticated features than logging ratio.
Logging Analysis and Prediction in Open Source Java Project

Figure 12. Pie chart of top 20 exception types, showing percentage in total contribution

Table 7. Logging ratio details of 6 common exceptions in the top 20 exception type list of the three projects

<table>
<thead>
<tr>
<th>Exception Type</th>
<th>Apache Tomcat</th>
<th>CloudStack</th>
<th>Hadoop</th>
</tr>
</thead>
<tbody>
<tr>
<td>Exception</td>
<td>37.25%</td>
<td>66.81%</td>
<td>27.72%</td>
</tr>
<tr>
<td>IOException</td>
<td>27.41%</td>
<td>54.69%</td>
<td>36.69%</td>
</tr>
<tr>
<td>Throwable</td>
<td>45.66%</td>
<td>72.24%</td>
<td>53.05%</td>
</tr>
<tr>
<td>InterruptedException</td>
<td>6.12%</td>
<td>25.15%</td>
<td>23.28%</td>
</tr>
<tr>
<td>ClassNotFoundException</td>
<td>32.14%</td>
<td>4.48%</td>
<td>6.49%</td>
</tr>
<tr>
<td>IllegalArgumentException</td>
<td>23.19%</td>
<td>0.96%</td>
<td>16.67%</td>
</tr>
</tbody>
</table>

The most frequent exception types, as well as their respective logging ratios, are project specific.

RQ 8: Do logged and nonlogged catch-blocks coexist together?

The Java programming language allows associating multiple catch-blocks (each with a different exception type) to a single try-block. In this subsection, the authors’ aim is to investigate whether a single try-block can have both logged and nonlogged catch-blocks or not. This research question is important to answer, as many times catch-block logging prediction tools use features from try-blocks. If the frequency of such try-blocks is very high, then it can affect the performance of such machine learning–based catch-block logging prediction tools. To answer this research question, the authors computed the count of try-blocks with both logged and nonlogged catch-blocks in all three projects. Table 8 shows that a very small percentage (i.e., ≈0.33 percent to 1.4 percent) of total try-blocks has both logged and nonlogged catch-blocks.

A very small percentage of try-blocks have both logged and nonlogged catch-blocks.

Content-Based Analysis of Low-Level Code Constructs

This section presents the experimental results of content-based analysis of low-level (catch-blocks) logged and nonlogged code constructs. The authors applied LDA for content analysis. LDA is a popular topic modeling technique and has been used widely in the past for topic identification in the source code and in many other research areas (Thomas et al., 2014; Maskeri et al., 2008; Pagano & Maalej, 2013).
following subsections describe the steps of the LDA model creation, results of LDA topic modeling, and the authors’ observations from the obtained results.

**Preprocessing Steps for LDA Analysis**

The preprocessing steps for LDA model creation are as follows:

1. To identify the topics present in the logged and nonlogged catch-blocks, the authors analyzed the contents of the try-blocks associated with logged and nonlogged catch-blocks. They created corpus consisting of the content of try-blocks associated with logged and nonlogged catch-blocks.
2. The authors performed prepossessing and removed all the English stop words, special characters, and operators. They removed English stop words such as “is,” “the,” and “of” from the analysis because they were mainly interested in identifying the core functionality of the code constructs that leads to logging. The authors then applied stemming on the obtained corpus. Stemming is useful in reducing inflected words to the same root words and hence helps in reducing the corpus size. The authors used the Python NLTK library for stop word removal and stemming (Natural Language Toolkit, n.d.).
3. The authors believe that words that occur in almost all the documents or that occur in very few documents may not be helpful in retrieving useful topics. Hence they removed all the words that occurred in 80 percent of the documents and in less than 2 percent of the documents.
4. The authors performed LDA for 10,000 runs because LDA gives better results when the number of iterations is increased. Previous studies in software engineering research have also used the same threshold value for LDA (Thomas et al., 2014).
5. The authors set the number of topics parameter for the LDA algorithm as 10.
6. The authors used a default value of other LDA parameters in the Python LDA library (Gensim, n.d.).

**RQ 9:** Do try-blocks associated with logged and nonlogged catch-blocks have different topics?

Table 9 shows the result obtained by LDA topic modeling on try-blocks associated with both logged and nonlogged catch-blocks. From this table, the authors observed that topics listed under try-blocks associated with logged and nonlogged catch-blocks are different. Hence they randomly picked some of the topics from the logged and nonlogged category and analyzed the differences in the associated code blocks. The authors drew the following interesting observations from this analysis:

<table>
<thead>
<tr>
<th>Project</th>
<th>Apache Tomcat</th>
<th>CloudStack</th>
<th>Hadoop</th>
</tr>
</thead>
<tbody>
<tr>
<td>Unique try-blocks</td>
<td>2,914</td>
<td>9,899</td>
<td>7,171</td>
</tr>
<tr>
<td>Try-block with more than one catch-block</td>
<td>254</td>
<td>1,002</td>
<td>653</td>
</tr>
<tr>
<td>Try-block with mix of logged and nonlogged catch-blocks</td>
<td>41 (1.40%)</td>
<td>31 (0.31%)</td>
<td>77 (1.07%)</td>
</tr>
</tbody>
</table>

Table 8. Details of try-blocks with multiple catch-blocks
They observed the “thread sleep” topic in the Apache Tomcat project. This topic is mentioned in the nonlogged catch-block category. The authors further analyzed occurrences of “thread sleep” in the Apache Tomcat project. They observed that in 84 occurrences of “thread sleep,” it occurred 71 times in try-blocks associated with nonlogged catch-blocks.

The authors observed the presence of a topic related to “socket” in both try-blocks associated with logged and nonlogged catch-blocks. They analyzed all 43 try-blocks consisting of socket and wrapper words and found that in try-blocks associated with logged catch-blocks, the socket wrapper is mostly used for close or error functions, whereas for try-blocks associated with nonlogged catch-blocks, the socket function is used for timeout operations. LDA is able to detect this difference, as shown in the Apache Tomcat project regarding topics 4 (logged catch-blocks) and 3 (nonlogged catch-blocks).

The authors analyzed the “result stub (topic 1)” topic from the CloudStack project. They found 161 occurrences of try-blocks consisting of both words. They also noticed that catch-blocks associated with all 161 try-blocks are nonlogged. LDA is able to detect this because the “request stub” topic is not present in logged catch-blocks.

The contextual information present in the try-blocks provides important information for the associated catch-block logging.

### Table 9. Topics discovered in try-blocks associated with logged and nonlogged catch-blocks

<table>
<thead>
<tr>
<th>Project</th>
<th>Logged Catch-Block</th>
<th>Nonlogged Catch-Block</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Topic</td>
<td>Word</td>
</tr>
<tr>
<td>Apache Tomcat</td>
<td>1. channel file</td>
<td>1. thread sleep</td>
</tr>
<tr>
<td></td>
<td>2. method param</td>
<td>3. socket status</td>
</tr>
<tr>
<td></td>
<td>3. context log</td>
<td></td>
</tr>
<tr>
<td></td>
<td>4. socket status</td>
<td>4. thread sleep</td>
</tr>
<tr>
<td></td>
<td></td>
<td>2. channel read</td>
</tr>
<tr>
<td></td>
<td></td>
<td>3. context get, null,</td>
</tr>
<tr>
<td></td>
<td></td>
<td>host</td>
</tr>
<tr>
<td></td>
<td></td>
<td>4. socket, statu, get,</td>
</tr>
<tr>
<td></td>
<td></td>
<td>wrapper</td>
</tr>
<tr>
<td>CloudStack</td>
<td>1. byte key</td>
<td>1. result stub</td>
</tr>
<tr>
<td></td>
<td>2. response value</td>
<td>2. result stub</td>
</tr>
<tr>
<td></td>
<td>3. network</td>
<td></td>
</tr>
<tr>
<td></td>
<td>4. vm host</td>
<td></td>
</tr>
<tr>
<td>Hadoop</td>
<td>1. key id 2. assert</td>
<td>1. user token</td>
</tr>
<tr>
<td></td>
<td>3. job conf 4. rm token</td>
<td>3. get response</td>
</tr>
<tr>
<td></td>
<td></td>
<td>4. file path</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
LOGGING PREDICTION MODEL FOR CATCH-BLOCKS

Using finding from our empirical analysis we propose a machine learning based catch-blocks logging prediction model, LogOpt (Lal & Sureka, 2016; Lal et al., 2016b). Based on this study we extract 46 distinguishing features for catch-blocks logging prediction (refer to Table 10 for details). These features have three properties: Type, Domain, and Class. Type of a feature specifies whether a feature is: textual, numeric or Boolean. Textual features can take any textual value. Numeric features can take any positive numeric value. Boolean features can take value either 0 or 1. Domain of the feature specifies part of the source code from where the feature is extracted. We identified three domains: try/catch, method_bt, and other. If a feature extracted from try/catch-block it will have domain ‘try/catch’. If a feature extracted from the first line of the containing methods to the previous line of try-block associated with target catch-block, it will have domain ‘method_bt’. If a feature extracted from some other part of source code, it will have domain ‘other’. Features can belong to positive class feature of negative class. Positive class features are beneficial in predicting logged catch-blocks whereas negative class features are beneficial in predicting nonlogged catch-blocks. We use total 46 features for catch-blocks logging prediction model building (refer to Table 10)

LogOpt Model Building

Using the 46 features, the authors propose LogOpt model for catch-blocks logging prediction. LogOpt is a machine learning based model. For LogOpt model training, the authors, first extract all the catch-blocks from the dataset and label them as ‘logged’ or ‘nonlogged’. A catch-block is marked as logged if it consists of at least one log statement; otherwise, it is marked as ‘nonlogged’. Second, the authors extract all the 46 features (textual, numeric, and Boolean) from the all the instances. Uses of textual features directly for machine learning model building can increase in model complexity. Hence, in the third step authors applied feature preprocessing techniques to clean the textual features. Authors applied camel case conversion, lower case, stop word removal, stemming, and tf-idf (Han et al., 2011) conversion. Fourth, author combine tf-idf representation of textual features with Boolean and numeric features and create final feature vector. Authors, then train machine learning algorithms such as Radom Forest (RF), J48, Support Vector Machine (SVM), on the final feature vector to create the LogOpt model. This model is then used for logging prediction on new instances.

LogOpt Model Evaluation

Authors evaluate performance of LogOpt model on all the three project (Apache Tomcat, CloudStack, and Hadoop). For testing the performance of LogOpt model, authors divided the dataset into two parts in a ratio of 70:30 using stratified random sampling (Han et al., 2011). 70% of the dataset is used for training and 30% of the dataset is used for testing. Since, dataset sampling can lead to biases in the result the authors created 10 such random samples and reported average results. Authors evaluated performance of the model using several machine learning classifiers (RF, SVM, J48). SVM classifier performs the best and give the highest F1-score of 76.79% (Apache Tomcat), 84.32% (CloudStack), and 67.16% (Hadoop) (Lal & Sureka, 2016; Lal et al., 2016b).
Table 10. Features used for building catch-block logging prediction model. Class: Positive (P), Negative (N). Domain: Try/Catch (T), Method_bt (M), Other (O).

<table>
<thead>
<tr>
<th>Type of Feature</th>
<th>Catch-Block Features (Class, Domain)</th>
<th>Explanation</th>
</tr>
</thead>
<tbody>
<tr>
<td>Total Features =</td>
<td>Textual (14) + Numeric (13)+ Boolean(19) =46 Feature</td>
<td></td>
</tr>
</tbody>
</table>
CONCLUSION AND FUTURE WORK

Source code logging is an important software development practice, and tools and techniques that can help software developers make optimal and strategic logging decisions can be beneficial. Analysis of logged and nonlogged code constructs can provide useful insights to improve current logging prediction tools. In this chapter, the authors performed statistical and content-based analysis of source code files and catch-blocks from three large open-source Java projects. They answered several research questions in this chapter. Following are the main research findings of this work:

- Fewer files consist of logging statements.
- Source code files with logging statements have a much larger average SLOC compared to those without logging statements.
- There is a positive correlation between the SLOC of logged files and their respective log statement counts.
- Try-blocks associated with logged catch-blocks have greater complexity than that of nonlogged catch-blocks for the Apache Tomcat and Hadoop projects.
- Some exception types contribute greatly to total catch-blocks, whereas there is little or no contribution in total logged catch-blocks.
- The logging ratio of an exception type is project specific.
- The LDA-based topic modeling technique is effective in discovering topics of logged and nonlogged code constructs.

Authors proposed a machine leaning based model proposed for catch-blocks logging prediction. Machine learning based model is found to be effective in catch-blocks logging prediction and give the highest F1-score of 84.32% on CloudStack project.

The authors think that this work provides a future direction for three lines of work: statistical analysis, content-based analysis, and machine learning based logging prediction model. Statistical analysis provides the ability to explore more deeply the features of logged code constructs. In this work, the authors analyzed a complexity metric (SLOC, operator count, etc.) with respect to logged and nonlogged code constructs. However, many other source code metrics, such as inheritance depth, and object-oriented metrics need to be evaluated for deeper analysis of logged and nonlogged files. Content-based research needs more exploration in terms of the topics present in the logged and nonlogged code constructs. In this work, the authors used LDA for topic modeling, and the initial results are encouraging. However, deeper analysis of code constructs with respect to multiple semantic techniques such as LDA and Latent Semantic Indexing (LSI) is required for in-depth analysis of the topics present in logged and nonlogged code constructs. In this work, authors propose a machine learning model based on static features from the source code for catch-blocks logging prediction. The proposed model can be extended for other type of code constructs such as if-blocks, while-loop, switch-case.
Logging Analysis and Prediction in Open Source Java Project

THREATS TO VALIDITY

- **Number and Type of Project:** The authors selected Apache Tomcat, CloudStack, and Hadoop projects for the study. All three projects are open-source, Java-based projects. Other types of projects, such as closed source, or projects written in other languages (e.g., C#, Python) need to be evaluated. Overall, the authors cannot draw any general conclusion that is applicable to all software logging. They believe that this study provides insight about logging practices of open-source, Java-based projects.

- **Quality of Ground Truth:** The authors assumed that logging statements inserted by software developers of Apache Tomcat, CloudStack, and Hadoop project are optimal. There is the possibility of errors or nonoptimal logging in the code by the developers, which can affect the results of the study. However, all three projects are long lived and are actively maintained; hence it is safe to assume that most of the code constructs have good (if not optimal) logging. The authors used 26 regular expressions to extract the logging statements from the source code. Manual analysis reveals that all the logging statements were extracted (to the best of the authors’ knowledge). However, there is still a possibility that the regular expressions missed some types of logging statements in the source code.

- **Machine Learning Model Evaluation:** At the time of evaluating the performance of LogOpt model, authors removed the three tool generated files from the dataset. Authors believe that using data from tool generated files for training as well as for prediction can cause bias in the performance of model.

REFERENCES


