# Analysis and Prediction of Log Statement in Open Source Java Projects

Sangeeta Lal[1], Neetu Sardana[1], and Ashish Sureka[2]

[1] Jaypee Institute of Information Technology (JIIT), India
sangeeta@jiit.ac.in,neetu.sardana@jiit.ac.in
[2] ABB Corporate Research, India
ashish.sureka@in.abb.com

**Abstract.** Log statements present in the source code provide important information about program execution which is helpful in several software development activities such as remote issue resolution, debugging, and load testing. However, log statements have a trade-off between cost and benefit and hence it is important to optimize the number of log statements in the source code. Previous studies show that optimizing log statements in the source code is a non-trivial activity and software developers often face difficulty in it. Several previous studies empirically analyze log statement and propose models for logging prediction. However, there are gaps in the literature which our study aims to address.
In this work, we aim to build tools and techniques which can help developers in optimizing the number of log statements in the source code. In order to do so, we first start by performing a survey of software developers from open-source projects. We then analyze properties of logged and non-logged code constructs at multiple levels. Using inputs from our empirical analysis we then propose machine learning based models for within-project catch-blocks and if-blocks logging prediction. We extend this study for cross-project logging prediction using ensemble based machine learning techniques. Our initial results are encouraging and show the possibility of making a robust machine learning based logging prediction tool for Java projects.

**Keywords:** Debugging, Logging, Machine Learning, Source Code Analysis, Tracing

## 1 Problem statement and its importance

Logging is a software development practice that is performed by inserting log statements in the source code. Log statements are used to record execution information about the program. For example, Listing 1.1 shows a try/catch block taken from the Tomcat project. This catch-block consists of a log statement which records the information about *login failure*. This recorded log can be used by the software developers at the time of debugging, in a case of login failure.

Log statements are customizable and provide feature for verbosity level modification. Hence, logging is a better alternative to commonly used *print* statements for debugging. In a survey performed by Fu et al. [4], 96% of the survey respondents agreed that log statements are important in system maintenance and development. In addition to debugging [20], logging is useful several other software development activities such as remote issue resolution [1] and load testing [6, 7].

Listing 1.1: Example of a try/catch-block code snippet from the Tomcat project

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

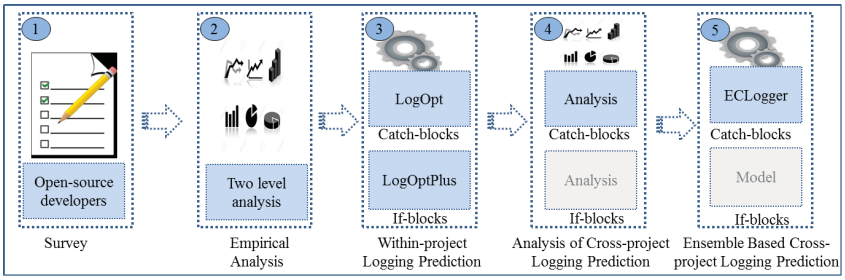


Fig. 1: Illustration of five step study objective to improve logging in open-source Java projects

Log statements are beneficial but they have trade-off between cost and benefit [4, 22]. Logging is an I/O intensive activity and hence excess of log statements in the source code can cause performance overhead to the system. For example, excessive or unnecessary logging is considered as one of the top 5 reasons for performance bottlenecks in .NET applications [5]. In addition, excess of log statements can generate too many trivial logs and can make debugging harder. Similar to excess logging sparse logging is also not good. Sparse logging can miss important debugging information and can potentially make debugging harder. Shang et al. [17] reported a case of a user who was complaining about less logging of catch-blocks in Hadoop project. Hence, it is important to optimize the number of log statements in the source code.

Previous research shows that optimizing log statements in the source code is a non-trivial task and software developers often face difficulty in it [4, 22]. In a survey by Zhu et al. [22], 68% of the participants said that they face logging

difficulties. Optimized source code logging is particularly challenging for OSS because source code logging is often based on domain knowledge & experience of software developers. Contribution to OSS is voluntary and hence, such domain knowledge is seldom documented in the real world. A study by Levesque [13] shows that OSS lack documentation. In addition to this, finding mentors in OSS is a challenging task. Due to which source code becomes challenging for new OSS developers. We believe that automated tools and techniques that can aid OSS developers in logging can be beneficial in improving both OSS product and process. New OSS developers can get guidance about source code logging at the times of coding which will improve the quality of OSS product. Similarly, automated logging checking tool can be run before every commit or release which can generate warnings about code constructs that need to be logged and hence will improve the OSS process.

Several recent studies work on empirically analyzing and predicting log statements in the source code for C# projects [4, 22]. These studies show that logged and non-logged code constructs have differentiating properties for C# projects. For example, Fu et al. [4] reported that catch-blocks consisting of *'FileNotFoundException'* and the corresponding try-block consisting of the keyword *'delete'* are often not logged. However, there does not exists any large scale & focused study which analyze properties of logged and non-logged code constructs for Java projects. Previous studies show that machine learning based models can be beneficial in predicting logged code constructs and propose machine learning based logging prediction models for C# projects [4, 22]. At present, there is no machine learning based model for logging prediction on Java projects. In addition to this, cross-project logging prediction is relatively unexplored yet. Logging prediction models for Java code constructs can be beneficial to the software developers because Java is one of the widely used programing language [2]. Therefore, my dissertation explore following:

"*Analyzing logged and non-logged code constructs to build machine learning based logging prediction model for Java project*".

To address the above, this work is broadly divided into five parts i.e, Research Objectives (RO's), also illustrated in Figure 1:

- **RO 1:** Analysis of open-source developers opinion towards source code logging on Java projects.
- **RO 2:** Perform a large-scale and in-depth study of logged and non-logged code constructs for Java projects.
- **RO 3:** Build a machine learning based within-project logging prediction model for Java projects.
- **RO 4:** Analysis of machine learning based logging prediction models for cross-project logging prediction for Java projects.
- **RO 5:** Build an ensemble of classifier based cross-project logging prediction model for Java projects.

Literature shows that currently there is no in-depth study on logged code constructs analysis or prediction for Java projects (refer to section 2). To address this gap, in RO 1, we first perform a survey of open-source developers to identify their opinion about logging on Java projects. The outcome of the survey reveals that software developers face difficulty in source code logging in Java project. We believe that automated tools and techniques that can aid software developers in source code logging can be beneficial. Hence, we propose machine learning based logging prediction models. The main challenge in machine learning based logging prediction is the identification of good features for training the machine learning model. To identify these features, in RO 2, we perform a large scale, focused and two-level empirical analysis of logged and non-logged Java code constructs. A code construct is consider as logged if it consists of at least one log statement otherwise it is considered as non-logged. This study reveal that there are several distinguishing features of logged and non-logged code constructs. Using some of the findings of our empirical study, in RO 3, we propose machine learning based logging prediction model for catch-blocks and if-blocks logging prediction model. We start by catch-blocks and if-blocks because they are one of the most frequent logged code snippets [14]. Empirical study of these machine learning based models show that they are effective in within-project (i.e., when training and testing data is from the same project) logging prediction. The performance of these models was not tested for cross-project (i.e., when training data and testing data are from the different projects) logging prediction. Hence, in RO 4, we analyze performance of several classifiers for cross-project logging prediction. The analysis reveals that different classifiers are complementary to each other for the task of cross-project catch-blocks logging prediction. Using this finding, in RO 5, we propose an ensemble of classifier based approach for cross-project catch-blocks logging prediction. This thesis provides a comprehensive study of source code logging with respect to Java projects. Following are the main scientific and technical contributions made by this thesis:

**Scientific Contribution:** In the literature, there are studies on logged and non-logged code constructs only for C\C++\C# projects. This thesis fills this significant research gap and perform in-depth study of logging in Java code constructs. The survey of open source developers indicates a need towards logging prediction tool for Java code constructs. The empirical study reveals the presence of distinguishing properties among logged and non-logged Java code constructs. This study propose machine learning based logging prediction models for catch-blocks and if-blocks logging. Analysis of these models on two Java projects reveals that machine learning based logging prediction models are effective in logged code construct prediction for Java projects. This study performs comprehensive analysis of several machine learning classifiers for cross-project catch-blocks logging prediction and show the effectiveness of using ensemble of classifier based approach for cross-project catch-blocks logging prediction.

**Technical Contribution:** This thesis have proposed several conceptual models

for logging prediction Java project. ***LogOpt*** [12] and ***LogOptPlus*** [11] models proposed in this thesis are useful for within-project catch-blocks and if-blocks logging prediction for Java projects. ***ECLogger*** [9] models proposed in this thesis is useful in improving the performance of cross-project catch-blocks logging prediction performance of Java projects.

## 2 Related Work

### 2.1 Empirical Analysis of Log statements

Fu et al. [4] analyze 100 randomly selected log statements from two closed-source systems (written in C#). They categories the log statements in five categories: assertion-check, return-value-check, exception, execution points, logic-branch and observing-point logging. They further perform detailed study of 70 non-logged catch-blocks and find reasons of not-logging. Several other studies analyze different aspect of logs. Yuan et al. [19] analyze 250 randomly sampled bug reports from five large C/C++ projects to find efficacy of logs for debugging. They report that most of the failures can not be diagnosed using existing logs. In another study, Yuan et al. [20] analyze four open source projects written in C/C++. They investigate amount of effort that software developers spend on modifying existing log statements. Shang et al. [16] analyze logs and report that log statements are often modified without considering the underlying application that use logs, and hence, log modification often results in break of functionality in log processing applications.

As the literature shows, currently there is no large scale & in-depth study of logged and non-logged code constructs for Java projects. Hence, as part of this work, we first perform a survey of software developers from open-source Java projects. In order to know their opinion about logging in Java projects (refer to section 3.1). We then perform, a two level, in-depth, and large scale study of logged and non-logged code constructs for open-source Java projects (refer to section 3.2).

### 2.2 Logged Code Construct Prediction

Several prior studies focus on improving logging statements in the code. Yuan et al. [21] propose *LogEnhancer* tool to enhance the content of log statements by adding causally-related information to improve failure diagnosis. Enhancing content of log statements is important but it does not consider cases of code snippets which are not logged. Yuan et al. [19] propose *ErrorLog* tool to log all the generic exceptions in C# code. This study shows the first steps towards automated logging but logging all the generic exceptions can cause too many log statements. Fu et al. [4] show the uses of machine learning based model for logging prediction. Zhu et al. [22] extended the study performed by Fu et al. and propose, *LogAdviosr*, an improved machine learning based model for logging prediction on C# projects. While there have been studies on machine

learning based logging prediction for C# projects, there has been no research study which focuses on logging prediction on Java projects. Hence, in this work we propose two machine learning based models for catch-blocks and if-blocks logging prediction for Java projects (refer to section 3.3).

### 2.3 Cross-project logging prediction

Cross-project logging prediction is important, as in real world there are many new or small projects. These projects do not have the sufficient amount of prior data to train the machine learning model. In such cases, it is important to train the prediction model from other large and long live projects. Cross-project prediction have been explored in detail for other software development activities such as defect prediction [15] and build co-change [18] prediction. However, cross-project logging prediction is relatively unexplored yet. Zhu et al. [22] perform an experiment related to cross-project logging prediction. They report that performance of cross-project logging prediction is degraded considerably as compared to within-project logging prediction. In this work, we perform in-depth analysis of several machine learning algorithms for cross-project catch-blocks logging prediction (refer to section 3.4) and also propose an ensemble of classifier based approach to improve cross-project logging prediction performance (refer to section 3.5).

## 3 Approach

In the following subsections, we give brief detail about each RO. We discuss motivation, research questions (RQ's), and approach used to answer these RQ's.

### 3.1 RO 1: Survey of Open-source Software Developers

We conduct a cross-sectional survey of developers from several open-source projects. We create a survey using SurveyMonkey [3] website and create a web link to it. We sent an email with the link to our survey to the project mailing lists (for example: chromium-dev@chromium.org) of Google Chromium, Apache OpenOffice, Apache Tomcat, Geronimo and Presto projects. Our survey consists of five mandatory questions. Table 2 lists all the five survey questions. In this survey, we ask software developers about their opinion towards source code logging. For example, we ask questions related to *'frequency of log churning*, *'log verbosity level assignment'*, and *'what & where to log ?'*. We believe that inputs from practitioners are needed to inform our research on logging.

[3] https://www.surveymonkey.com/

### 3.2 RO 2: Empirical Analysis of Logged and Non-logged Code Constructs

We perform two level -*high level (file level)* and *low level (catch-block)* level-analysis of logged and non-logged code on Java projects. We perform ***statistical-analysis*** of logged and non-logged files and answer three RQ's. First, we analyze the distribution of logged files in total files. Second, we analyze the complexity of logged and non-logged files. We use Source Lines of Code (SLOC) of a file as a measure of complexity for logged and non-logged files. Third, we analyze co-relation between file complexity and its log statement count.

We perform ***statistical-analysis*** of logged and non-logged catch-blocks and answer five RQ's. First, we analyze complexity of try-blocks associated with logged and non-logged catch-blocks. We measure complexity using three parameters: SLOC, number of arithmetic operators, and number of functions called in try-blocks. Second, we analyze logging ratio of all the exception types. Third, we analyze the contribution of all the exception types in total catch-blocks and in logged catch-blocks. Fourth, we identify top-20 exception types and compare their logging ratios across the projects. Fifth, we analyze whether a single try-block can have both logged and non-logged catch-blocks or not. We use LDA to perform ***content-analysis*** of try-blocks associated with logged and non-logged catch-blocks. We perform content analysis to find whether try-blocks associated with logged and non-logged catch-blocks consists of different topics on not. LDA is a popular topic modeling technique, but currently there is no study which analyzes topics present in try-blocks associated with logged and non-logged catch-blocks.

We perform two level analysis to explore whether logged and non-logged code constructs have differentiating properties at both the levels or not. Multi-level analysis is useful in feature identification phase of logging prediction tools. As it gives insights about which part of the source code is more useful for feature extraction for training the logging prediction model.

### 3.3 RO 3: With-in Project Logging Prediction

We use findings from our previous work [10] for feature set identification. We extract 46 and 28 distinguishing features for catch-blocks and if-blocks logging prediction. We categories each features based on its *domain*, *type* and *class*. *Domain*, identifies the part of source code from where the feature is extracted. *Type*, identifies whether the feature is boolean, numeric or textual. *Class*, identifies whether the feature is positive or negative. We extract features from different domains, as our previous study show that both high level and low level code construct show differentiating properties towards logged and non-logged code constructs. Using these features we propose two machine learning based models: ***LogOpt*** [12] and ***LogOptPlus*** [11]. *LogOpt* for catch-blocks and *LogOptPlus* for if-blocks logging prediction for Java projects. In this work, we answer two RQ's: first, we analyze properties of all the proposed features, second, we analyze the performance of the proposed machine learning models for the task of

logging prediction. This work serves as a first step towards machine learning based catch-blocks and if-blocks logging prediction for Java projects.

Table 1: Experimental Dataset Details

| **Type** | **Tomcat** | **CloudStack** | **Hadoop** |
|---|---|---|---|
| Version | 8.0.9 | 4.3.0 | 2.7.1 |
| No. of Java File | 2036 | 5350 | 6331 |
| Total Catch-Blocks | 3279 | 8077 | 7947 |
| Logged Catch-Blocks | 887 (27%) | 2792(34.56%) | 2078(26.14%) |
| Distinct Exception Types | 119 | 163 | 265 |
| Total If-Blocks | 16991 | 65392 | 32143 |
| Logged If-Blocks | 1423 (8.37%) | 5653 (8.64%) | 3407 (10.60%) |

### 3.4 RO 4: Cross-project Logging Prediction Analysis

Cross-project logging prediction is a relatively unexplored area. Hence, as a first step toward cross-project logging prediction, we perform a large scale study of cross-project catch-block logging prediction on Java projects. We explore effectiveness of nine machine learning classifiers for cross-project catch-blocks logging prediction. We carefully select algorithms belonging to different domains such as probabilistic, decision tree etc. We answer several RQ's. We first compare performances of different machine learning algorithms for within-project and cross-project logging prediction. We then analyze performance of algorithm with respect to different parameters in order to identify whether they provide complementary information or not and finally compare the performances of single-project and multi-project training models. The output of this analysis is beneficial for building a robust cross-project logging prediction model.

### 3.5 RO 5: Ensemble Based Cross-project Logging Prediction

During cross-project model building, we face two main challenges. First, non-uniform distribution of numerical attributes, second, vocabulary mis-match problem. In order to address these challenges, we perform standardization of the attributes and also propose uses of ensemble based learning. Ensemble based learning is useful in improving the accuracy of base machine learning algorithms but their effectiveness with respect to cross-project logging prediction is unexplored yet. As a first step towards effective cross-project logging prediction, we propose ***ECLogger***, a novel ensemble based model for cross-project catch-blocks logging prediction. ECLogger uses 9 base machine learning algorithms and three ensemble techniques. We create 8 EClogger$_{bagging}$ models, by applying bagging on 8 base machine learning algorithms. We create 466 ECLogger$_{AverageVote}$ models by applying average vote ensemble technique on every possible combination

of base machine learning algorithms, using group of 3-9 algorithms at a time. Similarly, we create 466 ECLogger$_{MajorityVote}$ models using majority vote ensemble technique. In this work, we answer two RQ's. First, we compare the performances of baseline machine learning classifiers with the ECLogger classifier for cross-project catch-blocks logging prediction individually for each source and target project pair. Second, we compare the performance of baseline machine learning classifiers with ECLogger classifiers for cross-project catch-blocks logging prediction average on all source and target project pairs. We believe that this work serves as a first step towards improving cross-project logging prediction performance for Java projects.

## 4 Results

In this section, we give brief detail about the experimental dataset used for this study and the metrics used for prediction model evaluation. We then present results obtained for each RO.

### 4.1 Experimental Dataset

We select three large open-source projects from Apache Software Foundation (ASF) for evaluation: Tomcat, CloudStack, and Hadoop. Apache Tomcat is a web server that implements Java EE specifications like Java Servlet, Java Sever Pages and Java EL. CloudStack provides public, private, and hybrid cloud solutions. CloudStack provides a highly available and scalable Infrastructure as a Service (IaaS) cloud computing platform for deployment and management of networks of virtual machines. Hadoop is a framework that enables distributed processing of large datasetset. Table 1 shows details of the experimental dataset. All the three projects used in our study are long lived Java projects having several years ($\approx$ 7 to 17 year) of history. Table 1 shows that all three projects have several thousands of SLOC. Tomcat, CloudStack and Hadoop have been previously used by the research community for logging and other studies [3][8][17][23].

### 4.2 Metrics

In this subsection, we describe the performance metrics used to evaluate the effectiveness of the prediction models. At the time of prediction, we count the total number of logged code constructs predicted as logged ($C_{l \to l}$), logged code constructs falsely predicted as non-logged ($C_{l \to n}$), non-logged code constructs predicted as non-logged ($C_{n \to n}$), and non-logged code constructs predicted and logged ($C_{n \to l}$). Using these 4 values, we compute the following metrics:

$$Logged\ Precision\ (LP) = \frac{C_{l \to l}}{C_{l \to l} + C_{n \to l}} \times 100 \tag{1}$$

$$Logged\ Recall\ (LR) = \frac{C_{l \to l}}{C_{l \to l} + C_{l \to n}} \times 100 \tag{2}$$

$$Logged\ F-measure\ (LF) = \frac{2 \times LP \times LR}{LP + LR} \times 100 \tag{3}$$

$$Accuracy\ (ACC) = \frac{C_{l \to l} + C_{n \to n}}{C_{l \to l} + C_{l \to n} + C_{n \to n} + C_{n \to l}} \times 100 \tag{4}$$

**Area under the ROC curve (RA):** RA measures the likelihood that a positive class instance is given a high likelihood score compared to a negative class instance. RA can take any value in the range 0 to 1, higher the better.

Table 2: Results of survey performed on software developers from open-source projects

| | |
|---|---|
| **Q1** | **How many years of experience do you have in software development?** |
| | ⊙Less than 1 year (0%) ⊙Between 1 and 5 years (25.53%) ⊙More than 5 years (76.47% ) |
| **Q2** | **Do you believe where and what to log is subjective?** |
| | ⊙Strongly Disagree (11.76% ) ⊙Disagree (11.76% ) ⊙Neutral (17.65% ) ⊙Agree (47.06%) ⊙Strongly Agree (11.76%) |
| **Q3** | **Fatal, Debug, Error and Info are various log levels. Do you believe assigning right verbosity level for a give case is challenging and developers make mistakes in verbosity level assignment?** |
| | ⊙Strongly Disagree (0%) ⊙Disagree (17.65%) ⊙Neutral (25.53%) ⊙Agree (25.53%) ⊙Strongly Agree (35.29%) |
| **Q4** | **Do you believe a static code or program analysis tool or checker which can guide a developer in recommending where to log and which log level to use will be useful?** |
| | ⊙Strongly Disagree (05.88%) ⊙Disagree (35.29%) ⊙Neutral (25.53%) ⊙Agree (29.41%) ⊙Strongly Agree (05.88%) |
| **Q5** | **In your experience, the modification and churn of logging code is:** |
| | ⊙Rare (47.06%) ⊙Between Rare and Common (35.29%) ⊙Common (17.65%) |

### 4.3 RO 1: Survey of Open-source Software Developers

We received a total of 17 responses of the survey. Among the 17 respondents more than 75% of the respondents have more than 5 years of experience in software development. Questions 2-4 are Likert questionnaire items in which the respondents specify their level of agreement or disagreement (ordinal variable) on a symmetric agree-disagree scale consisting of 5 choices. 60% of the respondents believe that where and what to log is subjective. 60% of the respondents believe that assigning right verbosity level for a given case is challenging and developers make mistakes in verbosity level assignment. 15% of the developers responded that in their experience, the modification and churn of logging code are common.

For Question 4, we conduct a Chi-Squared Test (non-parametric test) to test the goodness of fit between an expected frequency distribution (equal distribution between agree and disagree) and the observed frequency distribution. In the statistical significance testing, we get the p-value as 0.78. Since the p-value is greater than 0.1, we have no presumption against the null hypothesis (that a static code or program analysis tool or checker which can guide a developer in recommending where to log and which log level to use will be useful). The results of the survey reveal that more than 33% of the respondents believe that a checker to assist developers in automated logging can be useful.

> Software developers face challenges in source code logging and logging prediction tools can be beneficial.

### 4.4 RO 2: Empirical Analysis of Logged and Non-logged Code Constructs

We answer all the 9 research questions related to two level empirical analysis of logged and non-logged code constructs by conducting experiments on Tomcat, CloudStack, and Hadoop project (refer to Table 1). Our ***statistical-analysis*** of files shows that very less percentage of files are logged. For example, only 14.9% of files are logged in Tomcat project. Our analysis reveals that SLOC of logged files is considerably higher as compared to non-logged files and there exists a positive correlation between SLOC of a file and its log statement count. However, we notice presence of some very large non-logged files. The manual analysis reveals that these files are tool generated and hence do not consist of any log statements. ***Statistical-analysis*** of try-blocks show that for some projects try-blocks associated with logged catch-blocks have higher complexity as compared to that of non-logged catch-blocks. Our analysis reveals that few try-blocks i.e., $\leq$1.5%, consists of both logged and non-logged catch-blocks. We also observe that logging ratio of exception types is project specific. For example, exception type 'IOException', has logging ratio of 37.25%, 66.81%, and 27.72% for Tomcat, CloudStack, and Hadoop project, respectively. ***Content-analysis*** of try-blocks associated with logged and non-logged catch-blocks revels presence of different topics. For example, 'thread.sleep' topic is present in the non-logged catch-blocks of Tomcat project. Manual analysis reveals that in 84 occurrences of 'thread.sleep' it occurred 71 times in try-blocks associated with non-logged catch-blocks. More details on two level empirical analysis of logged and non-logged code constructs can be found in our published work [10].

> Empirical analysis reveals presence of different features in logged and non-logged code constructs at both the levels.

Table 3: LF for catch-blocks and if-blocks logging prediction using RF classifier

| Model | Type | Tomcat | CloudStack |
|---|---|---|---|
| **LogOpt** | **Catch-blocks** | 85.50% | 93.4% |
| **LogOptPlus** | **If-blocks** | 80.70% | 92.25% |

### 4.5 RO 3: With-in Project Logging Prediction

We empirically analyze all the proposed Boolean & numeric features and evaluate *LogOpt* and *LogOpt* models with five machine learning algorithms on two datasets (Tomcat and CloudStack). Results show that Random Forest (RF) classifier give the highest LF for both catch-blocks and if-blocks logging prediction for both the projects. RF classifier gives the highest LF of 93.4% and 92.25% for catch-blocks and if-blocks logging prediction (refer to Table 3). For both catch-blocks and if-blocks model gives the better results on CloudStack project. We analyze some of the exception types in both Tomcat and CloudStack project. Our analysis reveals the there are certain exception types in CloudStack project which are heavily non-logged. We believe that this is the reason for model performing better on the CloudStack project as compared to the Tomcat project for catch-blocks logging prediction. More details on logged code constructs prediction can be found in our published work [12] and [11].

> Machine learning based models are effective in catch-blocks and if-blocks logging prediction on Java projects.

### 4.6 RO 4: Cross-project Logging Prediction Analysis

We conduct our cross-project experiments on all the three projects (Tomcat, CloudStack, and Hadoop). We create six source & target project pairs by considering one project as source project and other two projects as target project (one at a time). Results show that performance of machine learning algorithms degraded considerably for cross-project catch-blocks logging prediction. We notice upto 6.37% to 18.05% drop in performance of machine learning algorithms for cross-project prediction as compared to within-project logging prediction. We also notice that performance of single-project and multi-project training models is similar. However, we notice that different classifiers are complementary to each other. For example, for CloudStack→Tomact project pair, ADTree algorithm gives the highest LP, ACC and RA values, whereas BN gives the highest LF. Hence, combination of different machine learning algorithms can be useful in improving cross-project catch-blocks logging prediction performance. More details on cross-project logging prediction analysis can be found in our published work [9].

Different classifiers provide complementary information and hence ensemble based approach can be effective in improving the cross-project catch-blocks logging prediction performance.

### 4.7 RO 5: Ensemble Based Cross-project Logging Prediction

We evaluate performances of *ECLogger* models on all the three projects (Tomcat, CloudStack, and Hadoop). For each source and target project pair, we report the ECLogger model giving the best results. We observe that ensemble based models are effective in improving the cross-project catch-blocks logging prediction performances. Results show that average vote based ECLogger model performs the best and give highest results (both individual as well as average performance) as compared to the baseline classifier. Average vote model gives the highest improvement of 3.12% (average LF) and 6.08% (average ACC). Overall, we observe that CloudStack project is more generalizable as compared to the Tomcat and the Hadoop project for cross-project catch-block logging prediction. Manual analysis reveals that CloudStack project provide support for both Tomcat and Hadoop projects and hence gives better results for cross-project prediction. More details on ensemble based cross-project logging prediction can be found in our published work [9].

Ensemble based model is effective in improving the cross-project catch-blocks logging prediction performance.

## 5 Threats to Validity

In this section, we discuss various threats to validity to this work.

### 5.1 External Validity

External validity refers to the generalization of the results. In our work, we investigate three open source Java projects which have different domains and sizes. However, the results may not be generalizable to other types of project such as closed source or projects written in other programming languages as different projects may have different logging practices.

### 5.2 Construct Validity

The construct threat refers to how we identify log statements in the source code. We create 26 regular expressions to find all the log statements. The manual inspection reveals that we identify all the types of log statements. However, there is still a possibility that we missed some types of log statements.

### 5.3 Internal Validity

Threat to internal validity refers to the error in our code and bias in sampling. We have double checked our code to remove errors from our code. To mitigate the sampling bias in training and testing dataset split creation, we create 10 training and testing dataset and report the average results.

## 6 Conclusions and Future Work

This work aims towards analysis and prediction of log statements on Java projects. We start by performing a survey of software developers from open-source projects. Results of the survey revel that developers face challenges in logging. Now in order to provide software developers with better logging mechanism, we perform an in-depth, focused, and two level empirical study of logged and non-logged code constructs. Preliminary results show distinguishing features between logged and non-logged code constructs which can be helpful in predicting logged code constructs. Using inputs from our empirical study we propose machine learning based models for catch-blocks and if-blocks logging prediction on Java projects. Initial results show that both the models are effective for within-project logging prediction. We analyze performance of this catch-block logging prediction mode for cross-project logging prediction. Our results reveal that performance of model degrades considerable for cross-project logging prediction as compared to within-project logging prediction. We then propose an ensemble based approach to improve cross-project catch-block logging prediction performance. Results show that ensemble based approach is effective in improving cross-project logging prediction performance. Our initial results are encouraging but much work yet remains to improve within-project and cross-project logging prediction performance. In future, we plan to extend the current work for other types of code constructs such as while loops, switch statements. ECLogger model proposed in this thesis is used for cross-project logging prediction only for catch-blocks. Hence, can be extended to other types of code constructs such as if-block. We plan to extend this work for other language such as C#, Python. We also plan to study the evolution of log statements over the lifetime of projects.

# Bibliography


[1] Blackberry enterprise server logs submission. `BlackBerryEnterpriseServerLogsSubmission`. [Online; accessed 4-June-2016].

[2] Java regains spot as most popular language in developer index. `http://www.infoworld.com/article/2909894/application-development/java-back-at-1-in-language-popularity-assessment.html`. [Online; accessed 19-March-2016].

[3] Boyuan Chen and Zhen Ming (Jack) Jiang. Characterizing logging practices in java-based open source software projects – a replication study in apache software foundation. *Empirical Software Engineering*, pages 1–45, 2016.

[4] Qiang Fu, Jieming Zhu, Wenlu Hu, Jian-Guang Lou, Rui Ding, Qingwei Lin, Dongmei Zhang, and Tao Xie. Where do developers log? an empirical study on logging practices in industry. In *Companion Proceedings of the 36th International Conference on Software Engineering*, ICSE Companion 2014, pages 24–33, 2014.

[5] Steven Haines. Top 5 performance problems in .net applications. `https://blog.appdynamics.com/net/top-5-performance-problems-in-net-applications/`. [Online; accessed 27-July-2016].

[6] Z. M. Jiang, A. E. Hassan, G. Hamann, and P. Flora. Automatic identification of load testing problems. In *Software Maintenance, 2008. ICSM 2008. IEEE International Conference on*, pages 307–316, Sept 2008.

[7] Z. M. Jiang, A. E. Hassan, G. Hamann, and P. Flora. Automated performance analysis of load tests. In *Software Maintenance, 2009. ICSM 2009. IEEE International Conference on*, pages 125–134, Sept 2009.

[8] Suhas Kabinna, Cor-Paul Bezemer, Weiyi Shang, and Ahmed E. Hassan. Examining the stability of logging statements. In *The 23rd IEEE International Conference on Software Analysis, Evolution, and Reengineering (SANER)*, 2016.

[9] S. Lal, N. Sardana, and A. Sureka. Eclogger: Cross-project catch-block logging prediction using ensemble of classifiers. *eInformatica Software Engineering Journal*, 2017.

[10] Sangeeta Lal, Neetu Sardana, and Ashish Sureka. Two level empirical study of logging statements in open source java projects. *International Journal of Open Source Software and Processes (IJOSSP)*, 6(1):49–73, 2015.

[11] Sangeeta Lal, Neetu Sardana, and Ashish Sureka. Logoptplus: Learning to optimize logging in catch and if programming constructs. In *2016 IEEE 40th Annual Computer Software and Applications Conference (COMPSAC)*, volume 1, pages 215–220, June 2016.

[12] Sangeeta Lal and Ashish Sureka. Logopt: Static feature extraction from source code for automated catch block logging prediction. In *9th India Software Engineering Conference (ISEC)*, pages 151–155, 2016.

[13] Michelle Levesque. Fundamental issues with open source software development. *First Monday*, 9(4), 2004.

[14] Heng Li, Weiyi Shang, and Ahmed E Hassan. Which log level should developers choose for a new logging statement? *Empirical Software Engineering*, pages 1–33, 2016.

[15] J. Nam, S. J. Pan, and S. Kim. Transfer defect learning. In *2013 35th International Conference on Software Engineering (ICSE)*, pages 382–391, May 2013.

[16] Weiyi Shang, Zhen Ming Jiang, Bram Adams, Ahmed E Hassan, Michael W Godfrey, Mohamed Nasser, and Parminder Flora. 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, 2014.

[17] Weiyi Shang, Meiyappan Nagappan, and Ahmed E. Hassan. Studying the relationship between logging characteristics and the code quality of platform software. *Empirical Software Engineering*, 20(1):1–27, 2015.

[18] X. Xia, D. Lo, S. McIntosh, E. Shihab, and A. E. Hassan. Cross-project build co-change prediction. In *2015 IEEE 22nd International Conference on Software Analysis, Evolution, and Reengineering (SANER)*, pages 311–320, March 2015.

[19] Ding Yuan, Soyeon Park, Peng Huang, Yang Liu, Michael M. Lee, Xiaoming Tang, Yuanyuan Zhou, and Stefan Savage. Be conservative: Enhancing failure diagnosis with proactive logging. In *Proceedings of the 10th USENIX Conference on Operating Systems Design and Implementation*, OSDI'12, pages 293–306, 2012.

[20] Ding Yuan, Soyeon Park, and Yuanyuan Zhou. Characterizing logging practices in open-source software. In *Proceedings of the 34th International Conference on Software Engineering*, ICSE '12, pages 102–112, 2012.

[21] Ding Yuan, Jing Zheng, Soyeon Park, Yuanyuan Zhou, and Stefan Savage. Improving software diagnosability via log enhancement. In *Proceedings of the Sixteenth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS XVI, pages 3–14, New York, NY, USA, 2011. ACM.

[22] Jieming Zhu, Pinjia He, Qiang Fu, Hongyu Zhang, M.R. Lyu, and Dongmei Zhang. Learning to log: Helping developers make informed logging decisions. In *Software Engineering (ICSE), 2015 IEEE/ACM 37th IEEE International Conference on*, volume 1, pages 415–425, May 2015.

[23] Thomas Zimmermann, Nachiappan Nagappan, Harald Gall, Emanuel Giger, and Brendan Murphy. Cross-project defect prediction: A large scale experiment on data vs. domain vs. process. In *Proceedings of the the 7th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on The Foundations of Software Engineering*, ESEC/FSE '09, pages 91–100, New York, NY, USA, 2009. ACM.

*Hammouda, I., Lundell, B., Madey, G. and Squire, M. (Eds.) Proceedings of the Doctoral Consortium at the 13th International Conference on Open Source Systems, Skövde University Studies in Informatics 2017:1, ISSN 1653-2325, ISBN 978-91-983667-1-6, University of Skövde, Skövde, Sweden.*



*Skövde University Studies in Informatics 2017:1*
*ISSN 1653-2325*
*ISBN 978-91-983667-1-6*

*www.his.se*

UNIVERSITY
OF SKÖVDE