Table of Contents

International Journal of Open Source Software and Processes

Volume 7 • Issue 2 • April-June-2016 • ISSN: 1942-3926 • eISSN: 1942-3934
An official publication of the Information Resources Management Association

Research Articles

1 A Comparative Analysis of Open Source Network Monitoring Tools
Ali Al Shidhani, The Research Council of Oman, Information and Communications Technology Research, Muscat, Oman
Khalil Al Maawali, Information Technology Authority, Seeb, Oman
Dawood Al Abri, Sultan Qaboos University, Department of Electrical and Computer Engineering, Muscat, Oman
Hadj Bourdoucen, Sultan Qaboos University, Communication and Information Research Center, Muscat, Oman

20 Analytical Study on Bug Triaging Practices
Anjali Goyal, Jaypee Institute of Information Technology, Department of Computer Science and Engineering, Noida, UP, India
Neetu Sardana, Jaypee Institute of Information Technology, Department of Computer Science and Engineering, Noida, UP, India

43 Improving Logging Prediction on Imbalanced Datasets: A Case Study on Open Source Java Projects
Sangeeta Lal, Jaypee Institute of Information Technology, Department of CSE & IT, Noida, Uttar-Pradesh, India
Neetu Sardana, Jaypee Institute of Information Technology Noida, Department of CSE & IT, Noida, Uttar-Pradesh, India
Ashish Sureka, ABB Corporate Research Center, Bangalore, India

72 Using Data Mining Techniques with Open Source Software to Evaluate the Various Factors Affecting Academic Performance: A Case Study of Students in the Faculty of Information Technology
Feras Hanandeh, Hashemite University, Faculty of Information Technology, Zarqa, Jordan
Majdi Y. Al-Shannag, Yarmouk University, Faculty of Information Technology, Irbid, Jordan
Maha Mahdi Alkhaffaf, World Islamic Sciences University, Department of Management Information Systems, Amman, Jordan

COPYRIGHT

The International Journal of Open Source Software and Processes (IJOSSP) (ISSN 1942-3926; eISSN 1942-3934), Copyright © 2016 IGI Global. All rights, including translation into other languages reserved by the publisher. No part of this journal may be reproduced or used in any form or by any means without written permission from the publisher, except for noncommercial, educational use including classroom teaching purposes. Product or company names used in this journal are for identification purposes only. Inclusion of the names of the products or companies does not indicate a claim of ownership by IGI Global of the trademark or registered trademark. The views expressed in this journal are those of the authors but not necessarily of IGI Global.

The International Journal of Open Source Software and Processes is indexed or listed in the following: ACM Digital Library; Bacon’s Media Directory; Cabell’s Directories; Compendex (Elsevier Engineering Index); DBLP; GetCited; Google Scholar; INSPEC; JournalTOCs; MediaFinder; Norwegian Social Science Data Services (NSD); SCOPUS; The Index of Information Systems Journals; The Standard Periodical Directory; Ulrich’s Periodicals Directory
Improving Logging Prediction on Imbalanced Datasets: A Case Study on Open Source Java Projects

Sangeeta Lal, Jaypee Institute of Information Technology Noida, Department of CSE & IT, Noida, Uttar-Pradesh, India
Neetu Sardana, Jaypee Institute of Information Technology Noida, Department of CSE & IT, Noida, Uttar-Pradesh, India
Ashish Sureka, ABB Corporate Research Center, Bangalore, India

ABSTRACT

Logging is an important yet tough decision for OSS developers. Machine-learning models are useful in improving several steps of OSS development, including logging. Several recent studies propose machine-learning models to predict logged code construct. The prediction performances of these models are limited due to the class-imbalance problem since the number of logged code constructs is small as compared to non-logged code constructs. No previous study analyzes the class-imbalance problem for logged code construct prediction. The authors first analyze the performances of J48, RF, and SVM classifiers for catch-blocks and if-blocks logged code constructs prediction on imbalanced datasets. Second, the authors propose LogIm, an ensemble and threshold-based machine-learning model. Third, the authors evaluate the performance of LogIm on three open-source projects. On average, LogIm model improves the performance of baseline classifiers, J48, RF, and SVM, by 7.38%, 9.24%, and 4.6% for catch-blocks, and 12.11%, 14.95%, and 19.13% for if-blocks logging prediction.

KEYWORDS

Data Sampling, Debugging, Ensemble Methods, Imbalanced Data, Logging, Machine Learning, Open Source, Tracing

INTRODUCTION

Debugging plays an essential role in the development of any Open Source Software (OSS), as the speed of debugging can be of vital importance in adopting any OSS. Logging, an important software development practice, is crucial for debugging in the production setting, and can play a major role in the success of any OSS. Logging is used to record execution information about the program. The recorded log assists the software developers in fixing bugs. An empirical study performed by Yuan et al. (2012a) on OSS showed that bug reports consisting of log statements are fixed 2.2 times faster than the bug report without log statements. Stack traces produced at the time of program failure are also useful in fixing the bug, but they only provide information about the exact point where the failure occurs, and do not give any information about the state just before the failure (StackExchange, n. d.). In contrast, log statements provide history about the failed event which is useful in debugging. In addition to debugging, logging is useful in several other software development activities, such as remote issue resolution (BlackBerry Enterprise Server Logs Submission, 2015), performance problem diagnosis (Nagaraj, Killian, & Neville, 2012), workload modeling (Sharma, Chudnovsky, Hellerstein, Rifaat, & Das, 2011), and load testing (Jiang, Hassan, Hamann, & Flora, 2008, 2009). The importance of logging can be considered from the fact that log statements are pervasive in OSS as various OSS
are heavily logged. For example, the widely used OpenSSH project consist of 3407 log statements (Yuan et al. (2012b)). The Tomcat, CloudStack, and Hadoop project consists of thousands of log statements (refer to Table 10 in the Appendix).

Log statements are useful, but they have a cost-benefit tradeoff. Excess log statements in source code can generate too many trivial logs, making debugging more challenging by hiding important debugging information. Excess log statements can also increase performance (I/O intensive activity) and cost (development and maintenance) overhead. Excessive logging is one reason for poor performance in several OSS, such as Tomcat, Jetty, JBoss (Granber, 2016a; Grabner, 2016b). Like excessive logging, sparse logging is also problematic. It can omit important debugging information and decrease the benefits of logging. Hence, it is important to optimize the number of log statements in the source code.

Optimizing log statements in the source code, or identifying code constructs that must be logged, is a nontrivial and technically challenging task. It happens because software developers and code contributors in OSS often are not provided with any formal guidelines about software logging. Hence, logging is often based on domain knowledge and experience of the software developers. Also, logging practices can differ from project to project, depending on the application need. This problem can be exaggerated in OSS because contributions to OSS are often voluntary. These systems may lack documentation (Levensque, 2005) or appropriate coding standards ((Fitzgerald, 2004). All the knowledge and experience is in the mind of the experienced software developer. In addition, finding mentors in OSS is also very challenging (Steinmacher et al. (2013). As a result, source code logging can be challenging for new OSS developers who lack experience and domain knowledge. Previous studies show that software developers face difficulties in identifying source code constructs that need to be logged or in optimal source code logging (Fu, et al., 2014; Zhu et al., 2015). Hence, tools and techniques to help software developers make informed logging decisions in the source code could be beneficial.

Several recent studies propose machine-learning-based models to predict log statements in source code help software developers in identifying the source code constructs that must be logged (Fu et al., 2014; Lal & Sureka, 2016; Lal, Sardana, & Sureka, 2016; Saini, Sardana, & Lal, 2016; Zhu et al., 2015). These techniques use static features from the source code to train the machine-learning-based model used to predict logged and nonlogged source code constructs. However, a major challenge in machine-learning-based logged code construct prediction is that only a small percentage of code constructs are logged. There is an unequal distribution of logged and nonlogged constructs, and this distribution differs for various types of code constructs (i.e., if-blocks, catch-blocks). For example, in OSS such as Hadoop, Tomcat, and CloudStack, which are representative OSS systems in their class of application, an imbalance occurs in terms of logged and nonlogged catch-block and if-block constructs. These three OSS log less than 35% of the catch-blocks and less than 11% of the if-blocks. This is also known as the class-imbalance problem (He & Garcia, 2009). The imbalance does not mean developers omitted logging statements; rather, it shows that the majority of the if-blocks and try-catch-blocks do not require tracing.

The Class-imbalance problem makes predicting positive class instances (logged source code constructs) with high accuracy a challenging task. The literature reports that in the case of imbalanced dataset several machine-learning algorithms are biased toward the majority class or negative class (i.e., nonlogged code constructs), and perform poorly in recognition of the minority class or positive class. Hence, their prediction performance degrades severely in cases of imbalanced data. The Class-imbalance is a well-known problem in the area of machine learning. It has been studied in the context of several software engineering domains such as blocking bug prediction (Xia, Lo, Shihab et al., 2015) and defect prediction (Liu, Miao, & Zhang, 2014). However, no focused study exists that analyzes class-imbalance problems in the context of logging prediction. Previous studies on logging prediction use sampling—i.e., either majority class undersampling (Fu et al., 2014; Lal & Sureka, 2016, Lal et al., 2016, Saini et al., 2016) or minority class oversampling (Zhu et al., 2015)—to balance the dataset.
address the class-imbalance problem of the logged and nonlogged constructs. Sampling techniques are useful in balancing the dataset artificially, but identifying the most suitable sampling technique and rate of sampling is challenging (Estabrooks & Japkowicz, 2001). Hence, it is important to analyze the performance of various algorithms in the context of imbalanced logging prediction. In this study, the authors investigate the problem of logging prediction on imbalanced datasets within the context of OSS systems, to answer the following six Research Questions (RQs):

**RQ1:** What are the performances of base classifiers for logging prediction on imbalanced datasets? Specifically, the authors intend to investigate performances of three base classifiers—J48, Random Forest (RF), and Support Vector Machines (SVM)—for catch-blocks and if-blocks logging prediction on imbalanced datasets.

**RQ2:** What impact do various decision threshold boundaries have on logging prediction performances of base classifiers on imbalanced datasets? In particular, can shifting classification decision threshold boundaries have a positive impact on logging prediction on imbalanced datasets?

**RQ3:** What are the performances of LogIm models for logging prediction on imbalanced datasets? Specifically, the authors investigate performance of the proposed ensemble and decision-threshold based model, LogIm, for logging prediction on imbalanced datasets.

**RQ4:** What are the logging prediction performances of LogIm models in the context of various projects? In particular, the authors investigate performance of all the LogIm models with respect to various projects.

**RQ5:** What impact do various decision threshold boundaries have on logging prediction performance of LogIm models on imbalanced datasets? Specifically, the authors investigate behavior of all the LogIm models with respect to various decision threshold boundaries.

**RQ6:** What is the time complexity of various LogIm models? Specifically, the authors investigate the amount of time each LogIm model takes to predict logged or nonlogged code constructs.

To answer these research questions, the authors conducted a comprehensive study. First, they analyze performances of three base classifiers—J48, Random Forest (RF), and Support Vector Machines (SVM)—for catch-blocks and if-blocks logging prediction on imbalanced datasets. Second, they propose the LogIm model for logging prediction on imbalanced datasets. LogIm is a machine-learning-based framework that leverages the concept of ensemble methods and various decision boundaries to improve logging prediction on imbalanced datasets. The authors use three base classifiers (J48, RF, and SVM) and six ensemble techniques (average vote, majority vote, maximum vote, bagging, boosting, and stacking), and create ten ensemble LogIm models. The objective is to gain insights about source-code logging in open-source systems in particular, rather than proprietary or closed-source software systems. Hence, the authors evaluate performance of LogIm models on three large, open-source Java projects: Tomcat, CloudStack, and Hadoop, containing a total of 19,303 catch-blocks and 114,526 if-blocks.

Results show that LogIm models are effective in improving the catch-block and if-block logging prediction performance on imbalanced datasets. RF and the bagging-based LogIm model (LBAR) perform best and give the highest average F-score of 80.65% and 62.29% respectively for catch-blocks and if-blocks logging prediction. On average, the LBAR model improves the baseline classifiers J48, RF, and SVM by 7.38%, 9.24%, and 4.6% respectively for catch-blocks logging prediction, and by 12.11%, 14.95%, and 19.13% respectively for if-blocks logging prediction. Overall, the LogIm models give the best performance on the CloudStack project for both catch-blocks and if-blocks logging prediction. In this study, the authors ask if it is possible to identify general features (and features not specific to any one project) extracted from source code, which could serve as predictors for logging statements. The authors also attempt to investigate and examine patterns and regularities in OSS, which can be exploited to automate the task of inserting logging statements at the correct position.
The result shows that common practices and developer behaviors across various open-source software projects can be abstracted for the purpose of building a logging-prediction model.

In summary, this study makes the following novel and unique contributions:

1. The authors analyze performances of J48, RF, and SVM classifiers for catch-blocks and if-blocks logging prediction on imbalanced datasets;
2. The authors propose ensemble and decision-threshold-based frameworks, LogIm, for catch-blocks and if-blocks logging prediction on imbalanced datasets;
3. The authors present results of comprehensive analysis of the LogIm model on three large, open-source Java projects (i.e., Tomcat, CloudStack, and Hadoop). The experimental results demonstrate that LogIm is effective in improving logging prediction performance on imbalanced datasets.

RELATED WORK

In this section, the authors discuss prior research regarding logging applications, logging analysis and prediction, ensemble learning and imbalanced learning, and prediction models uses in OSS.

Logging Applications

Log statements present in the source code, as well as log messages generated at the time of program execution, have been found useful in software engineering applications (Jiang et al., 2008, 2009; Mariani & Pastore, 2008; Shang, Nagappan, & Hassan, 2015; Xu, Huang, Fox, Patterson, & Jordan, 2009; Yuan et al., 2010). Shang et al. (2015) proposed various product and process metrics based on log statements for software defect prediction. Yuan et al. (2010) proposed a constraint-satisfaction-based tool, SherLog, which uses log messages for root-cause identification of the failures. SherLog uses log messages generated by failed production runs and source code to obtain data flow and control flow information. Hence, it does not require any reproduction of the error for failure diagnosis. Mariani & Pastore (2008) propose a heuristic-based log-analysis approach to finding root causes of failures caused by multiple events. Their method generates finite automata-based models from the logs generated during failed execution. They compare these with the logs generated during successful runs, to provide the software developers with information about illegal events. Jiang et al. (2008, 2009) use log messages for load-testing problems. Xu et al. (2009) propose a method for mining log messages to detect system runtime problems. The wide use of logs in several software development activities shows the importance of logging. Outcome of all these studies is dependent on the quality of log statements inserted in the source code. The work presented in this paper, improves the quality of logging in the source code and hence complements the above-mentioned studies.

Logging Analysis and Improvement

Several prior studies focus on analysis and improvement of logging (Fu et al., 2014; Kabinna, Shang, Bezemer, & Hassan, 2016; Lal, Sardana, & Sureka, 2015b; Lal & Sureka, 2016; Lal, Sardana & Sureka, 2016; Lal, Sardana & Sureka, 2017; Li, Shang, & Hassan, 2016a; Li, Shang, Zou & Hassan., 2016b; Saini et al., 2016; Yuan et al., 2012a; Yuan, Park, Zhou, & Savage, 2012c; Zhu et al., 2015). Yuan et al. (2012a) proposed a conservative automated logging tool (ErrLog) that logs all generic exceptions in source code. Fu et al. (2014) analyzed C# code constructs and reported five categories of logged code snippets: assertion check, return-value-check, exception types, logic-branch, and observing point. They also proposed a machine-learning-based framework for catch-blocks and return-value-check code snippets logging prediction. Zhu et al. (2015) extended the study performed by Fu et al., proposing LogAdvisor, a tool for exception-snippets and return-value-check code snippets prediction. LogAdvisor gives F1-scores of up to 93.4% and 92.7% for exception types and return value code.
construct prediction, respectively. Lal et al. (2015b) analyze properties of logged and non-logged code construct, and report that logged and non-logged code constructs have distinguishing features. Lal & Sureka (2016) propose LogOpt, and Lal et al. (2016) propose the LogOptPlus tool for catch-blocks and if-blocks logging prediction on Java projects, respectively. Saini et al. (2016) propose the Logger4U tool for if-block logging prediction on Java projects. Previous studies on logging code construct prediction perform experiments on balanced datasets. Also, previous studies do not use comprehensive sets of machine-learning algorithms to evaluate their proposed models. In contrast to these studies, the authors focus on uses of ensemble and decision-threshold-based approaches for logging prediction on imbalanced datasets. In this work, the authors consider J48, RF, and SVM as base classifiers, because previous studies show that these algorithms perform the task of logging code construct prediction better than other algorithms (Fu et al., 2014; Lal & Sureka, 2016, Lal et al. 2016; Saini et al., 2016; Zhu et al., 2015).

There are several other studies that improve other aspects of logging. For example, Yuan et al. (2012c) propose a model for enhancing content of existing log statements in the code. Li et al. (2016a) propose a model for predicting verbosity level of log statements in the code. Kabinna et al. (2016) propose a model for predicting stability of log statements. The work presented in this study complements these studies and predicts which code constructs must be logged.

Imbalance Learning and Ensemble Learning

The class imbalance problem is pervasive in the area of machine learning, and the literature proposes several techniques to address this problem (Chawla, Bowyer, Hall, & Kegelmeyer, 2002; Chawla, Lazarevic, Hall, & Bowyer, 2003; Guo & Viktor, 2004; Kubat & Matwin, 1997). These approaches can be sorted into four main categories: data sampling, cost-sensitive learning, evaluation metrics, and ensemble-based approaches. Data sampling is the primary approach to address the class imbalance problem. Data sampling techniques either undersample majority class or oversample minority class data points, to balance the data set (Chawla et al., 2002; Guo & Viktor, 2004). Cost-sensitive learning-based approaches use various cost metrics to describe the cost associated with incorrectly classifying a particular data point (Sun, Kamel, Wong, & Wang, 2007; Ting, 2000). For example, in defect prediction studies, the cost of wrongly classifying a defect-prone module as non-defect-prone is higher than the cost of incorrectly classifying a non-defect-prone module as defect-prone (Liu et al., 2014). Uyar, Bener, Ciracy, and Bahceci (2006) showed true positive rate improvement of up to 13.6% for imbalanced IVF implantation prediction, by adjusting the optimal decision threshold. Evaluation metrics-based approaches propose new metrics for imbalanced dataset evaluation. Ensemble methods (i.e., combining classifiers) are an effective technique to improve prediction performance. Xia et al. (2015) use ensemble and decision threshold learning-based methods to improve prediction performance of blocking bug prediction. Chawla et al. (2003) propose the SMOTEboost algorithm that embeds SMOTE procedure during the boosting iteration.

All the previous studies on logging prediction use data-sampling methods to address the class-imbalance problem. Sampling is easy to implement, but fine-tuning it most effectively can be challenging. It is unclear which sampling technique is the most effective, or what rate of sampling should be used (Estabrooks & Japkowicz, 2001). In contrast to previous logging prediction studies, the authors use an ensemble and threshold-based approach to improve logging prediction performance on imbalanced datasets.

Improving OSS Quality with Prediction Models

OSS projects are very successful today. However, an OSS project team faces several challenges, such as sustaining end-user confidence, maintaining an acceptable level of quality, and improving usability (Schmidt & Porter, 2001). In the literature, researchers propose several prediction models to help OSS developers and to improve the overall quality of OSS. For example, Anvik, Hiew, and Murphy (2006) propose a model to predict an expert who can fix the bug. Sun, Lo, Wang, Jiang,
and Khoo (2010) propose model for duplicate bug report prediction. Xia et al. (2015) propose a method for blocking bug prediction. Steinmacher, Wiese, and Gerosa (2012) propose a prediction model to recommend mentors to newcomers in OSS. Knab, Pinzger, and Bernstein (2016) propose a machine-learning-based model leveraging source-code features for defect prediction in OSS. Kim, Whitehead, and Zhang (2008) use a source-code-based prediction model to predict a commit as clean or buggy. These studies show that prediction models can be of great importance in OSS projects. The authors analyzed several previous studies on logging analysis and prediction, to find the kind of software projects used there. The analysis revealed that several studies conducted experiments on OSS projects (Kabinna et al., 2016; Lal et al., 2015b; Lal & Sureka, 2016; Lal et al., 2016; Li et al., 2016a; Li et al., 2016b; Saini et al., 2016; Yuan et al., 2012a; Yuan, Park, & Zhou, 2012b; Yuan et al., 2012c). This shows that logging prediction models can be of great importance in improving the quality of OSS projects.

BACKGROUND

In this paper, the authors use three base machine-learning algorithms and six ensemble techniques. The following subsections provide a brief introduction to each machine-learning algorithm and ensemble technique.

Machine-Learning Algorithms

**J48**

The J48 algorithm is the WEKA (Hall et al., 2009) implementation of the C4.5 algorithm. In the training phase, the J48 algorithm identifies the attribute with maximum information gain. J48 then create branches in the tree for each value of the selected attribute. J48 identifies branches in the tree for which there is no conflict in the data points regarding the class label; i.e., all the data points belong to the same class. Such branches are terminated and the respective class label is assigned to that branch. This process is repeated until the final tree is created. At the time of prediction, the new data instance traverses the tree from root to leaf, based on its attribute values, to find its class label.

**Random Forest (RF)**

Random Forest (RF) (Breiman, 2001) is an ensemble-based algorithm. In the training phase, the RF algorithm creates several decision trees, generated from the sample training dataset. At the time of tree generation, at each step the RF algorithm selects the attributes randomly for splitting or branch creation. In the prediction phase, RF takes a majority vote of these decision trees to predict the label of the new instance.

**Support Vector Machine (SVM)**

Support Vector Machine (SVM) (Han, Pei, & Kamber, 2011) is one of the extensively used classification algorithms. The SVM algorithm plots the data in some higher-dimensional feature space, in order to find a hyper plane that separates the data points belonging to various classes. New instances are mapped into the same space and predicted to belong to the class in which they fall.

**Ensemble Methods**

**Bagging**

Bootstrap Aggregating (Bagging) (Quinlan, 1996) is an ensemble technique, helpful in improving the overall performances of base classifiers. Given a dataset D of size n, Bagging first creates m dataset i, {1, 2, . . ., m}, such that IDi| ≤ n, using random sampling (with replacement). After creating the datasets, Bagging trains the base classifier on each subset D i to create m classifiers. At the time of prediction, output of each of the m classifiers is combined using majority voting.
Stacking

Stacking generalization (stacking) (Wolpert, 1992) is an ensemble method used to combine methods of different types. Stacking divides the training set into two disjoint sets. It trains several base learners on the first part, and tests the base learner on second part. Then, using the prediction as input and labels as output, it trains a higher-level learner.

Boosting

The boosting (Han et al., 2011) ensemble technique assigns weights to each data point present in the training set. It performs multiple iterations through the training set, to learn multiple classifiers. For example, suppose $M_i$ classifier is learned in the $i^{th}$ iteration. Now, in $i + 1^{th}$ iteration, weights of the data points are updated to improve the prediction of the $M_{i+1}$ classifier. Weights are updated such that the data points misclassified by the $M_i$ classifier are given a higher weight than other data points.

Voting

The voting (Sewell, 2008) ensemble method first generates $m$ base models by training some supervised machine-learning algorithm(s), such as decision tree and naive Bayes, on the training datasets. However, there are many ways to generate base models. For example, some base machine-learning algorithms can be trained on varying splits of the same training dataset; or varying base machine-learning algorithms can be trained on the same dataset; or some other method can be used. At the time of prediction, the output of these base models is combined to generate the final prediction. For example, the Average Vote (AV) ensemble method computes the average of the confidence score given by each base model, to compute the final score. The model compares the final score with the decision threshold value (default value is 0.5). If the final score is greater than the decision threshold, the data instance is predicted as logged; otherwise, it is predicted as non-logged. The Maximum Vote (MV) ensemble technique computes the maximum score among the confidence scores given by each base model. If the maximum score is greater than 0.5, the instance is predicted as logged; otherwise, it is predicted as non-logged. The Majority Vote (MV) ensemble method takes the majority vote of the predictions of these base models to make the prediction. If the majority of the base models predict an instance as logged, it is predicted as logged; otherwise, it is predicted as non-logged.

LOGIM MODEL

Figure 1 presents the framework of the proposed LogIm model. LogIm consists of two main phases: the model-building phase and the prediction phase. In the model-building phase, the authors build various LogIm models, which are then used for prediction in the prediction phase. There are four main steps in the model-building phase: training instance collection, feature extraction, preprocessing, and LogIm model-building (refer to Step 1 to Step 4 in Figure 1). In the prediction phase, LogIm models predict the label of code constructs in the test dataset (refer to Step 5 in Figure 1). Algorithm 1 presents the sequence of operations performed by the LogIm model (refer to Table 1 for information about notations used in Algorithm 1).

Model Building

Step 1 (Training Instances Collection)

Our framework takes historical data instances with known labels as input. Hence, in Step 1, the authors collect all the training instances ($TR$) (refer to line 4 in Algorithm 1). Data instances can be either catch-blocks or if-blocks.
Step 2 (Feature Extraction)

In Step 2, the authors perform feature extraction (refer to line 5 in Algorithm 1). The authors extract all the features proposed by Lal & Sureka (2016) and Lal et al. (2016) for catch-blocks and if-blocks logging prediction, respectively, as initial features ($FV_{TR}^{I}$). They propose 46 and 28 features for catch-blocks and if-blocks logging prediction, respectively. They categorize features based on type, domain, and class. Type identifies whether a feature is numerical, Boolean, or textual. Domain defines part of the source code from which a particular feature is extracted. There are three domains: method_bt (for catch-block) or method_bi (for if-block), try-catch (for catch-blocks) or if (for if-blocks), and other (for both types of code construct). Class identifies whether a feature is positive or negative. Positive features are useful in logged code construct prediction, whereas negative features are useful in non-logged code construct prediction. Table 2 presents the listing of all the features. It categorizes the features based on their type, and also presents their domain and class.
Step 3 (Preprocessing)

In Step 3 the authors apply several preprocessing techniques to clean the initial feature vector (refer to line 6 in Algorithm 1), applying four preprocessing steps to textual features. First, the authors separate all the terms combined using camel casing. For example, ‘illegalArgument’ is converted to ‘illegal’ and ‘Argument.’ Second, the authors apply stemming to the terms, in order to reduce inflected terms to their root form. For example, terms ‘running’ and ‘runs’ will be converted to their root form ‘run’ after applying stemming. Third, the authors convert all the terms to lower case. Fourth, the authors convert all the terms to their tf-idf representation. The authors combine tf-idf representation of textual features with numerical and Boolean features, to create the final feature vector ($F_{F}$). The authors use Python NLTK (“Natural Language Toolkit,” n.d.) library for the first three steps, and WEKA for the fourth step. In addition to the preprocessing techniques used in this work, feature-selection techniques can also be used to reduce the dimensionality of the feature vector. The authors tested
several feature-selection techniques (such as information gain and gain ratio), but did not find much improvement in the results; hence, the authors do not apply any feature-selection techniques in this work.

**Step 4 (LogIm Model Creation)**

In Step 4, the authors create various LogIm models (refer to line 7 in Algorithm 1). The authors use three machine-learning algorithms (J48, RF, and SVM) for base classifier learning. These base classifiers are combined with six ensemble techniques for final LogIm model creation. The authors use bagging, boosting, stacking, maximum vote, average vote, and majority vote ensemble techniques with these base classifiers, and create ten LogIm models. The authors create the LogIm Stacking (LST) model by using stacking, an ensemble technique of RF, J48, and SVM classifiers, with logistic regression as a meta-classifier. The authors create the LogIm Maximum Vote (LMXV) model by using maximum vote ensemble technique on RF, J48, and SVM classifiers. Similarly, the authors create LogIm Average Vote (LAV) and LogIm Majority Vote (LMV) models using average vote and majority vote ensemble technique on base classifiers. The authors create LogIm Bagging J48 (LBAJ), LogIm Bagging RF (LBAR), and LogIm Bagging SVM (LBAS) models by applying bagging ensemble technique to J48, RF, and SVM classifiers, respectively. The authors create LogIm Boosting J48 (LBOJ), LogIm Boosting RF (LBOR), and LogIm Boosting SVM (LBOS) models by applying boosting ensemble technique to J48, RF, and SVM classifiers, respectively. Hence, the authors create a total of ten LogIm models (LogIm\_Models).

**Prediction Phase**

**Step 5 (Prediction)**

In the prediction phase, LogIm models are used to predict labels of the instances present in the test dataset \( FV_{TS} \). The authors extract all three types of features (mentioned in the Step 2) from the target instance, and apply all the preprocessing steps. In addition to the preprocessing steps mentioned in Step 3, the authors apply one additional step. In the test instances, the authors filter features that were not present in the training data (refer to line 28 in Algorithm 1). Thus, the authors obtain the final feature vector for test instances \( FV_{TR} \). The authors then apply all ten LogIm models to the test dataset, and report the best value for each LogIm model with the respective decision threshold (refer to line 16 in Algorithm 1).

**EXPERIMENTAL DETAILS**

In this section, the authors discuss details related to the experiments performed in this work.

**Experimental Dataset Selection**

The authors conducted the logging-prediction experiment on open-source projects from the Apache Software Foundation (ASF). ASF consists of a large number of actively maintained projects, leading the authors to think that projects from ASF would have good logging. In addition, experiments on open-source projects would facilitate replication of the work presented in this paper by other researchers. Table 9 in the Appendix shows the four criteria that the authors consider while selecting projects for logging-prediction study. The authors select Java projects, as Java is one of the most widely used languages (Krill, 2015). In addition, the authors select projects consisting of at least 1,000 files of both catch-blocks and if-blocks, in order to draw statistically significant conclusions about the result. Three projects match the criteria: Tomcat, CloudStack, and Hadoop. The Tomcat (Apache Tomcat, n. d.) Web server implements many Java EE specifications. CloudStack (Apache CloudStack, n. d.) provides public, private, and hybrid cloud solutions. The Hadoop (Apache Hadoop, n.d.) framework provides distributed processing functionalities for big data. All three projects are actively maintained and widely used.
Table 2. Features used for catch-blocks and if-blocks logging prediction. Taken from previously published work by Lal and Sureka, (2016) (catch-block features) and Lal et al. (2016) (if-block). P,T: class = positive, domain = try/catch; P,M: class = positive, domain = method_bt; P,O: class = positive, domain = other; N,T: class = negative, domain = try/catch; and N,M: class = negative, domain = method_bt.

<table>
<thead>
<tr>
<th>Type of Feature</th>
<th>Catch-Block Features</th>
<th>If-Block Features</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Textual Features</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1. Catch Exception Type (P,T)</td>
<td>1. Log Levels in Method_BT (P,I)</td>
<td></td>
</tr>
<tr>
<td>2. Log Levels in Try Block (P,T)</td>
<td>2. Operators in Method_BT (P,M)</td>
<td></td>
</tr>
<tr>
<td>3. Log Levels in Method_BT(P,M)</td>
<td>3. Variable Declaration Name in Method_BT (P,M)</td>
<td></td>
</tr>
<tr>
<td>4. Operators in Try Block(P,T)</td>
<td>4. Call Name in Method_BT (P,M)</td>
<td></td>
</tr>
<tr>
<td>5. Operators in Method_BT(P,M)</td>
<td>5. Method Parameters (Type) (P, O)</td>
<td></td>
</tr>
<tr>
<td>7. Method Parameters (Name) (P,O)</td>
<td>7. Container Package Name (P,O)</td>
<td></td>
</tr>
<tr>
<td>8. Container Package Name (P,O)</td>
<td>8. Container Class Name (P,O)</td>
<td></td>
</tr>
<tr>
<td>9. Container Class Name (P,O)</td>
<td>9. Container Method Name (P,O)</td>
<td></td>
</tr>
<tr>
<td>10. Container Method Name (P,O)</td>
<td>10. IF Expression (P,I)</td>
<td></td>
</tr>
<tr>
<td>11. Variable Declaration Name in Try Block(P,T)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>12. Variable Declaration Name in Method_BT(P,M)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>13. Method Call Name in Try Block(P,T)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>14. Method Call Name in Method_BT(P,M)</td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Numerical Features</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1. Size of Try Block <a href="P,T">LOC</a></td>
<td>1. Size of Method_BT[LOC] (P,M)</td>
<td></td>
</tr>
<tr>
<td>2. Size of Method_BT<a href="P,M">LOC</a></td>
<td>2. Log Count in Method_BT (P,M)</td>
<td></td>
</tr>
<tr>
<td>7. Variable Declaration Count in Try Block(P,T)</td>
<td>7. IF Count in Method_BT (P,I)</td>
<td></td>
</tr>
<tr>
<td>8. Variable Declaration Count in Method_BT(P,M)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>9. Method Call Count in Try Block(P,T)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>10. Method Call Count in Method_BT(P,M)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>11. Method Parameter Count(P,O)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>12. IF Count in Try Block(P,T)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>13. IF Count in Method_BT(P,M)</td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Boolean Features</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1. Previous Catch Blocks(P,T)</td>
<td>1. Logged Method_BT (P,M)</td>
<td></td>
</tr>
<tr>
<td>2. Logged Previous Catch Blocks (P,T)</td>
<td>2. Method have Parameter (P, O)</td>
<td></td>
</tr>
<tr>
<td>3. Method have Parameter (P, O)</td>
<td>3. IF in Method_BT (P,M)</td>
<td></td>
</tr>
<tr>
<td>4. Logged Try Block(P,T)</td>
<td>4. Null Condition (P, I),</td>
<td></td>
</tr>
<tr>
<td>5. Logged Method_BT(P,M)</td>
<td>5. InstanceOf Condition (P,I)</td>
<td></td>
</tr>
<tr>
<td>6. IF in Try(P,T)</td>
<td>6. Throw/Throws in IF Block (N,I)</td>
<td></td>
</tr>
<tr>
<td>7. IF in Method_BT(P,M)</td>
<td>7. Throw/Throws in Method_BT (N,M)</td>
<td></td>
</tr>
<tr>
<td>8. Throw/Throws in Try Block(N,T)</td>
<td>8. Return in IF Block (N,I)</td>
<td></td>
</tr>
<tr>
<td>9. Throw/Throws in Catch Block(N,T)</td>
<td>9. Return in Method_BT (N,M)</td>
<td></td>
</tr>
<tr>
<td>10. Throw/Throws in Method_BT(N,M)</td>
<td>10. Assert in Method_BT (N,M)</td>
<td></td>
</tr>
<tr>
<td>11. Return in Try Block(N,T)</td>
<td>11. Assert in IF Block (N,I)</td>
<td></td>
</tr>
<tr>
<td>12. Return in Catch Block(N,T)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>13. Return in Method_BT(N,M)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>14. Assert in Try Block(N,T)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>15. Assert in Catch Block(N,T)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>16. Assert in Method_BT(N,M)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>17. Thread.Sleep in Try Block(N,T)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>18. InterruptedException Type(N,T)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>19. Exception Object “Ignore” in Catch(N,T)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Total Features =</td>
<td>Textual (14) + Numeric (13)+ Boolean(19) = 46 Feature</td>
<td>Textual (10)+ Numeric (7)+ Boolean (11) = 28 Features</td>
</tr>
</tbody>
</table>
Experimental Dataset Preparation

The authors extract all the target code constructs (catch-blocks and if-blocks) from the three projects. A code construct is marked as logged if it consists of at least one logging statement; otherwise, it is marked as non-logged. Authors count the log statements inserted using libraries, rather than ad hoc logging consisting of print statements. This shows that open source projects enforce and monitor usage of logging libraries. Since developers from varying projects prefer varying formats for writing log statements, the authors use a total of 26 regular expressions (Lal, Sardana, & Sureka, 2015a) to extract all log statements. Table 10 in the Appendix shows the details of the experimental dataset used in this work. These three projects consist of 19,303 catch-blocks and 114,526 if-blocks. Table 10 illustrates that all three projects have imbalanced dataset distribution for both catch-blocks and if-blocks, as the number of non-logged code constructs is much higher than that of logged code constructs. For example, less than 11% of if-blocks are logged in all three projects. The authors observed a variation in terms of the percentage of catch-blocks and if-blocks logged across the three projects. This shows that the amount of information required for diagnostics and auditing varies across software systems.

Experimental Design

The authors created a 70-30 training-testing dataset split to conduct logging prediction experiments using random sampling. At the time of random-split creation, the authors kept the ratio of logged and non-logged code constructs the same in both training and testing. Results obtained by conducting experiments on a random sample can have biases. Hence, the authors created 10 random samples, and report average results to reduce bias in the results.

Evaluation Metrics

The authors use four metrics—precision, recall, accuracy, and F-measure—to evaluate the effectiveness of classifiers. Computing the F-measure requires precision and recall metrics. Since the imbalanced data and accuracy metric is biased toward data distribution, the authors use F-measure to compare the performance of various classifiers. At the time of logging prediction, four outcomes were possible:

1. **True Positive (TP):** A logged code construct is predicted as logged (l→l);  
2. **False Negative (FN):** A logged code construct is predicted as non-logged (l→n);  
3. **True Negative (TN):** A non-logged code construct is predicted as non-logged (n→n);  
4. **False Positive (FP):** A non-logged code construct is predicted as logged (n→l).

To compute various metrics, the authors count the total number of true positives ($C_{TP}$), false negatives ($C_{FN}$), true negatives ($C_{TN}$), and false negatives ($C_{FP}$). Following are the details of various metrics:

- **Logged precision:** Percentage of predicted logged code constructs that are actually labeled as logged:

\[
Logged\ Precision (LP) = \frac{C_{TP} * 100}{C_{TP} + C_{FP}} \tag{1}
\]

- **Logged recall:** Percentage of labeled logged code constructs that are predicted as logged:

\[
Logged\ Recall (LR) = \frac{C_{TP} * 100}{C_{TP} + C_{FN}} \tag{2}
\]
- **Logged F-measure**: The weighted average of precision and recall metrics. In this work, we have used the value of weight as 1, giving equal weight to precision and recall:

\[
\text{Logged } F - \text{measure } (LF) = \frac{2 \times LP \times LR \times 100}{LP + LR}
\]  

(3)

- **Accuracy**: Correctly predicted (logged or non-logged) code constructs as a percentage of the total number of code constructs:

\[
\text{Accuracy } (ACC) = \frac{C_{TP} + C_{TN} \times 100}{C_{TP} + C_{FN} + C_{TN} + C_{FP}}
\]  

(4)

**EXPERIMENTAL RESULTS**

Following are answers to the six research questions, including motivation, approach, and results of each.

**RQ 1: What is the Performance of Base Classifiers for Logging Prediction on Imbalanced Datasets?**

**Motivation**

In RQ 1, the authors compare performances of the three base classifiers to identify the best classifier for logging prediction on imbalanced datasets. The answer to this research question can provide important insights about choosing the best classifier for further improving logging prediction performance on imbalanced datasets.

**Approach**

To answer RQ 1, the authors compare the catch-blocks and if-blocks logging prediction performances of J48, RF, and SVM classifiers, using four metrics (i.e., LP, LR, LF, and ACC) for comparison.

**Results**

Table 3 and Table 4 present the LP, LR, LF, and ACC values for catch-blocks and if-blocks logging prediction, using base classifiers for all three projects. For catch-blocks, SVM classifier gives the highest average LF and LR, 76.09% and 75.13% respectively (refer to Table 3). For if-blocks, J48 classifier gives the highest average LF and LR of 50.18% and 42.60% respectively (refer to Table 4). RF classifier gives the highest average LP of 83.88% and 76.99% for catch-blocks and if-blocks logging prediction, respectively. SVM classifier gives the highest average ACC of 86.71% for catch-blocks logging prediction, whereas RF classifier gives the highest average ACC of 93.99% for if-blocks logging prediction. These results show that no dominant classifier gives the best results for both catch-blocks and if-blocks logging prediction. In addition, results indicate that classifiers are complementary to each other and, if combined in some manner, may improve the logging prediction results. Hence, in this work the authors use all three classifiers with multiple ensemble techniques to improve the logging prediction performance on an imbalanced dataset.

**Note**

SVM and J48 classifier gives the highest LF for catch-blocks and if-blocks logging prediction, respectively. The authors do not observe any dominant classifier that gives the best results for both if-blocks and catch-blocks logging prediction. However, varying classifiers provide complementary information for logging prediction.
RQ 2: What Impact do Various Decision Threshold Boundaries Have on Logging Prediction Performance of Base Classifiers on Imbalanced Datasets?

Motivation

In RQ 2, the authors compute the classification performance at various threshold values to identify better threshold boundaries for imbalanced datasets. At the time of classification, classifiers produced a score between [0, 1.0]. An instance is classified as negative if its classification score is less than 0.5; otherwise, it is classified as positive. A threshold boundary of 0.5 is suitable for balanced datasets. However, for imbalanced datasets, it may not be appropriate. Hence, for this RQ, the authors explore effects of various threshold boundaries on prediction performances of base classifiers, in the context of logging prediction on imbalanced datasets.
Approach

To answer RQ 2, the authors compute the logging prediction performances of base classifiers at various decision threshold boundaries. The authors vary the threshold values between 0.1 to 0.9, using a step size of 0.1 and computing the prediction performance at each value.

Result

Figure 2 and Figure 3 present the results of catch-blocks and if-blocks logging prediction, respectively, at various decision threshold boundaries. Using various decision threshold boundaries for both catch-blocks and if-block logging prediction most affects prediction performance of the RF classifier. For example, the RF classifier gives LF values of 59.99% for catch-blocks logging prediction on the Hadoop project at a threshold of 0.5. The LF values increase to 69.32%, an increase of 9.42%, at a threshold value of 0.3. Similarly, for if-blocks logging prediction, the RF classifier gives the highest LF of 50.96% at threshold value 0.3, an increase of 13.64% in the LF value at threshold 0.5. Performance of the J48 classifier is least affected by application of various threshold boundaries for both catch-blocks and if-blocks logging prediction. This indicates that application of various threshold boundaries can improve logging prediction performances of some classifiers for imbalanced datasets.

Note

RF classifier is affected the most by using different decision threshold boundaries.
RQ 3: What are the Performances of LogIm Models for Logging Prediction on Imbalanced Datasets?

Motivation

In RQ 3, the authors investigate the performances of ten LogIm models for catch-blocks and if-blocks logging prediction on imbalanced datasets. The answer to this RQ can provide important insights about the effectiveness of ensemble techniques with various threshold boundaries for logging prediction on imbalanced datasets.

Approach

To answer RQ 3, the authors compute LF values of all ten ensemble-based LogIm models (i.e., LAV, LMV, LMXV, LST, LBAJ, LBAR, LBAS, LBOJ, LBOR, and LBOS). For each LogIm model, the authors compute LF values at all nine threshold values from 0.1 to 0.9, and report the best results. The authors compare results of the LogIm models with the three baseline classifiers.

Results

Table 5 and Table 6 present the catch-blocks and if-blocks logging prediction results for all ten LogIm models. Figure 4 and Figure 5 present the improvements in the average LF values obtained by all ten LogIm models, as compared to the baseline classifiers. For both catch-blocks and if-blocks logging prediction, LBAR models give the best results. The LBAR model gives the highest average LF of 80.65%, an improvement of 4.56% as compared to the SVM classifier for catch-blocks logging prediction (refer to Table 5 and Figure 4c). For if-blocks logging prediction, the LBAR classifier gives the highest average LF of 62.29%, an improvement of 12.11% as compared to the J48 classifier (refer to Table 6 and Figure 4a). Other LogIm models (except LBOS) also show considerable improvements in the LF values, compared to all three baseline classifiers for both catch-blocks and if-blocks logging prediction. Even though LBOS performs poorly compared to SVM (for catch-blocks) and J48 (for if-blocks), it performs better than other base classifiers. Table 5 and Table 6 also show that most of the LogIm models give highest LF at threshold values less than 0.5. Hence, the authors infer that an ensemble and threshold-based model is effective in improving the catch-blocks and if-blocks logging prediction performances on imbalanced data.

<table>
<thead>
<tr>
<th>S.No</th>
<th>LogIm Model</th>
<th>TH</th>
<th>Tomcat (LF%)</th>
<th>TH</th>
<th>CloudStack (LF%)</th>
<th>TH</th>
<th>Hadoop (LF%)</th>
<th>AVG.</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>LAV</td>
<td>0.4</td>
<td>78.6 ± 1.04</td>
<td>0.5</td>
<td>87.86 ± 0.85</td>
<td>0.4</td>
<td>71.33 ± 0.84</td>
<td>79.26</td>
</tr>
<tr>
<td>2</td>
<td>LMV</td>
<td>0.8</td>
<td>77.38 ± 0.71</td>
<td>0.9</td>
<td>87.9 ± 0.73</td>
<td>0.9</td>
<td>68.94 ± 0.92</td>
<td>78.07</td>
</tr>
<tr>
<td>3</td>
<td>LMXV</td>
<td>0.4</td>
<td>77.51 ± 1.43</td>
<td>0.4</td>
<td>87.12 ± 0.58</td>
<td>0.4</td>
<td>70.46 ± 0.9</td>
<td>78.36</td>
</tr>
<tr>
<td>4</td>
<td>LST</td>
<td>0.4</td>
<td>79.15 ± 1.15</td>
<td>0.3</td>
<td>88.67 ± 0.58</td>
<td>0.3</td>
<td>72.94 ± 0.91</td>
<td>80.25</td>
</tr>
<tr>
<td>5</td>
<td>LBAJ</td>
<td>0.4</td>
<td>77.35 ± 1.66</td>
<td>0.5</td>
<td>87.82 ± 0.61</td>
<td>0.3</td>
<td>71.72 ± 0.75</td>
<td>78.96</td>
</tr>
<tr>
<td>6</td>
<td>LBAR</td>
<td>0.3</td>
<td>79.14 ± 0.97</td>
<td>0.5</td>
<td>88.83 ± 0.54</td>
<td>0.3</td>
<td>73.97 ± 0.54</td>
<td>80.65</td>
</tr>
<tr>
<td>7</td>
<td>LBAS</td>
<td>0.4</td>
<td>77.94 ± 1.42</td>
<td>0.3</td>
<td>85.41 ± 0.61</td>
<td>0.4</td>
<td>68.67 ± 1.06</td>
<td>77.34</td>
</tr>
<tr>
<td>8</td>
<td>LBOJ</td>
<td>0.1</td>
<td>78.73 ± 1.62</td>
<td>0.2</td>
<td>88.68 ± 0.54</td>
<td>0.1</td>
<td>72.96 ± 1.08</td>
<td>80.12</td>
</tr>
<tr>
<td>9</td>
<td>LBOR</td>
<td>0.1</td>
<td>76.32 ± 1.94</td>
<td>0.1</td>
<td>89.42 ± 0.63</td>
<td>0.1</td>
<td>70.43 ± 1.05</td>
<td>78.72</td>
</tr>
<tr>
<td>10</td>
<td>LBOS</td>
<td>0.2</td>
<td>74.76 ± 1.7</td>
<td>0.1</td>
<td>83.29 ± 0.5</td>
<td>0.1</td>
<td>65.67 ± 0.91</td>
<td>74.57</td>
</tr>
</tbody>
</table>
Note

LBAR model gives the highest average LF values for both catch-blocks and if-blocks logging prediction on imbalanced dataset. Overall, LogIm models are effective in improving the catch-blocks and if-blocks logging prediction performance.

Table 6. Performances of LogIm models for if-blocks logging prediction: TH:Threshold, AVG:Average

<table>
<thead>
<tr>
<th>S.No</th>
<th>LogIm Model</th>
<th>TH</th>
<th>Tomcat (LF%)</th>
<th>TH</th>
<th>CloudStack (LF %)</th>
<th>TH</th>
<th>Hadoop (LF %)</th>
<th>AVG.</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>LAV</td>
<td>0.3</td>
<td>55.18 ± 1.11</td>
<td>0.3</td>
<td>68.38 ± 0.83</td>
<td>0.2</td>
<td>50.53 ± 0.77</td>
<td>58.03</td>
</tr>
<tr>
<td>2</td>
<td>LMV</td>
<td>0.3</td>
<td>47.32 ± 1.27</td>
<td>0.9</td>
<td>64.18 ± 1</td>
<td>0.5</td>
<td>39.39 ± 1.18</td>
<td>50.30</td>
</tr>
<tr>
<td>3</td>
<td>LMXV</td>
<td>0.3</td>
<td>54.88 ± 1.18</td>
<td>0.3</td>
<td>68.21 ± 0.59</td>
<td>0.3</td>
<td>50.31 ± 0.62</td>
<td>57.80</td>
</tr>
<tr>
<td>4</td>
<td>LST</td>
<td>0.2</td>
<td>57.55 ± 1.69</td>
<td>0.2</td>
<td>70.72 ± 0.99</td>
<td>0.2</td>
<td>52.47 ± 1.05</td>
<td>60.25</td>
</tr>
<tr>
<td>5</td>
<td>LBAJ</td>
<td>0.2</td>
<td>54.18 ± 1.52</td>
<td>0.3</td>
<td>68.57 ± 0.63</td>
<td>0.3</td>
<td>50.41 ± 1.29</td>
<td>57.72</td>
</tr>
<tr>
<td>6</td>
<td>LBAR</td>
<td>0.2</td>
<td>59.65 ± 1.16</td>
<td>0.3</td>
<td>71.97 ± 0.97</td>
<td>0.2</td>
<td>55.25 ± 0.55</td>
<td>62.29</td>
</tr>
<tr>
<td>7</td>
<td>LBAS</td>
<td>0.3</td>
<td>49.8 ± 1.45</td>
<td>0.3</td>
<td>61.71 ± 0.81</td>
<td>0.3</td>
<td>42.89 ± 1.43</td>
<td>51.47</td>
</tr>
<tr>
<td>8</td>
<td>LBOJ</td>
<td>0.1</td>
<td>59.01 ± 1.46</td>
<td>0.2</td>
<td>71.38 ± 1.62</td>
<td>0.1</td>
<td>52.5 ± 1.52</td>
<td>60.96</td>
</tr>
<tr>
<td>9</td>
<td>LBOR</td>
<td>0.1</td>
<td>56.81 ± 1.98</td>
<td>0.1</td>
<td>69.53 ± 1.02</td>
<td>0.1</td>
<td>48.88 ± 1.73</td>
<td>58.41</td>
</tr>
<tr>
<td>10</td>
<td>LBOS</td>
<td>0.3</td>
<td>47.27 ± 1.74</td>
<td>0.3</td>
<td>57.65 ± 0.63</td>
<td>0.2</td>
<td>39.1 ± 1.12</td>
<td>48.01</td>
</tr>
</tbody>
</table>

Figure 4. Improvement in LF values for catch-blocks logging prediction by LogIm models as compared to the baseline classifiers

Figure 5. Improvement in LF values for if-blocks logging prediction by LogIm models as compared to the baseline classifiers
RQ 4: What is Logging Prediction Performance of LogIm Models Within the Context of Various Projects?

Motivation

In RQ 4, the authors investigate performance of different LogIm models with respect to projects used for logging prediction. The answer to this RQ can provide important insights about dataset characteristics with respect to logging prediction—i.e., some classifiers may give better prediction results on some projects.

Approach

To answer RQ 4, the authors compute LF for all ten LogIm models for each project, and make box-and-whisker plots, a convenient and simple method of analyzing groups of numerical data. Box plots are also useful in identifying outliers and comparing distributions.

Result

Figure 6 presents the Box-Whisker plot for catch-blocks and if-blocks logging prediction performances of LogIm models for all three projects. Figure 6b shows that all the LogIm models give better overall performance for the CloudStack project than for the Tomcat and the Hadoop projects, for both catch-blocks and if-blocks logging prediction. For example, for catch-blocks logging prediction, LogIm models give a median LF of 87.9% for the CloudStack project, compared to median values of 77.7% and 70.9% for the Tomcat and the Hadoop projects respectively (refer to Figure 6a). Overall, all the LogIm models give higher values for catch-blocks logging prediction than for if-blocks logging prediction. For example, for the Hadoop project, the authors achieve the median LF value of 70.9% for catch-blocks logging prediction, compared to 50.4% for if-blocks logging prediction (refer to Figure 6a and Figure 6b).

Note

All the LogIm classifier give better performances for catch-blocks logging prediction as compared to if-blocks logging prediction. LogIm models give the best results on CloudStack project for both catch-blocks and if-blocks logging prediction.
RQ 5: What Impact do Various Decision Threshold Boundaries Have on Logging Prediction Performance of LogIm Models on Imbalanced Datasets?

Motivation

In RQ 5, the authors compute performance of various LogIm models at various threshold points. The answer to this RQ will provide important insights about LogIm models that are more susceptible to changes in decision threshold boundary. It will also provide insight into changes (if any) in the nature of base classifiers combined with ensemble techniques, with respect to changes in decision threshold boundary for logging prediction.

Approach

To answer RQ 5, the authors compute the logging prediction performance of all ten LogIm models at various decision threshold boundaries from 0.1 to 0.9, with a step size of 0.1, computing the prediction performance at each threshold value.

Result

Figure 7 and Figure 8 present the catch-blocks and if-blocks logging prediction performances, respectively, of all the LogIm models. For each LogIm model, the authors plot its LF values for the three projects. Figure 7 and Figure 8 illustrate that six LogIm models (LAV, LMXV, LST, LBAJ, LBAR, and LBAS) show deviation with respect to various threshold values for catch-blocks and if-blocks logging prediction. The LBAR model shows the highest standard deviation of 24.03 in the LF values for catch-blocks logging prediction for the Tomcat project. Bagging-based LogIm models (LBAJ, LBAR, LBAS) show higher variation in the logging prediction performance than boosting-based models (LBOJ, LBOR, LBOS) for both catch-blocks and if-blocks. Deviations in the performance of models are project-independent—i.e., some models show deviation with respect to all projects, while some models do not show any deviation.

Figure 7. Performances of LogIm models at various threshold values for catch-blocks logging prediction
Bagging based LogIm models give higher variation in the logging prediction performances as compared to boosting based LogIm models.

**RQ 6: What is the Time Complexity of Various LogIm Models?**

**Motivation**

In RQ 6, the authors compare the prediction time of various LogIm models with the baseline classifiers. Analyzing prediction time of various classifiers is important because a model that gives good prediction results with a very high prediction time will be impractical in the real world. The authors compare the prediction time, but not the model-building time, since models can be built offline.

**Approach**

The authors compute performances of the LogIm prediction models at all threshold values, and report the threshold that gives the best results. For LogIm models, the authors compute the time taken at each threshold and report the average results. These results are further averaged over 10 test sets used in this work. For baseline classifiers, the authors report average prediction time on the 10 datasets.

**Result**

Table 7 shows the time taken by various classifiers for catch-blocks and if-blocks logging prediction. Table 7 shows that base classifiers took less time than other LogIm models. The LogIm model is suitable for real world uses, as it can label thousands of catch-blocks and if-blocks in only few seconds.
The prediction time of LogIm models is higher as compared to base classifiers but it is still acceptable as it can label thousands of catch-blocks and if-blocks in few seconds.

**Overall Performance Summary of the LogIm Models**

Table 8 shows the performance summary of LogIm models, presenting the average improvements in LF achieved by all the LogIm models, the highest LF, and the maximum average standard deviation. In this table, * indicates the highest LF, √ indicates increase, and × indicates decrease in the logging prediction performance, as compared to the best baseline classifier. The LBAR model gives the highest LF and the highest standard deviation for both catch-blocks and if-blocks logging prediction. All the LogIm classifiers give the highest LF for the CloudStack project. Bagging-based LogIm models give higher LF than boosting-based LogIm models.

**DISCUSSION**

**Improving Logging Quality in OSS Development Process Using LogIm**

The proposed log statement prediction model, LogIm, can be applied to improve both the open-source software system and processes. As an open-source software system evolves and becomes widespread, new developers interested in the project join, taking responsibility for writing code for various parts of the system. The LogIm model can capture and learn the logging behavior from the source code, and serve as a recommendation engine or guide for developers new to the project. New developers can get guidance on where to log from the logging prediction tool, based on past logging practices and behaviors in the source code. Improving the effectiveness and efficiency of new developers within a voluntary-contribution-based OSS context, in terms of inserting log statements optimally within the source code, will lead to improved quality of the software system.

The LogIm model can also improve the source code commit and release process. The LogIm model can serve as a static code analysis tool, generating warnings for source-code constructs that
should be logged and traced during program execution. The OSS project team can set process guidelines to ensure that all warnings must be removed or discussed before the source code is released or versioned. A logging prediction tool can help in improving OSS processes, as the project team can set guidelines on not committing or shipping code that does not have enough logging at the right locations, as suggested by the LogIm tool.

Effectiveness of LogIm for Logging Prediction

The proposed LogIm tool can be beneficial in logging prediction in the real world, where the data is often imbalanced, and applying appropriate sampling technique can be tricky for model learning. The experimental results on three OSSs show that the LogIm model is more effective in logging prediction for both catch-blocks and if-blocks logging on imbalanced datasets as compared to the baseline classifiers. The LBAR model gives average LF of 80.65% for catch-blocks logging prediction, and average LF of 62.29% for if-blocks prediction. The LBAR model gives respective improvement of 4.6% and 12.11% for in catch-blocks and if-blocks logging prediction as compared to the best baseline classifier. Hence, if the LogIm model is implemented in the real world setting it can be beneficial in improving the logging quality in the source code. The time the LogIm model takes for prediction is reasonable, as it can predict logging for thousands of code constructs in a few seconds. Hence, the proposed model can be easily integrated into source-code development IDE, where it can be used to provide logging recommendations to the software developers without adding much overhead in IDE performance.

Potential Improvements and Future Directions

To improve the quality of logging in OSS and to help software developers make strategic logging decisions, as a first step the authors propose the LogIm model in this study. The proposed LogIm model can be extended in multiple ways to further improve the logging quality in OSS. The LogIm model predicts logging on two types of code constructs: catch-blocks and if-blocks. The proposed model can be extended to other types of code constructs, such as switch cases and while loop. As an initial step toward logging, the LogIm model predicts only one aspect of logging i.e., code constructs that must be logged. LogIm can be integrated with other logging automation tools. Yuan et al. (2012c)
propose the LogEnhancer tool to enhance the content of current logging statements by adding more information. Li et al. (2016a) propose a method for verbosity level prediction of log statements. The LogIm model can easily be integrated with these models, and a comprehensive tool can be developed to predict many aspects of logging. Currently, the LogIm model learns the logging prediction features statically from the code. The LogIm model can be improved to take logging feedback dynamically from the software developers.

THREATS TO VALIDITY

1. **Number and Types of Projects:** The authors conduct all experiments on only three projects, i.e., Tomcat, CloudStack, and Hadoop. These three are open-source Java-based projects. Hence, the authors cannot draw any general conclusion that is applicable to logging prediction on all types of software projects. Other types of software projects, such as closed source, must be evaluated to test the generalizability of the model proposed in this work;

2. **Quality of Ground Truth:** The authors use 26 regular expressions to extract all the logging statements from catch-blocks and if-blocks. Manual analysis of a few code constructs reveals that the authors extracted all logging statements. However, there is still a possibility that the authors missed some types of logging statement in the source code. Also, there is a possibility of errors or non-optimal logging in the source code by the software developers, which can affect results of this study. However, since all three software projects used in this work are long lived and actively maintained, the authors can assume that the code constructs have good logging;

3. **Classification Algorithms:** The authors use only three base classifiers with six ensemble techniques. The authors selected these three base classifiers because previous studies reported that these classifiers give the best results for logging prediction. Performance of more classifiers must be evaluated to identify the best classifier and the best ensemble technique. Also, the authors use default parameters for all classifiers. Parameter tuning is required to further improve prediction performance;

4. **Dataset Sampling:** At the time of dataset sampling, the authors kept the ratio of logged and non-logged code constructs the same in the training and testing datasets. This can cause bias in the results. However, the authors created 10 such training and testing samples, and report average results to minimize bias in the results;

5. **Time Complexity:** The authors compute the time all base classifiers and LogIm models take to analyze their time complexities. Note that the authors use WEKA implementation of these algorithms, and a Windows server computed these time complexities. Hence, type of implementation used in the WEKA tool and other system-related factors could affect time taken by each algorithm.

CONCLUSION

This study addresses improving machine-learning-based logging prediction on imbalanced datasets. The authors first analyze the performances of three base classifiers (J48, RF, and SVM) in the context of imbalanced catch-blocks and if-blocks logging prediction on imbalanced datasets. The authors propose LogIm, an ensemble and decision-threshold-based framework for catch-blocks and if-blocks logging prediction on imbalanced datasets. The authors use three base classifiers (J48, RF, and SVM) and six ensemble techniques (average vote, maximum vote, majority vote, stacking, bagging, and boosting). Combining the base classifiers with ensemble techniques and generating ten LogIm models (LAV, LMV, LMXV, LST, LBAJ, LBAR, LBAS, LBOJ, LBOR, and LBOS), the authors evaluate performance of LogIm models on three large, open-source projects: Tomcat, CloudStack, and Hadoop.

The authors conclude that LogIm predictive models are effective in improving the catch-block and if-block logging prediction performance on imbalanced datasets in open-source software systems.
Results show that bagging and the RF-based model (i.e., LBAR) performs best for both catch-blocks and if-blocks logging prediction. LBAR gives the average LF of 80.65% and 62.29% for catch-blocks and if-blocks logging prediction, respectively. On average, LBAR improves the best baseline classifiers by 4.6% and 12.11% for catch-blocks and if-blocks logging prediction, respectively. The experimental results on the performance of base classifiers in imbalance datasets show that certain classifiers (such as SVM and J48) provide complementary information for logging prediction. Investigating the impact of various decision threshold boundaries on logging prediction performance of base classifiers on imbalanced datasets leads to the conclusion that the RF classifier is affected most by varying decision threshold boundaries. The authors conclude that there is a variance in the performance of the LogIm models with respect to open-source software systems, and the performance of LogIm models on CloudStack was the best, compared to the other systems in the experimental dataset. Investigating the impact of bagging and boosting approaches on logging prediction performances of LogIm models on imbalanced datasets results in the conclusion that bagging-based LogIm models produce greater variation in logging prediction performance than boosting-based LogIm models.

The work presented here opens doors to many interesting future directions. The authors plan to extend this work in four main dimensions. First, the authors will combine ensemble techniques with various sampling techniques to further improve the logging prediction performance on imbalanced data. Second, the authors will evaluate performance of various classifiers using other metrics, such as the ROC curve. Third, the authors will evaluate the performance of logging prediction models on other types of project, including closed source or projects written in other programming languages, such as C# and Python. Finally, the authors will work on designing a threshold-learning algorithm on a training set. Another future direction of this work is the application of deep learning techniques for the purpose of building prediction models.
REFERENCES


Saini, S., Sardana, N., & Lal, S. Logger4u: Predicting debugging statements in the source code. In *8th International Conference on Contemporary Computing (IC3)*. 2016. doi:10.1109/IC3.2016.7880255


**APPENDIX: DATA SET DETAILS**

Table 9 shows the four essential criteria that we use to select the experimental projects for this study. Table 10 shows the experimental dataset details: version, number of Java files, SLOC, log statement count, total catch-blocks, logged catch-blocks, total if-blocks, and logged if-blocks for each project.

**Table 9. Project selection criteria**

<table>
<thead>
<tr>
<th>S. No.</th>
<th>Parameter</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.</td>
<td>Programming Language</td>
<td>Java</td>
</tr>
<tr>
<td>2.</td>
<td>Number of Java Files</td>
<td>≥ 1000</td>
</tr>
<tr>
<td>3.</td>
<td>Number of Catch-Blocks</td>
<td>≥ 1000</td>
</tr>
<tr>
<td>4.</td>
<td>Number of If-Blocks</td>
<td>≥ 1000</td>
</tr>
</tbody>
</table>

**Table 10. Experimental dataset details**

<table>
<thead>
<tr>
<th>Type</th>
<th>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>No. of Java File</td>
<td>2036</td>
<td>5350</td>
<td>6331</td>
</tr>
<tr>
<td>SLOC (LocMetrics, 2007)</td>
<td>273419</td>
<td>849857</td>
<td>926644</td>
</tr>
<tr>
<td>Log Statements Count</td>
<td>2703</td>
<td>10428</td>
<td>10108</td>
</tr>
<tr>
<td>Total Catch-Blocks</td>
<td>3279</td>
<td>8077</td>
<td>7947</td>
</tr>
<tr>
<td>Logged Catch-Blocks</td>
<td>887 (27%)</td>
<td>2792(34.56%)</td>
<td>2078 (26.14%)</td>
</tr>
<tr>
<td>Distinct Exception Types</td>
<td>119</td>
<td>163</td>
<td>265</td>
</tr>
<tr>
<td>Total If-Blocks</td>
<td>16991</td>
<td>65392</td>
<td>32143</td>
</tr>
<tr>
<td>Logged If-Blocks</td>
<td>1423 (8.37%)</td>
<td>5653 (8.64%)</td>
<td>3407 (10.60%)</td>
</tr>
</tbody>
</table>
Sangeeta Lal is an Assistant Professor in the department of computer science in Jaypee Institute of Information Technology, Noida, India. She has completed her masters from IIIT-D and currently pursuing her PhD from Jaypee Institute of Information Technology under the guidance of Dr. Neetu Sardana and Dr. Ashish Sureka. She has published papers in many national and international conferences. Her research interest is in mining software repositories. She has experience in working in various domains of software repositories such as source code, bug reports, version control system, and Q&A websites. Currently she is working on log statements analysis in Java code.

Neetu Sardana is an Assistant Professor (Senior Grade) in the Department of Computer Science in Jaypee Institute of Information Technology, Noida. She has more than 14 years of teaching experience. She has published papers in many national and international conferences and journal. Her research interest is in the area of Mining Software Repositories, Social Network Analysis and Web Mining. She had supervised several M.Tech Thesis and B.Tech Major Projects. Currently she is guiding four PhD’s. She is also serving as a PC member of IC3 conference which is held every year in JIIT-Noida.

Ashish Sureka is a Principal Researcher at ABB Corporate Research Center (India). He was a Faculty Member (at IIIT-Delhi) from July 2009 to October 2014 and a visiting researcher at Siemens Corporate Research from August 2014 - July 2015. His current research interests are in the area of Mining Software Repositories, Software Analytics, and Social Media Analytics. He graduated with an MS and PhD degree in Computer Science from North Carolina State University (NCSU) in May 2002 and May 2005 respectively. He has worked at IBM Research Labs in USA, Siemens Research Lab (India) and was a Senior Research Associate at the R&D Unit of Infosys Technologies Limited before joining IIIT-D as a Faculty Member in July 2009. He has received research grants from Department of Information Technology (DIT, Government of India), Confederation of Indian Industry (CII) and Department of Science and Technology (DST, Government of India). He has published several research papers in international conferences and journals, graduated several PhD and MTech students, organized workshops co-located with conferences, and received best paper awards. He was selected for ACM India Eminent Speaker Program. He holds seven granted US patents.