e-Informatica Software Engineering Journal, Volume 11, Issue 1, 2017, pages: 7–38, DOI 10.5277/e-Inf170101

ECLogger: Cross-Project Catch-Block Logging Prediction Using Ensemble of Classifiers

Sangeeta Lal*, Neetu Sardana*, Ashish Sureka**
*Jaypee Institute of Information Technology, Noida, Uttar-Pradesh, India
**ABB Corporate Research, Bangalore, India
sangeeta@jiit.ac.in, neetu.sardana@jiit.ac.in, ashish.sureka@in.abb.com

Abstract

Background: Software developers insert log statements in the source code to record program execution information. However, optimizing the number of log statements in the source code is challenging. Machine learning based within-project logging prediction tools, proposed in previous studies, may not be suitable for new or small software projects. For such software projects, we can use cross-project logging prediction.
Aim: The aim of the study presented here is to investigate cross-project logging prediction methods and techniques.
Method: The proposed method is ECLogger, which is a novel, ensemble-based, cross-project, catch-block logging prediction model. In the research We use 9 base classifiers were used and combined using ensemble techniques. The performance of ECLogger was evaluated on on three open-source Java projects: Tomcat, CloudStack and Hadoop.
Results: $\text{ECLogger}_{\text{Bagging}}$, $\text{ECLogger}_{\text{AverageVote}}$, and $\text{ECLogger}_{\text{MajorityVote}}$ show a considerable improvement in the average Logged F-measure (LF) on 3, 5, and 4 source→target project pairs, respectively, compared to the baseline classifiers. $\text{ECLogger}_{\text{AverageVote}}$ performs best and shows improvements of 3.12% (average LF) and 6.08% (average ACC – Accuracy).
Conclusion: The classifier based on ensemble techniques, such as bagging, average vote, and majority vote outperforms the baseline classifier. Overall, the $\text{ECLogger}_{\text{AverageVote}}$ model performs best. The results show that the CloudStack project is more generalizable than the other projects.

Keywords: classification, debugging, ensemble logging, machine learning, source code analysis, tracing

1. Introduction

Logging is an important software development practice that is typically performed by inserting log statements in the source code. Logging helps to trace the program execution. In the case of failure, software developers can use this tracing information to debug the source code. Logging is important because this is often the only information available to the developers for debugging because of problems in recreating the same execution environment or because of unavailability of the input used (security/privacy concerns of the user). Logging statements have many applications, such as debugging [1] workload modelling [2], performance problem diagnosis [3], anomaly detection [4], test analysis [5,6], and remote issue resolution [7].

Source code logging is important, but it has a trade-off between the cost and the benefit [8–11]. Excessive logging in the source code can cause performance and cost overhead. It can also decrease the benefits of logging by generating too many trivial logs, which can potentially make debugging more difficult by hiding important debugging information. Excessive logging can also cause a severe performance bottleneck for a system. In a recent blog, inefficient logging was considered to be a major factor for Tomcat performance problems [12]. Similarly to excessive logging, sparse logging is also problematic. Sparse logging can make logging ineffective by missing important debugging information. Shang et al. [13] reported an experience from a user who was complaining about sparse logging of catch-blocks in Hadoop. Hence, it is important to optimize the number of logging statements in the source code. However, previous research shows that optimizing log statements in the source code is challenging, and developers often face difficulties with this task [8–11].

Several recent studies have proposed tools and techniques to help developers optimize log statements in the source code by automatically predicting the code constructs that need to be logged [8, 10, 11]. These techniques learn a prediction model from the history of the project (applying supervised learning from annotated training data) to predict logging on new code constructs. Predicting logged code constructs will work well if a sufficient amount of training data is available to train the model. However, many real-world open-source and closed-source applications and new or small projects do not have sufficient prior training data to construct the prediction model. There are several long-lived and large projects that have collected massive amounts of data. One can use training data from these project(s) (source project(s)) to predict logging on a particular project (target project) of interest, i.e. one can perform **cross-project** logging prediction. Cross-project prediction is also called transfer learning, which consists of transferring predictive models trained from one project (source project) to another project (target project). Cross-project logging prediction can have several benefits: 1) multiple projects can be used for training the model, and hence, good practices can be learned from many projects, and 2) the model can be refined offline over a period of time to improve the performance of logging prediction.

Cross-project logging prediction is an important and a technically challenging task. There are two main challenges in cross-project logging prediction: 1) vocabulary mis-match problems and 2) differences in the domain of numerical attributes. The vocabulary mis-match problem can arise due to the use of different terms in the source code of different projects. For example, the Tomcat project has 119 unique exception types, whereas the Hadoop project has 265 unique exception types. Our analysis of these exception types shows that 193 exception types present in the Hadoop project do not exist in the Tomcat project. Similarly, the domain of numerical attributes may not be the same in different projects. For example, the average SLOC of try-blocks associated with logged catch-blocks is 6.98 and 10.65 for the Tomcat and CloudStack projects, respectively. Hence, it is important to create a prediction model that uses generalized properties for cross-project logging prediction rather than domain-specific properties.

In this paper, the Authors propose **ECLogger**, a cross-project, catch-block logging prediction framework that addresses the aforementioned challenges. To address the first challenge (vocabulary mis-match problem), ECLogger performs data standardization prior to learning the model. Data standardization helps to normalize the data in a specific range and hence helps to address the problem of data heterogeneity [14]. To address the second challenge (non-uniform distribution of numerical attributes problem), ECLogger, uses an ensemble of classifiers-based approach. Ensemble-based techniques capture the strength of multiple base classifiers [15]. In this work, 9 base classifiers (AdaBoostM1, ADTree, Bayesian network, decision table, J48, logistic regression, Naïve Bayes, random forest and radial basis function network) were used. ECLogger combines these algorithms with three ensemble techniques, i.e. bagging, average vote and majority vote. 8 $\text{ECLogger}_{\text{Bagging}}$, 466 $\text{ECLogger}_{\text{AverageVote}}$ and 466 $\text{ECLogger}_{\text{MajorityVote}}$ models, i.e. a total of 940 models are created. The performance of ECLogger on three large and popular open-source Java projects: Tomcat, CloudStack and Hadoopare evaluated. The experimental results reveal that $\text{ECLogger}_{\text{Bagging}}$, $\text{ECLogger}_{\text{AverageVote}}$ and $\text{ECLogger}_{\text{MajorityVote}}$ show maximum improvements of 4.6%, 7.04% and 5.39% in the logged

F-measure, respectively, compared to the baseline classifier.

2. Related work and novel research contributions

In this section, previous works closely related to the study presented in this paper are discussed. They are organized and presented in multiple lines of research. Then the novel research contributions of this work in the context of existing work is presented.

2.1. Logging applications

Log statements present in the source code generate log messages at the time of software execution. Log statements and log messages were widely used in the past for different purposes [3, 5–7, 13, 16–18]. Shang et al. [13] used log statements present in a file to predict defects. Shang et al. proposed various product and process metrics using log statements to predict post-release defects. Nagaraj et al. [3] used good and bad logs of the system to detect performance issues in the system. Nagaraj et al. [3] developed a tool, DISTALYZER, that helps developers in finding components responsible for poor system performance. Xu et al. [18] worked on mining console logs from a distributed system at Google to find anomalies in the system. Yuan et al. [17] used log information to find the root cause of a failure. Yuan et al. developed a tool, SherLog, that can use log information to find information about failed runs without any re-execution of the code. Log messages are also helpful in fixing bugs, as the empirical study performed by Yuan et al. [1] showed that bug reports consisting of log messages were fixed 2.2 times faster compared to bug reports not consisting of log messages. Log messages are also useful in test analysis [5, 6], remote issue resolution [7], security monitoring [19], anomaly detection [4, 18], and usage analysis [20]. Many tools have also been proposed to gather log messages [21, 22]. Our work is complementary to these studies, focuses on improving logging in the catch-blocks, and can be beneficial for studies that work on analysing the log information.

2.2. Logging code analysis and improvement

Logging statements are very important in software development (refer to subsection 2.1), and hence, logging improvements have attracted attention from many researchers in the software engineering community [1, 8–11, 17, 23, 24]. Yuan et al. [25] performed a study to identify a set of generic exception types that cause most of the system failures. Yuan et al. [25] proposed a conservative approach to log all of the generic exception types. Fu et al. [8] studied the logging practices of developers on C# projects and reported the five most frequently logged code constructs. Zhu et al. [11] and Fu et al. [8] proposed a machine learning-based framework for logging prediction of exception-type and return-value-check code snippets on C# projects. Lal et al. [9, 10] proposed a machine learning-based framework for catch-block and if-block logging prediction on Java projects. All three approaches use static features from the source code for logging prediction.

Yuan et al. [24] proposed the LogEnhancer tool to help developers in enhancing the current log statements. LogEnhancer strategically identifies the variables that need to be logged, and experimental results obtained by Yuan et al. [24] showed that LogEnhancer correctly identifies the logged variables 95% of the time. In another study, Yuan et al. [1] proposed a code clone-based tool to predict the correct verbosity level of log statements. Log statements have an option to assign a verbosity level (e.g. debug, info, or trace) as an indicator of the severity level. An incorrect verbosity level to a log statement can have implications on software debugging and other related aspects [26, 27]. Kabinna et al. [23] performed a prediction on the stability (i.e. how likely a logging statement will be modified) of logging statements. Logging statements that are frequently modified may cause log processing applications to crash, and hence, timely logging stability prediction can be beneficial [23]. Our work is an extension of the logging prediction studies performed by Fu et al. [8], Zhu et al. [11], and Lal et al. [10]. In contrast to these studies, which perform within-project logging prediction, we emphasize on cross-project logging prediction.

2.3. Machine learning applications in logging

Machine learning has been found to be useful in various software engineering applications, such as logging prediction [8,10,11], performance issue diagnosis [3], defect prediction [28], and clean and buggy commit prediction [29]. Fu et al. [8] and Zhu et al. [11] applied the C4.5/J48 algorithm for logging prediction. Lal et al. [10] applied several other machine learning algorithms. These algorithms are Adaboost (ADA), decision tree, Gaussian Naïve Bayes (GNB), K-nearest neighbor (KNN), and random forest (RF)) for logging prediction. This article considers J48, ADA, Naïve Bayesian (NB), and RF for cross-project catch-block logging as the experimental results by previous studies [8, 10, 11] show that these algorithms perform better than the others. Additionally, the logistic regression (LR), Bayesian network (BN), decision table (DT), radial basis function network (RBF), and alternating decision trees (ADT) algorithms are considered in this work. These machine learning algorithms have never been explored for logging prediction but have been found to be useful in other branches of software engineering, such as defect prediction [30], software project risk prediction [31], and re-opened bug prediction [32]. The selection of these algorithms is not random or arbitrary; rather, algorithms belonging to different domains of classification algorithms were selected, for example, J48 and ADT are decision tree-based algorithms, NB and BN are probabilistic algorithms, and RBF is an artificial neural network-based algorithm.

2.4. Ensemble methods

Ensemble methods are learning algorithms that construct a prediction model from a set of base classifiers, and new data points are classified by taking a vote (weighted) of predictions made by base classifiers [33]. An ensemble consists of base classifiers that are combined in some way to predict the label of the new instance. Any base classification algorithm, such as a neural network, a decision tree or any other machine learning algorithm, can be used to generate the base classifiers from the training data. The generalization ability of an ensemble is typically considerably better than that of base classifiers [34]. Ensemble methods can use a single or multiple base classification algorithms [35–38]. Bagging [38], boosting [38], average vote [39], majority vote [39], and stacking [40] are some of the ensemble methods. Previous research shows that ensemble methods are useful in improving the performance of machine learning frameworks in various software engineering applications, such as defect prediction [15], cross-project defect prediction [30, 41], and blocking bug prediction [42]. However, ensemble methods have not been explored for cross-project logging prediction. In this work, three ensemble methods are applied, namely, bagging [38], average vote [39] and majority vote [39], to construct the cross-project logging prediction model.

2.5. Cross-project prediction

Cross-project prediction trains the model on one (or more) project(s) to make predictions on another project of interest. There are two types of cross-project prediction: supervised and unsupervised [43, 44]. The supervised techniques have some labelled instances available from the target project, whereas the unsupervised ones have all unlabelled instances fromthe target project. In the literature, cross-project prediction has been applied in various applications, such as defect prediction [14, 30, 41], build co-change prediction [45], and sentiment classification [46]. However, cross-project logging prediction is a relatively unexplored area, which is theprimary focus of this work. To the best knwoledge of the Authors, only Zhu et al. [11] have performed a basic exploration of cross-project logging prediction. This study is different from that of Zhu et al. in many aspects: 1) cross-project catch-block logging prediction is performed on Java projects, whereas Zhu et al. considered C# projects; 2) a focused and in-depth study is performed, whereas Zhu et al.performed only a basic experiment on cross-project logging prediction; and 3) an ensemble of classifiers is proposed, whereas Zhu et al. only used the J48 classifier [39] for cross-project logging prediction.

2.6. Research contributions

In context to related work, this work makes the following novel and unique research contributions.

1. A comprehensive analysis of single classifiers is performed for within-project and cross-project logging prediction. Furthermore, the performances of single-project and multi-project training models are comapred for cross-project logging prediction (refer to section 6.2).
2. ECLogger, a tool based on an ensemble of machine learning algorithms, is proposed for cross-project catch-block logging prediction on Java projects. ECLogger uses static features from the source code for cross-project catch-block logging prediction. We create 8 $\text{ECLogger}_{\text{Bagging}}$, 466 $\text{ECLogger}_{\text{AverageVote}}$ and 466 $\text{ECLogger}_{\text{MajorityVote}}$ models, i.e. a total of 940 models (refer to section 4).
3. The results of a comprehensive evaluation of ECLogger are presented on three large and popular open-source Java projects: Tomcat, CloudStack and Hadoop. The experimental results demonstrate that ECLogger is effective in improving the performance of the cross-project catch-block logging prediction (refer to section 6.3).

3. Background

In this paper, 9 base machine learning algorithms and three ensemble techniques are proposed. The following subsections provide a brief introduction to each of the 9 machine learning algorithms and the 3 ensemble techniques.

3.1. Machine learning algorithms

3.1.1. AdaBoostMI1 (ADA)

AdaBoostM1 (ADA) [47] is an extension of the simple AdaBoost algorithm for multi-class classification. There are two main steps in the ADA algorithm: boosting and ensemble creation. In the boosting phase, ADA first assigns a weight to each data point present in the database (D). Initially, all the data points are assigned an equal weight. The weights assigned to the data points are updated in subsequent iterations. In each iteration, ADA constructs a prediction model (M_i) by training some base machine learning algorithm, such as a decision tree or a neural network, on a sample (D_i) of D. In each iteration, the error rate of the model M_i is computed, and the weights of incorrectly classified data points are increased, whereas the weights of correctly classified data points are decreased. Using this strategy, ADA generates k prediction models, i.e. M_i, where $i \in \{1, 2, \ldots, k\}$. In the ensemble phase, the k models generated in the boosting phase are linearly combined. For prediction on a new instance, the weighted vote of the prediction made by these k prediction models is taken. ADA is an ensemble based algorithm. However, this work consideres default WEKA [48] implantation of ADA as a single classification algorithm in Bagging [38], Average Vote [49] and Majority Vote [49] (without the loss of generality).

3.1.2. Alternating decision tree (ADT)

The alternating decision tree (ADT) [50] is a generalization of the decision tree algorithm for classification. The ADT algorithm constructs a tree-like structure (i.e. ADT tree) for prediction. The ADT tree consists of decision nodes and prediction nodes in alternating order. Decision nodes specify a prediction condition, whereas prediction nodes consist of a single number. In the ADT tree, prediction nodes are present both as the root and as leaves. At the time of prediction, the ADT algorithm maps each data point in the ADT tree following all the paths for which decision nodes are true and summing the value of prediction nodes that are traversed. The prediction of an instance is based on the sign of the sum of the prediction values from the root to leaf, i.e. an instance is classified as logged (+ve class) if the sign is positive; otherwise, it is classified as non-logged (–ve class).

3.1.3. Bayesian network (BN)

Bayesian network (BN) [51, 52] algorithm uses a probabilistic graphical model for classification. The BN algorithm generates a probabilistic model (a directed acyclic graph (DAG)) in the training phase that is used to predict labels in the prediction phase. This model shows a probabilistic relationship or dependency between random variables. Nodes represent random variables, and edges between the nodes represent the probabilistic dependencies among the variables. In particular, a directed edge from variables X_i to X_j indicates that the value taken by the variable X_j depends on X_i. In the BN algorithm, a reasoning process can operate by propagating information in any direction, and each variable is independent of its nondescendents given the state of its parents.

3.1.4. Decision table (DT)

The decision table (DT) [53] classification algorithm consists of a decision table that is constructed in the training phase and is used to make predictions in the prediction phase. A decision table consists of two main components: schema and body [53]. The schema of the decision table consists of a set of features included in the table, and the body consists of labelled instances. In the training phase, the DT algorithm determines the set of features and labelled instances to retain in the decision table. The algorithm searches through the feature space (using the wrapper model [54]) to determine the optimal set of features that enhances prediction accuracy. Once the decision table is constructed, prediction on a new instance is performed by searching in the decision table for an exact match of the features. If there is a match, i.e. the algorithm finds some labelled instances matching the unlabelled instance, it returns the majority class of labelled instances. Otherwise, it returns the majority class present in the table.

3.1.5. J48

The J48 algorithm is an open-source implementation of the C4.5 algorithm in the WEKA tool [48]. The J48 algorithm constructs a decision tree in the training phase that is used to make predictions in the prediction phase. To create the decision tree, in each iteration, the J48 algorithm selects the attribute with the highest information gain [39], i.e. the attribute that most effectively discriminates the various data points. Now, for each attribute, the J48 algorithm finds the set of values for which there is no ambiguity among the data points regarding the class label, i.e. all data points having this value belong to the same class. It terminates this branch and assigns it the class (or label) [55].

3.1.6. Logistic regression (LR)

The logistic regression (LR) [56] model is a generalization of the linear regression model for binary classification. The LR model computes a score for each data point (Score(d_i)). If the value of Score(d_i) is greater than 0.5, the instance is predicted as logged (+ve class); otherwise, it is predicted as non-logged (–ve class). Equation (1) shows the general formula for computing the logistic regression model. In Equation (1), $\alpha, w_1, w_2, \dots w_n$ represent the linear combination coefficients, and $x_1, x_2, \dots, x_n$ represent the features used in the prediction model. The larger the value of w_i is, the larger the impact of the feature x_i is on the prediction outcome.

$$P(d_i) = \frac{e^{\alpha + w_1x_1 + w_2x_2 + \cdots + w_nx_n}}{1 + e^{w_1x_1 + w_2x_2 + \cdots + w_nx_n}} \qquad (1)$$

3.1.7. Naïve Bayes (NB)

The Naïve Bayes (NB) classifier [39] is a simple probabilistic classifier based on Bayes theorem. NB uses a feature vector and input label to generate a simple probabilistic model.

This probabilistic model is used to predict the label of an instance in the prediction phase. The NB algorithm considers each attribute to be equally important and independent [55]. NB is one of the simplest machine learning methods and is known to provide good performance in text categorization and numerical data [57,58].

3.1.8. Random forest (RF)

Random forest (RF) [36, 39] is an ensemble method that uses decision trees as the base classification algorithm. RF generates multiple decision trees using bagging [38] and random feature selection. Each decision tree is generated from the bootstrap sample of the data. At the time of tree generation, at each node, RF selects a subset of features (randomly) to split. Once all the decision trees are generated, prediction on a new instance is performed by taking the majority vote of the predictions of individual decision trees. RF is one of the fastest learning algorithms and is suitable for large datasets [39]. RF is an ensemble based algorithm. However, in this work we consider default WEKA [48] implantation of RF as a single classification algorithm in Bagging [38], Average Vote [49] and Majority Vote [49] (without the loss of generality).

3.1.9. Radial basis function network (RBF)

The radial basis function network (RBF) [59] is a type of artificial neural network that uses a radial basis as an activation function. There are three main layers in RBFNetwork, i.e. input layer, hidden layer and output layer. The input layer corresponds to the features, i.e. source code attributes. The hidden layer is used to connect the input layer to the output layer and consists of radial basis functions. The output layer performs the mapping to the outcomes to predict, i.e. logged or non-logged. The network learning is divided into two parts: first, weights are learned from the input layer to the hidden layer and then from the hidden layer to the output layer.

3.2. Ensemble techniques

3.2.1. Bagging

Bootstrap aggregating (bagging) [38] is an ensemble technique that can be combined with other supervised machine learning algorithms. Given a dataset D of size n, bagging first creates m datasets, i.e. D_i , $i \in \{1, 2, \ldots, m\}$. The size of each D_i is n_i, such that $n_i = n$. Since D_is are generated by random sampling (with replacement) from D, some data points can be missing and others can be repeated in D_i. Bagging trains a supervised machine learning algorithm, such as a decision tree, NB, or BN, on each D_i and generates m classifiers. For prediction, the output of these m classifiers is combined using majority vote. Bagging is helpful in improving the overall performance of supervised machine learning algorithms as it helps to avoid the data overfitting problem [39].

3.2.2. Voting

Voting is one of the easiest ensemble techniques. Voting first generates m base models by training some supervised machine learning algorithm(s) (base algorithm(s)), such as a decision tree, NB, or BN, on the training datasets. Base models can be generated in multiple ways, such as training some base machine learning algorithm on different splits of the same training dataset, using the same dataset with different base machine learning algorithms, or some other method. At the time of prediction, the output of these base models is combined to generate the final prediction. For example, the average vote [49] ensemble method computes the average of the confidence score given by each base model to compute the final score. The final score is then compared with a threshold value. If the confidence score is greater than the threshold value, the given instance is predicted as logged (+ve class); otherwise, it is predicted as non-logged (–ve class). Similarly, the majority vote [49] ensemble method takes the majority vote of the predictions of these base models to make the prediction, i.e. if the majority of the base models predict an instance

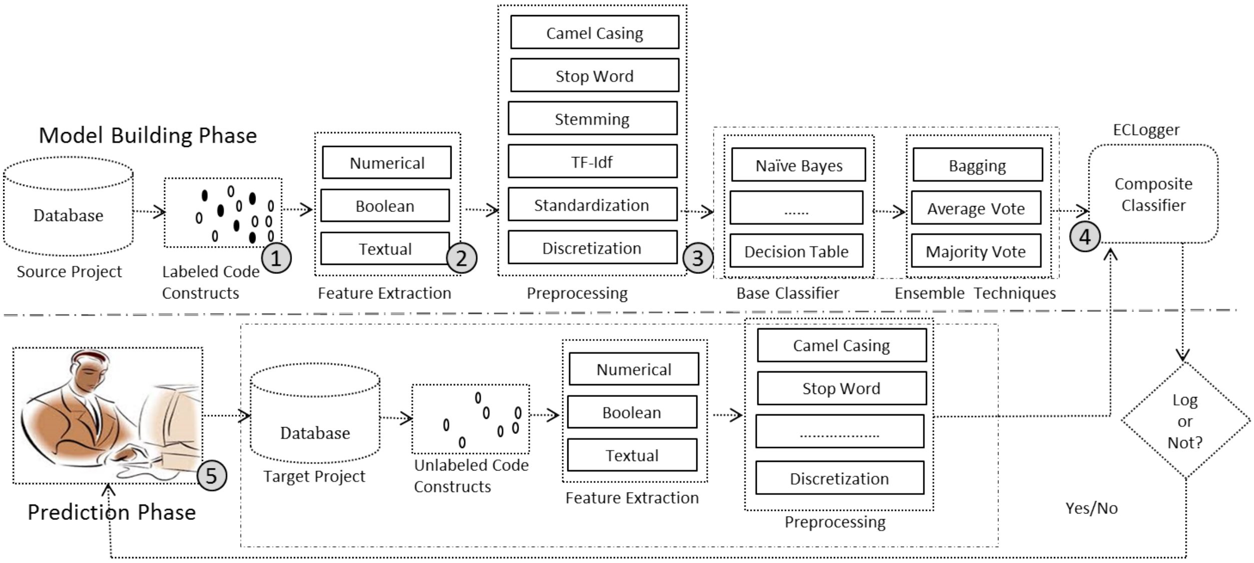


Figure 1. Overview of the proposed ECLogger framework

as logged, it is predicted as logged; otherwise, it is predicted as non-logged.

4. ECLogger model

Figure 1 presents the framework of ECLogger. It consists of two main phases: model building and prediction (refer to Figure 1). In the model building phase, a cross-project logging prediction model is build from the labelled instances of the source project. There are 4 main steps in the model building phase: training instance collection (Step 1), feature extraction (Step 2), pre-processing (Step 3), and ECLogger model building (Step 4) (refer to Step 1 to Step 4 in Figure 1). In the prediction phase, the label (logged or non-logged) of the new instance in the target project is predicted (refer to Step 5 in Figure 1). Algorithm 1 shows the sequence of operations performed by the ECLogger model and the details of the experimental setup (refer to Table 1 for details regarding the notations used). In Algorithm 1, lines 2–6, 11, 15 and 21–22 correspond to the experimental setup, whereas, other lines correspond to the steps of the ECLogger model. The lines 24–26 and 28–32 defines the functions that are part of the experimental setup. The lines 34–39 and 41–49 define the functions that are part of the ECLogger model. The following are the main steps of the ECLogger model:

4.1. Phase 1: (model building)

Training instance collection (step 1): The experimental dataset consists of three projects: Tomcat, CloudStack and Hadoop. One project is considered as the source project ($\mathcal{SP}$), i.e. training project, and the other two projects as the target project ($\mathcal{TP}$), i.e. testing project, a single project at a particular instance. Using this, 6 source and target project pairs are created (lines 7–10 in Algorithm 1). EClogger extracts all logged and non-logged catch-blocks ($\mathcal{CB}_{SP}$) from the source project for training.

Feature extraction (step 2): ECLogger extracts all the features from the source catch-blocks ($\mathcal{CB}_{SP}$) for training as initial source features ($\mathcal{FV}^{I}_{SP}$) (refer to function ExtractFeatures(), i.e. lines 34–39 in Algorithm 1). All 46 features proposed by Lal et al. [10] are used for catch-block logging prediction on Java projects (refer to Table 2). These features are selected because they have shown promising results for within-project catch-block logging prediction [10]. Lal et al. [10] described three properties for the features, i.e. domain, type

Algorithm 1. ECLogger Algorithm

```
procedure ECLogger
    $\mathcal{P} = \{\mathcal{P}_{\mathrm{T}}, \mathcal{P}_{\mathrm{C}}, \mathcal{P}_{\mathrm{H}}\}$
    $\mathcal{A} = \{\mathcal{A}_{\mathrm{ADA}}, \mathcal{A}_{\mathrm{ADT}}, \mathcal{A}_{\mathrm{BN}}, \mathcal{A}_{\mathrm{DT}}, \mathcal{A}_{\mathrm{J48}}, \mathcal{A}_{\mathrm{LR}}, \mathcal{A}_{\mathrm{NB}}, \mathcal{A}_{\mathrm{RF}}, \mathcal{A}_{\mathrm{RBN}}\}$
    $\mathcal{EA} = \{\mathcal{EA}_{\mathrm{BA}}, \mathcal{EA}_{\mathrm{MV}}, \mathcal{EA}_{\mathrm{AV}}\}$
    M=10
    $\mathcal{CS} = \{3, 4, 5, 6, 7, 8, 9\}$
    for all $S \in \mathcal{P}$ do
        for all $T \in \mathcal{P}$ do
            if $S \neq T$ then
                $\mathcal{SP} = S$, $\mathcal{TP} = T$
                $\mathcal{CB}_{\mathcal{SP}}$ = ReadCompleteData($\mathcal{SP}$)
                $\mathcal{FV}^{I}_{\mathcal{SP}}$ = ExtractFeatures($\mathcal{CB}_{\mathcal{SP}}$)
                $\mathcal{FV}^{F}_{\mathcal{SP}}$ = Preprocess($\mathcal{FV}^{I}_{\mathcal{SP}}$)
                ECLogger$_{\mathrm{Model}}$[] = BuildModel($\mathcal{FV}^{F}_{\mathcal{SP}}, \mathcal{A}, \mathcal{EA}, \mathcal{CS}$)
                $\mathcal{CB}_{\mathcal{TP}}[M]$ = ReadBalanceData($\mathcal{TP}$)
                for $i = 1$ to size(ECLogger$_{\mathrm{Model}}$) do
                    for $j = 1$ to $M$ do
                        $\mathcal{FV}^{I}_{\mathcal{TP}}$ = ExtractFeatures($(\mathcal{CB})_{\mathcal{TP}}[j]$)
                        $\mathcal{FV}^{F}_{\mathcal{TP}}$ = Preprocess($\mathcal{FV}^{I}_{\mathcal{TP}}$)
                        $\mathcal{PD}[i][j]$ = ApplyModel($\mathcal{FV}^{F}_{\mathcal{TP}}$, ECLogger$_{\mathrm{Models}}[j]$)
                    $\mathcal{AR}[i] = \frac{\sum_{j=1}^{M} \mathcal{PD}[i][j]}{M}$
                $\mathcal{BM}_{\mathcal{SP} \to \mathcal{TP}}$ = FindBestModel(AR, ECLogger$_{\mathrm{Models}}$)
procedure ReadCompleteData($P$)
    $\mathcal{CB}$ = ReadCatchBlocks($P$)
    return $\mathcal{CB}$
procedure ReadBalanceData($P, M$)
    $\mathcal{CB}$ = ReadCatchBlocks($P$)
    $\widehat{\mathcal{CB}}$ = Randomize($\mathcal{CB}$)
    $\mathcal{BS}[]$ = Generate_M_BalanceSamples($\widehat{\mathcal{CB}}$)
    return $\mathcal{BS}$
procedure ExtractFeatures($\mathcal{CB}$)
    ${}_{T}\mathcal{FV}$ = getTextualFeatures($\mathcal{CB}$)
    ${}_{N}\mathcal{FV}$ = getNumericFeatures($\mathcal{CB}$)
    ${}_{B}\mathcal{FV}$ = getBooleanFeatures($\mathcal{CB}$)
    $\mathcal{FV} = \{{}_{T}\mathcal{FV},\ {}_{N}\mathcal{FV},\ {}_{B}\mathcal{FV}\}$
    return $\mathcal{FV}$
procedure Preprocess($\mathcal{FV}$)
    ${}_{T}\hat{\mathcal{FV}}$ = TF_IDFConversion(Stemming(StopWordRemoval(CamelCaseSeparation(${}_{T}\mathcal{FV}$))))
    if It is Test Data then
        ${}_{T}\widetilde{\mathcal{FV}}$ = FilterFeatureNotTrainData(${}_{T}\widehat{\mathcal{FV}}$)
    else
        ${}_{T}\widetilde{\mathcal{FV}} = {}_{T}\widehat{\mathcal{FV}}$
    $\mathcal{FV}^{F}$ = Discretization(Standardization(Combine(${}_{T}\widetilde{\mathcal{FV}},\ {}_{B}\mathcal{FV},\ {}_{N}\mathcal{FV}$)))
    return $\mathcal{FV}^{F}$
```

Table 1. Notations used in the ECLogger Algorithm (i.e. Algorithm 1)

Notation	Meaning	Notation	Meaning
$\mathcal{P}$	Projects	$\mathcal{A}$	Algorithms
$\mathcal{P}_X$	Project X, where $X \in$ {Tomcat (T), CloudStack (C), Hadoop (H)}	$\mathcal{A}_X$	Algorithm X, where $X \in$ {ADA, ADT, BN, DT, J48, LR, NB, RF, RBN}
$\mathcal{SP}$	Source Project	$\mathcal{EA}$	Ensemble technique
$\mathcal{TP}$	Target Project	$\mathcal{EA}_X$	Ensemble technique X, where $X \in$ {Bagging (BA), Majority Vote (MV), Average Vote (AV)}
$\mathcal{CS}$	Combination Size	$\mathcal{CB}$	Catch-Blocks
$\widehat{\mathcal{CB}}$	Randomized catch-blocks	$\mathcal{CB}_X$	Catch-Blocks of X project, where $X \in \{\mathcal{SP}, \mathcal{TP}\}$
$\mathcal{FV}$	Feature Vector	$\widehat{\mathcal{FV}}$	Feature vector obtained after pre-processing textual features
$\widetilde{\mathcal{FV}}$	Feature vector obtained after filtering undesired features	${}_Z\mathcal{FV}_X^Y$	Feature Vector for project X of type Y and domain Z, where $X \in \{\mathcal{SP}, \mathcal{TP}\}$, $Y \in$ {Initial (I), Final (F)} and $Z \in$ {Textual (T), Numerical (N), Boolean (B)}
$\mathcal{BS}$	Balance SubSamples	$\mathcal{PD}$	Prediction results
$\mathcal{AR}$	Average values of the prediction results	$\mathcal{BM}_{\mathcal{X} \to \mathcal{Y}}$	Best model for $X(\mathcal{SP})$ and $Y(\mathcal{TP})$
i, j, M, P	Temporary Variables	ECLogger$_{\text{Models}}$	All the 940 models generated by ECLogger

and class. *Domain* indicates from which part of the source code a particular feature is extracted. *Type* indicates whether a features is numeric, boolean, or textual. *Class* indicates whether a feature belongs to a positive class or a negative class. In Table 2, the features are categorized based on their *type*. ECLogger extracts all three types of features for model building. For example, the size a try-block (refer to numeric feature 1 in Table 2) is a numeric feature that computes the SLOC of try-blocks associated with logged and non-logged catch-blocks and that belong to the try/catch domain. All the features with their respective properties are listed in Table 2.

Pre-processing (step 3): Six pre-processing steps are applied to clean the initial source features ($\mathcal{FV}_{SP}^{I}$). First the textual features are celaned. All the terms concatenated using the camel-casing in the textual features (i.e. 'getTarget' is converted to 'get', 'target') are separated. Subsequently, all the English stop words from the textual features are removed. The used stop word list was provided by the Python nltk tool [60]. Then stemming is applied (the Porter stemming algorithm by the nltk tool [60]) on all the textual features and converted all the textual features to their tf-idf transformation to create the textual feature vector. The textual feature vector is then combined with numerical and boolean feature vectors. To address the problem of data heterogeneity in the source and target projects, data standardization was performed, i.e. feature values were converted to a z-distribution. Nam et al. [14] demonstrated the usefulness of data normalization for the cross-project defect prediction problem. Finally, all the features were discretized, as some algorithms, such as Naïve Bayes, work only with discretized data. Using this, the final feature vector ($\mathcal{FV}_{SP}^{F}$) for training the model (refer to function Preprocess() is obtained, i.e. lines 41–49 in Algorithm 1).

ECLogger model building (step 4): ECLogger models were built using 9 base classifiers (ADA, ADT, BN, DT, J48, LR, NB, RF, and RBF) and three ensemble techniques (bagging, average voting and maximum voting). Bagging was applied on 8 of the 9 base classifiers. We create 8 ECLogger$_{\text{Bagging}}$ mod-

Table 2. Features used for cross-project catch-block logging prediction taken from previously published work by Lal et al. [10]. PT: class = positive, domain = try/catch; PM: class = positive, domain = method_bt; PO: class = positive, domain = other; NT: class = negative, domain = try/catch; and NM: class = negative, domain = method_bt

Features					
S. No.	Textual (Class Domain)	S. No.	Numeric (Class Domain)	S. No.	Boolean (Class Domain)
1	Catch Exception Type (PT)	1	Size of Try Block [LOC] (PT)	1	Previous Catch Blocks (PT)
2	Log Levels in Try Block (PT)	2	Size of Method_BT[LOC] (PM)	2	Logged Previous Catch Blocks (PT)
3	Log Levels in Method_BT (PM)	3	Log Count Try Block (PT)	3	Logged Try Block (PT)
4	Operators in Try Block (PT)	4	Log Count in Method_BT (PM)	4	Logged Method_BT (PM)
5	Operators in Method_BT (PM)	5	Count of Operators in Try Block (PT)	5	Method have Parameter (PO)
6	Method Parameters (Type) (PO)	6	Count of Operators in Method_BT (PM)	6	IF in Try (PT)
7	Method Parameters (Name) (PO)	7	Variable Declaration Count in Try Block (PT)	7	IF in Method_BT (PM)
8	Container Package Name (PO)	8	Variable Declaration Count in Method_BT (PM)	8	Throw/Throws in Try Block (NT)
9	Container Class Name (PO)	9	Method Call Count in Try Block (PT)	9	Throw/Throws in Catch Block (NT)
10	Container Method Name (PO)	10	Method Call Count in Method_BT (PM)	10	Throw/Throws in Method_-BT (NM)
11	Variable Declaration Name in Try Block (PT)	11	Method Parameter Count (PO)	11	Return in Try Block (NT)
12	Variable Declaration Name in Method_BT (PM)	12	IF Count in Try Block (PT)	12	Return in Catch Block (NT)
13	Method Call Name in Try Block (PT)	13	IF Count in Method_BT (PM)	13	Return in Method_BT (NM)
14	Method Call Name in Method_BT (PM)			14	Assert in Try Block (NT)
				15	Assert in Catch Block (NT)
				16	Assert in Method_BT (NM)
				17	Thread.Sleep in Try Block (NT)
				18	Interrupted Exception Type (NT)
				19	Exception Object "Ignore" in Catch (NT)
Total Features		=	46 (Textual (14) + Numeric (13) + Boolean (19))		

els, i.e. Bagging$_{\text{ADA}}$, Bagging$_{\text{ADT}}$, Bagging$_{\text{BN}}$, Bagging$_{\text{J48}}$, Bagging$_{\text{LR}}$, Bagging$_{\text{NB}}$, Bagging$_{\text{RF}}$ and Bagging$_{\text{RBF}}$. Bagging$_{\text{ADA}}$ is an ECLogger model that is generated by applying bagging on the ADA classifier. Bagging was not applied on the decision table (DT) classifier because of its high time complexity.

The number of created ECLogger average vote models was 466. One can take an average vote of n classifiers to perform a logging prediction on a new code construct. For example, ADA-ADT-BN is one possible combination of 3 classifiers which can be chosen to take an average vote. In this case,the best value of n (i.e. number of classifiers to take) is not known similarly to the information which classifiers are the most suitable for cross-project logging prediction. Hence, all possible combinations of base classifiers are created for $n = \{3, 4, 5, 6, 7, 8, 9\}$. Using this strategy, 466 ECLogger$_{\text{AverageVote}}$ models are created. Similarly to ECLogger$_{\text{AverageVote}}$, 466 ECLogger$_{\text{MajorityVote}}$ models are created. 940 distinct ECLogger models (ECLogger$_{\text{Models}}$[]) are created for cross-project logging prediction (line 14 in Algorithm 1).

4.2. Phase 2: (prediction)

Prediction (step 5): In the prediction phase, ECLogger$_{\text{Models}}$ are used to predict the label of

Table 3. Experimental dataset details

Type	Tomcat	CloudStack	Hadoop
Version	8.0.9	4.3.0	2.7.1
No. of Java Files	2,036	5,350	6,331
SLOC*	273,419	849,857	926,644
Log Statements Count	2,703	10,428	10,108
Total Catch-blocks	3,279	8,077	7,947
Logged Catch-Blocks	887 (27%)	2,792 (34.56%)	2,078 (26.14%)
Distinct Exception Types	119	163	265

* Computed using: http://www.locmetrics.com/.

a new code construct in the target project. All the catch-blocks are extracted from the target project and all the pre-processing techniques described in Step 3 are applied. In addition to these pre-processing steps, one additional filtering step is applied in the prediction phase. In cross-project prediction, there is a possibility that some features that are present in the source project ($\mathcal{FV}_{SP}^{F}$) may not be available in the target project (because of a vocabulary mismatch). Hence, in the target project, the features that are absent in the source project (line 44 in Algorithm 1) are eliminated. Using this, the final feature vector ($\mathcal{FV}_{TP}^{F}$) for the target project instance is created. Then all the ECLogger models to predict the labels of target project instances are applied. For each source and target project pair, the ECLogger$_{\text{Model}}$($\mathcal{BM}_{SP \to TP}$) that provides the best performance (measured in terms of average LF) is then identified.

5. Experimental details

In this section, we present details related to the experiments performed in this work. We present details regarding dataset selection, dataset preparation, experimental environment, design of the experiment, and evaluation metrics.

5.1. Experimental dataset selection

To facilitate the replication of this study, all of our experiments were conducted on open-source Java projects from the Apache Software Foundation (ASF[1]). The ASF consists of a large number of actively maintained and widely used projects. Hence, it is believed that the projects from the ASF consist of good logging and are suitable for our study. We select projects from the ASF that match the following criteria:

1. **Number of files:** The selected projects have have at least 1000 files so that statistically significant conclusions can be drawn.
2. **Number of catch-blocks:** The selected projects have at least 1000 catch-blocks so that statistically significant conclusions can be drawn.
3. **Programing language:** The selected projects are written in the Java programing language. Java projects are selected because Java is one of the most widely used programming languages [61].

Three projects matching the above criteria are selected: Tomcat [62], CloudStack [63], and Hadoop [64] (see Tab. 3). All of these projects are widely used and have previously been used in logging studies [10,13,23,65].

5.2. Experimental dataset preparation

The catch-blocks (see Tab. 3) are extracted from the three projects, i.e. Tomcat, CloudStack and Hadoop. A catch-block is marked as logged (+ve class) if it consists of at least one log statement; otherwise, it is marked as non-logged (–ve class). Numerous variations are observed in the usage format of log statements in the three projects. Hence, 26 regular expressionsare created to ex-

[1] http://www.apache.org/

tract all types of logging statements present in the catch-blocks.

5.3. Experimental environment

The WEKA [48] implementation is used for all the classifiers. The default parameters are used for all the classifiers. All of our experiments are run on Windows Server 2012, with 64 GB RAM, 64-Bit operating system, and an Intel® Xeon® CPU E5-2640 0, 2.50 GHz processor (2 processors), 6 cores per processor.

5.4. Design of the experiment

Two types of experiments were performed: within-project and cross-project catch-block logging prediction. The following presents the experimental design for both types of predictions:
Within-project prediction: To compute the within-project logging prediction, 10 equal-sized balanced datasets for each project were created, namely,Tomcat, CloudStack, and Hadoop. Because the number of –ve class (non-logged) instances is higher than that of +ve class (logged) instances, the subsampling of –ve class instances were performed to make the dataset balanced. In this way, 10 random samples (with replacement) were created from the database. The majority class (–ve class) subsampling technique was used in previous studies to balance the dataset [10,66]. On the balanced dataset, a 70/30 training-testing split is used and the average results over the 10 datasets are reported.
Cross-project prediction: To conduct the cross-project logging prediction experiment, training and testing datasets are created. All the catch-blocks of the source projects are used for training. For the purpose of testing, 10 balanced subsamples of catch-blocks of the target projects are created, i.e. the same as the ones created for the within-project logging prediction. Using this, 10 datasets that have the same training dataset and different testing datasets are created. The results are computed for each of the 10 datasets and report the average results over 10 datasets. Training and testing datasets are created (it is preferred solution to using 10-fold cross validation) to compare the effectiveness of multiple models. This is because in the cross-project prediction the model is trained on the source project and tested on the traget project. Furthermore, separation of training and testing data using 10-fold cross-validation is challenging in this context.

5.5. Evaluation metrics

In this subsection, the performance metrics used to evaluate the effectiveness of the prediction model is described. Five metrics were used in the evaluation process: precision, recall, accuracy, F-measure, and area under the ROC curve. All of these are widely used metrics and were previously used in logging prediction and defect prediction studies [8,10,11,30,67]. There are four possible outcomes while predicting the logging of a code construct:

1. Predicting logged code construct as logged, $l \rightarrow l$ (true positive).
2. Predicting logged code construct as non-logged, $l \rightarrow n$ (false negative).
3. Predicting non-logged code constructs as non-logged, $n \rightarrow n$ (true negative).
4. Predicting non-logged code constructs as logged, $n \rightarrow l$ (false positive).

After constructing the classifier on the training set, its performance on the test set can be evalauted. The total number of logged code constructs predicted as logged ($C_{l\rightarrow l}$), logged code constructs falsely predicted as non-logged ($C_{l\rightarrow n}$), non-logged code constructs predicted as non-logged ($C_{n\rightarrow n}$), and non-logged code constructs predicted and logged ($C_{n\rightarrow l}$) are computed. Using these 4 values, the following metrics are defined:
Logged Precision: It shows the percentage of code constructs that are correctly labelled as logged among those labelled as logged.

$$\text{Logged Precision } (LP) = \frac{C_{l\rightarrow l}}{C_{l\rightarrow l} + C_{n\rightarrow l}} \times 100 \quad (2)$$

Logged Recall: It shows the proportion of logged code constructs that are correctly labelled as logged.

$$\text{Logged Recall } (LR) = \frac{C_{l\to l}}{C_{l\to l} + C_{l\to n}} \times 100 \quad (3)$$

Logged F-measure: It is a metric that combines logged precision and recall. Precision and recall metrics have a trade-off. One can increase precision (recall) by decreasing recall (precision) [39, 68]. Hence, it is difficult to evaluate the performance of different prediction algorithms using only one of the precision or recall metrics. F-measure computes the weighted harmonic mean of precision and recall and is hence useful in overcoming the precision and recall trade-off. It has been widely used in the software engineering literature for performance evaluation [42, 69, 70]. Equation (4) shows the formula to compute the LF metric. In this equation, β is a weighting parameter, where the value of β less than one emphasizes precision and greater than one emphasizes recall. In this paper, $\beta = 1$, which gives equal weightage to both precision and recall, is used.

$$\text{Logged F-measure } (LF) = \frac{(\beta^2 + 1) \times LP \times LR}{\beta^2 \times LP + LR} \times 100 \quad (4)$$

Accuracy: It computes the percentage of code constructs that are correctly labelled as logged or non-logged to the total number of code constructs. It is also a widely used metric for evaluating the performance of prediction models. Accuracy is found to be a biased metric in the case of imbalanced datasets. However, in this work, testing was performed only on balanced datasets.

$$\text{Accuracy } (ACC) = \frac{C_{l\to l} + C_{n\to n}}{C_{l\to l} + C_{l\to n} + C_{n\to n} + C_{n\to l}} \times 100 \quad (5)$$

Area under the ROC curve (*RA*): It measures the likelihood that a logged code construct is given a high likelihood score compared to a non-logged code construct. RA can take any value in the range 0 to 1. In general, higher RA values are considered better, i.e. an RA value of 1 is the best.

6. Experimental results

In this section, the eight identified research questions (RQs) are addressed. The following subsections elaborate the motivation, approach, and results for each of the identified RQs.

6.1. Research questions

Eight RQs are categorized in two dimensions. RQ1–RQ4 investigate the performance of single classifiers for cross-project catch-block logging prediction, whereas RQ5–RQ8 examine the performance of the ECLogger models.

Research Objective 1 (RO1): Performance of the single classifier for cross-project catch-block logging prediction

- **RQ1:** How is the performance of within-project different from cross-project catch-block logging prediction?
- **RQ2:** Which is better, the single-project or multi-project training model for cross-project catch-block logging prediction?
- **RQ3:** Are different classifiers complimentary to each other when applied to cross-project catch-block logging prediction?
- **RQ4:** Are the algorithms that perform best for within-project and cross-project catch-block logging predictions identical?

Research Objective 2 (RO2) : Performance of ensemble-based classifiers, i.e. ECLogger models, for cross-project catch-block logging prediction.

- **RQ5:** What is the performance of ECLogger$_{\text{Bagging}}$ for cross-project catch-block logging prediction?
- **RQ6:** What is the performance of ECLogger$_{\text{AverageVote}}$ for cross-project catch-block logging prediction?
- **RQ7:** What is the performance of ECLogger$_{\text{MajorityVote}}$ for cross-project catch-block logging prediction?
- **RQ8:** What is the average performance of the baseline classifier and ECLogger$_{\text{Models}}$ over all the source and target project pairs?

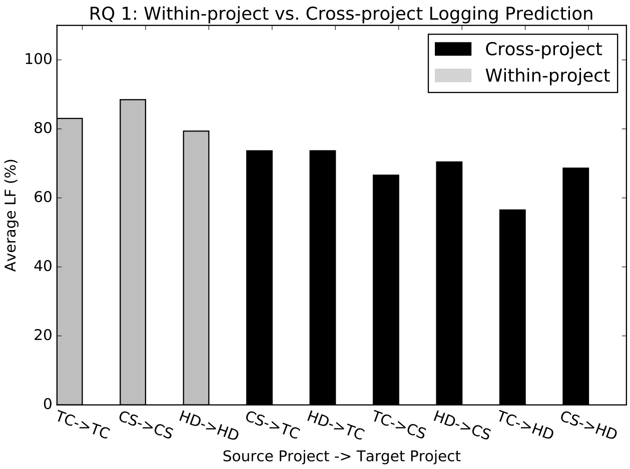


Figure 2. The highest average *LF* of within-project and cross-project logging predictions. CS: CloudStack, TC: Tomcat, and HD: Hadoop

Table 4. Within-project catch-block logging prediction results (using a single classifier). ML ALGO: Machine Learning Algorithm

Project: Tomcat
Total Instances: 1,774, Features: 1,522

ML ALGO	Avg. *LP* (%)	Avg. *LR* (%)	Avg. *LF* (%)	Avg. *ACC* (%)	Avg. *RA* (%)
ADA	75.13 ± 4.76	78.55 ± 11.82	75.97 ± 2.84	76.56 ± 1.13	86.6 ± 0.97
ADT	73.82 ± 3.72	88.59 ± 8.86	80.08 ± 2.05	79.06 ± 1	88.16 ± 0.99
BN	74.79 ± 1.07	81.92 ± 0.75	78.18 ± 0.55	78.08 ± 0.7	87.45 ± 0.76
DT	76.19 ± 2.2	72.12 ± 5.82	73.98 ± 3.16	75.81 ± 2.39	84.12 ± 1.76
J48	80.45 ± 1.7	83.45 ± 2.5	81.92 ± 1.95	82.35 ± 1.81	86.17 ± 2.06
LR	79.98 ± 2.12	86.35 ± 1.2	**83.03** ± 1.36	83.06 ± 1.53	91.64 ± 0.94
NB	74.56 ± 1.12	81.76 ± 0.77	77.99 ± 0.61	77.88 ± 0.76	87.25 ± 0.74
RF	80.93 ± 2.77	82.71 ± 1.96	81.79 ± 2	82.33 ± 2.07	90.37 ± 1.07
RBF	57.98 ± 0.98	93.14 ± 3.63	71.42 ± 0.92	64.3 ± 1.08	75.19 ± 0.87

Project: CloudStack
Total Instances: 5,584, Features: 1,332

ML ALGO	Avg. *LP* (%)	Avg. *LR* (%)	Avg. *LF* (%)	Avg. *ACC* (%)	Avg. *RA* (%)
ADA	72.28 ± 3.94	93.34 ± 7.06	81.13 ± 0.4	78 ± 1.23	85.9 ± 1.31
ADT	79.74 ± 1.99	92.42 ± 3.01	85.54 ± 0.39	84.16 ± 0.43	92.11 ± 0.48
BN	73.6 ± 0.45	94.89 ± 0.45	82.9 ± 0.3	80.14 ± 0.39	89.34 ± 0.4
DT	83.18 ± 1.27	85.34 ± 2.49	84.23 ± 1.63	83.8 ± 1.56	91.54 ± 1.17
J48	88.43 ± 1.25	88.12 ± 2	88.25 ± 0.74	88.1 ± 0.66	91.69 ± 0.58
LR	87.61 ± 0.41	87.28 ± 0.83	87.45 ± 0.52	87.28 ± 0.49	94.16 ± 0.53
NB	73.54 ± 0.49	94.76 ± 0.49	82.81 ± 0.32	80.04 ± 0.43	89.2 ± 0.39
RF	86.21 ± 0.96	90.86 ± 0.99	**88.47** ± 0.85	87.98 ± 0.88	94.93 ± 0.28
RBF	55.02 ± 1.52	100 ± 0	70.97 ± 1.28	58.44 ± 2.64	57.79 ± 2.68

Project: Hadoop, Type: Catch-Block
Total Instances: 4,156, Features: 1,322

ML ALGO	Avg. *LP* (%)	Avg. *LR* (%)	Avg. *LF* (%)	Avg. *ACC* (%)	Avg. *RA* (%)
ADA	73.74 ± 0.89	74.74 ± 2	74.22 ± 1.13	74.53 ± 0.92	81.06 ± 0.42
ADT	75.28 ± 1.93	78.64 ± 1.88	76.89 ± 0.81	76.79 ± 1	83.17 ± 0.25
BN	74.11 ± 1.12	65.18 ± 0.89	69.35 ± 0.72	71.72 ± 0.72	81.06 ± 0.63
DT	76.76 ± 1.66	76.13 ± 2.56	76.4 ± 1.12	76.93 ± 0.96	83.27 ± 0.7
J48	77.89 ± 1.89	79.74 ± 1.59	78.78 ± 0.93	78.91 ± 1.1	81.57 ± 1.21
LR	78.74 ± 1.08	80 ± 0.94	**79.36** ± 0.62	79.57 ± 0.68	87.25 ± 0.5
NB	74.08 ± 1.09	65.13 ± 0.84	69.31 ± 0.69	71.69 ± 0.7	80.97 ± 0.63
RF	77.9 ± 0.91	77.75 ± 1.27	77.82 ± 0.94	78.25 ± 0.88	86.28 ± 0.65
RBF	57.07 ± 0.81	76.68 ± 4.87	65.39 ± 2.27	60.27 ± 1.33	59.32 ± 1.64

6.2. RO1: Performance of the single classifier for cross-project catch-block logging prediction

In this subsection four RQs (RQ1–RQ4), which investigate the performance of single classifiers, are answered. The questions related to the variation in performance of a single classifier for within-project and cross-project logging predictions using multiple evaluation metrics, using both single-project and multi-project training models are answered.

6.2.1. RQ1: How is the performance of within-project different from cross-project catch-block logging prediction

Motivation: IIn RQ1, the effectiveness of within-project and cross-project logging prediction models (using a single classifier) are compared. Cross-project logging prediction is challenging, and hence, it is important to identify the performance variation of the cross-project logging prediction model compared to that of the within-project logging prediction model. The results from this investigation can provide important insights and motivation for constructing the cross-project logging prediction model.

Approach: To answer RQ1, the performances of single classifiers for within-project and cross-project logging prediction are compared. The average *LF* is used to compare the performances of different classifiers.

Results: Table 4 presents the detailed results of within-project catch-block logging prediction for all three projects. Our experimental results show that the RF and LR models outperform other algorithms in terms of average *LF*. The highest average *LF* of 83.03%, 88.47%, and 79.36% for the within-project catch-block logging prediction was achieved on the Tomcat, CloudStack and Hadoop projects, respectively. Figure 2 shows the highest average *LF* values from the within-project and cross-project experiments. Figure 2 shows that the highest average *LF* for all six cross-project results is lower than all three within-project results. Table 5, Table 6 and Table 7 show the detailed cross-project logging prediction results (using a single classifier). These experimental results show that for the cross-project logging prediction, the highest average *LF* of 73.66%, 70.42% and 68.62% was achieved for the Tomcat, CloudStack and Hadoop projects, respectively. A 6.37% to 18.05% decrease was observed in the classification performance for cross-project logging prediction compared to within-project logging prediction. The performance of the RBF classifier is the worst for cross-project logging prediction. For all six pairs of source and target project pairs, RBF provides an average *LF* of 0% and average *ACC* of 50%, i.e. predicting all the code constructs as non-logged (refer to Table 5, Table 6, and Table 7).

> A 6.37% to 18.05% decrease was observed in the average *LF* for cross-project logging prediction compared to within-project logging prediction.

6.2.2. RQ2: Which is better, the single-project or multi-project training model for cross-project catch-block logging prediction?

Motivation: In RQ2, the objective is to examine the effectiveness of multi-project training for cross-project logging prediction. Thus it is necessary to ascertain whether *information fusion* enhances the accuracy of the cross-project logging prediction. Training a predictive model from multiple projects is one type of information fusion-based approach and was shown to enhance accuracy because it involves combining information from multiple sources. Few studies in the past used multi-project training for cross-project defect prediction [30, 71]. However, for cross-project logging prediction, this has yet to be explored. The answer to this RQ can provide important insights about selecting the single-project or multi-project cross-project logging prediction model.

Approach: Approach: To answer RQ2, 9 pairs of source and target projects are created, i.e. 6 pairs consisting of one source project and 3 pairs consisting of two source projects.

Results: Figure 3 presents the histogram of the average *LF* and average *ACC* val-

Table 5. Cross-project catch-block logging prediction results (using a single classifier) for Tomcat (target project). ML ALGO: Machine Learning Algorithm

Project: CloudStack→Tomcat
Total Instances (Source): 8077, Total Instances (Target): 1,774, Features: 1,304

ML ALGO	Avg. *LP* (%)	Avg. *LR* (%)	Avg. *LF* (%)	Avg. *ACC* (%)	Avg. *RA* (%)
ADA	50.07 ± 0.1	97.07 ± 0	66.06 ± 0.08	50.13 ± 0.19	57.11 ± 0.18
ADT	78.26 ± 0.79	69.11 ± 0	73.4 ± 0.35	74.95 ± 0.44	82.96 ± 0.45
BN	66.36 ± 0.57	82.75 ± 0	**73.65** ± 0.35	70.39 ± 0.54	77.16 ± 0.51
DT	60.85 ± 0.55	82.19 ± 0	69.93 ± 0.36	64.65 ± 0.61	77.56 ± 0.43
J48	58.64 ± 0.41	70.35 ± 0	63.96 ± 0.24	60.37 ± 0.42	61.4 ± 0.72
LR	64.2 ± 0.78	66.74 ± 0	65.45 ± 0.4	64.76 ± 0.62	69.84 ± 0.51
NB	66.39 ± 0.56	82.41 ± 0	73.54 ± 0.34	70.34 ± 0.52	77.19 ± 0.51
RF	62.08 ± 0.63	56.82 ± 0	59.33 ± 0.29	61.05 ± 0.46	63.64 ± 0.49
RBF	0 ± 0	0 ± 0	0 ± 0	50 ± 0	50 ± 0

Project: Hadoop→Tomcat
Instances (Source): 7,947, Instances (Target): 1,774, Features: 1,313

ML ALGO	Avg. *LP* (%)	Avg. *LR* (%)	Avg. *LF* (%)	Avg. *ACC* (%)	Avg. *RA* (%)
ADA	85.5 ± 0.88	49.15 ± 0	62.42 ± 0.24	70.41 ± 0.3	79.45 ± 0.32
ADT	79.37 ± 0.95	55.36 ± 0	65.22 ± 0.32	70.48 ± 0.42	77.96 ± 0.44
BN	74.74 ± 0.67	72.49 ± 0	73.6 ± 0.33	73.99 ± 0.44	80.17 ± 0.47
DT	84.98 ± 0.94	52.2 ± 0	64.67 ± 0.27	71.48 ± 0.34	77.17 ± 0.33
J48	65.82 ± 0.65	65.95 ± 0	65.88 ± 0.33	65.85 ± 0.5	66.8 ± 0.66
LR	76.52 ± 0.72	54.57 ± 0	63.7 ± 0.25	68.91 ± 0.34	76.99 ± 0.43
NB	74.76 ± 0.64	72.6 ± 0	**73.66** ± 0.31	74.04 ± 0.41	80.19 ± 0.47
RF	67.91 ± 1.06	39.46 ± 0	49.91 ± 0.29	60.4 ± 0.46	67.2 ± 0.52
RBF	0 ± 0	0 ± 0	0 ± 0	50 ± 0	52.77 ± 0.93

Table 6. Cross-project catch-block logging prediction results (using a single classifier) for CloudStack (target project). ML ALGO: Machine Learning Algorithm

Project: Tomcat→CloudStack
Total Instances (Source): 3,279, Total Instances (Target): 5,584, Features:1,425

ML ALGO	Avg. *LP* (%)	Avg. *LR* (%)	Avg. *LF* (%)	Avg. *ACC* (%)	Avg. *RA* (%)
ADA	87.84 ± 0.49	53.19 ± 0	66.26 ± 0.14	72.91 ± 0.17	81.45 ± 0.22
ADT	90.12 ± 0.41	52.79 ± 0	**66.58** ± 0.11	73.5 ± 0.13	80.95 ± 0.13
BN	63.46 ± 0.39	69.41 ± 0	66.3 ± 0.21	64.72 ± 0.34	71.7 ± 0.38
DT	72.75 ± 0.33	45.38 ± 0	55.89 ± 0.1	64.19 ± 0.14	74.41 ± 0.16
J48	66.36 ± 0.44	56.91 ± 0	61.28 ± 0.19	64.03 ± 0.28	63.32 ± 0.34
LR	80.48 ± 0.56	48.14 ± 0	60.24 ± 0.16	68.23 ± 0.21	74.94 ± 0.14
NB	63.36 ± 0.39	69.23 ± 0	66.16 ± 0.21	64.59 ± 0.34	71.7 ± 0.38
RF	80.84 ± 0.38	37.29 ± 0	51.03 ± 0.08	64.22 ± 0.11	75.45 ± 0.18
RBF	0 ± 0	0 ± 0	0 ± 0	50 ± 0	63.11 ± 0.44

Project: Hadoop→CloudStack
Instances (Source): 7,947, Instances (Target): 5,584, Features: 1,313

ML ALGO	Avg. *LP* (%)	Avg. *LR* (%)	Avg. *LF* (%)	Avg. *ACC* (%)	Avg. *RA* (%)
ADA	83.44 ± 0.59	49.61 ± 0	62.22 ± 0.16	69.88 ± 0.21	79.79 ± 0.25
ADT	88.64 ± 0.36	51.33 ± 0	65.01 ± 0.1	72.37 ± 0.12	81.86 ± 0.16
BN	64.25 ± 0.29	77.87 ± 0	70.41 ± 0.17	67.27 ± 0.27	76.79 ± 0.29
DT	84.71 ± 0.5	45.95 ± 0	59.58 ± 0.12	68.83 ± 0.16	74.65 ± 0.2
J48	64.58 ± 0.38	58.45 ± 0	61.37 ± 0.17	63.2 ± 0.27	65.21 ± 0.28
LR	83.19 ± 0.5	55.73 ± 0	66.75 ± 0.16	72.23 ± 0.2	79.03 ± 0.2
NB	64.25 ± 0.29	77.9 ± 0	**70.42** ± 0.17	67.27 ± 0.27	76.8 ± 0.29
RF	83.91 ± 0.57	36.1 ± 0	50.48 ± 0.1	64.59 ± 0.15	73.11 ± 0.25
RBF	0 ± 0	0 ± 0	0 ± 0	50 ± 0	57.72 ± 0.42

Table 7. Cross-project logging prediction results (using a single classifier) for Hadoop (target project). ML ALGO: Machine Learning Algorithm

Project: Tomcat→Hadoop
Total Instances (Source): 3,279, total instances (target): 4,156, features: 1,425

ML ALGO	Avg. *LP* (%)	Avg. *LR* (%)	Avg. *LF* (%)	Avg. *ACC* (%)	Avg. *RA* (%)
ADA	85.91 ± 0.8	37.63 ± 0	52.34 ± 0.15	65.73 ± 0.2	78.22 ± 0.32
ADT	87.76 ± 0.7	33.88 ± 0	48.89 ± 0.11	64.58 ± 0.15	77.04 ± 0.18
BN	73.67 ± 0.72	45.57 ± 0	56.31 ± 0.21	64.64 ± 0.3	69.96 ± 0.39
DT	83.69 ± 0.74	34.31 ± 0	48.67 ± 0.12	63.81 ± 0.18	72.14 ± 0.45
J48	67.57 ± 0.61	36.77 ± 0	47.62 ± 0.15	59.56 ± 0.24	70.75 ± 0.36
LR	82.12 ± 1.01	26.23 ± 0	39.76 ± 0.12	60.26 ± 0.2	73.99 ± 0.35
NB	73.58 ± 0.68	45.86 ± 0	**56.5** ± 0.2	64.69 ± 0.29	69.47 ± 0.38
RF	82.76 ± 0.99	21.13 ± 0	33.66 ± 0.08	58.36 ± 0.15	69.35 ± 0.4
RBF	0 ± 0	0 ± 0	0 ± 0	50 ± 0	55.4 ± 0.34

Project: CloudStack→Hadoop
Instances (source): 8,077, instances (target): 4,156, features: 1,304

ML ALGO	Avg. *LP* (%)	Avg. *LR* (%)	Avg. *LF* (%)	Avg. *ACC* (%)	Avg. *RA* (%)
ADA	50.29 ± 0.06	98.12 ± 0	66.5 ± 0.05	50.57 ± 0.12	54.24 ± 0.1
ADT	79.55 ± 0.56	52.12 ± 0	62.97 ± 0.18	69.36 ± 0.23	76.51 ± 0.27
BN	57.26 ± 0.3	79.74 ± 0	66.65 ± 0.2	60.11 ± 0.36	68.2 ± 0.39
DT	77.99 ± 0.36	61.26 ± 0	**68.62** ± 0.14	71.99 ± 0.18	76.31 ± 0.14
J48	73.4 ± 0.65	58.13 ± 0	64.88 ± 0.25	68.53 ± 0.35	69.99 ± 0.45
LR	72.76 ± 0.95	55.58 ± 0	63.02 ± 0.36	67.38 ± 0.5	71.83 ± 0.61
NB	57.31 ± 0.28	79.69 ± 0	66.67 ± 0.19	60.16 ± 0.34	68.16 ± 0.39
RF	66.35 ± 0.77	46.92 ± 0	54.97 ± 0.26	61.56 ± 0.41	67.32 ± 0.41
RBF	0 ± 0	0 ± 0	0 ± 0	50 ± 0	50 ± 0

ues of multi-project cross-project catch-block logging prediction. Figure 3a reveals that there is no dominant approach between single-project and multi-project. In certain instances, multi-project training increased the prediction performance, and in other cases it has decreased the prediction performance. For example, in the CloudStack project when single source-project training is used, the highest average *LF* of 66.5% (source project Tomcat) and 70.42% (source project Hadoop) are achived (refer to Table 6). In contrast, when multi-project training is used and both Tomcat and Hadoop are applied to build the model, the highest average *LF* of 67.74% is achieved. Hence, multi-project training causes a 1.24% decrease and a 2.68% increase in the prediction performance of single-project training when Tomcat and Hadoop are used, respectively. A similar result is observed for the *ACC* metric (refer to Figure 3b).

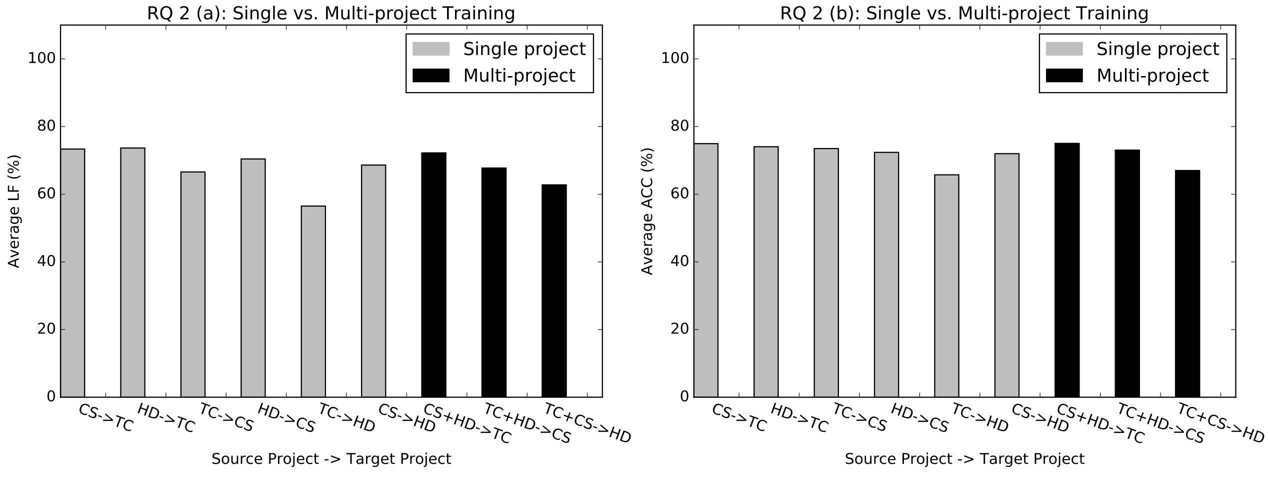
> There is no dominant approach among the single-project and multi-project cross-project catch-block prediction models.

6.2.3. RQ3: Are different classifiers complimentary to each other when applied to cross-project catch-block logging prediction?

Motivation: In RQ3, the objective is to examine the performance of individual classifiers on multiple evaluation metrics. The evaluation of a predictive model or a classifier can be performed using several metrics or measures, and the selected set of metrics depends on the classification task and problem. The Authors believe that the answer to this research question will provide important insights about combining different classifiers (using an ensemble of classifiers) for improving the cross-project logging prediction performance.

Approach: Individual classifiers are compared on 5 evaluation metrics, namely, average *LP*, average *LR*, average *LF*, average *ACC*, and average *RA*, to identify whether a single classifier dominates and provides the highest values over all the evaluation metrics.

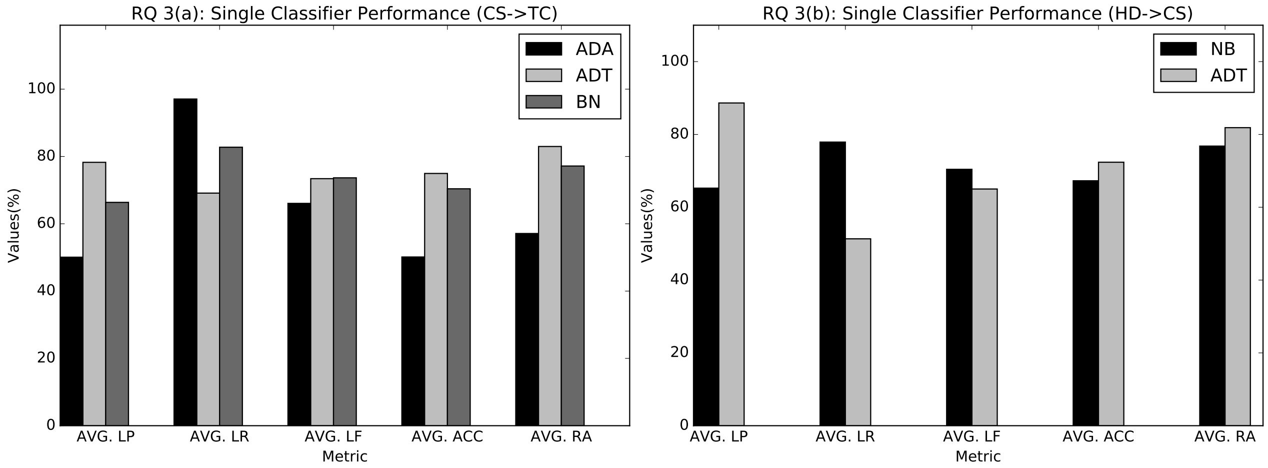
Results: The results indicate that different classifiers are complementary to each other. For



(a) Average *LF* (%)

(b) Average *ACC* (%)

Figure 3. The Highest Average *LF* of Single-project and Multi-project Training Logging Prediction Models. CS: CloudStack, TC: Tomcat, and HD: Hadoop



(a) Classifier performance for the CS→TC project pair(b) Classifier performance for the HD→CS project pair

Figure 4. The Results (average *LP*, *LR*, *LF*, *ACC* and *RA*) of Selected Single Classifiers. CS: CloudStack, TC: Tomcat, and HD: Hadoop

example, consider the results obtained on the following source and target project pair:

CloudStack (source)→Tomcat (target): Figure 4a presents the histogram of all five metrics (*LP*, *LR*, *LF*, *ACC* and *RA*) for the CloudStack→Tomcat project pair for the ADA, ADT and NB classifiers. ADA, ADT and NB are selected because these three classifiers provide the best results for cross-project catch-block logging prediction on the CloudStack→Tomcat project pair. Figure 4a shows that the ADT model provides the highest average *LP*, *ACC* and *RA* values, whereas ADA and BN provide the highest average *LR* and *LF*, respectively (refer to Table 5 for detailed results).

Hadoop (source)→CloudStack (target): Similarly to Figure 4a, Figure 4b presents the histogram of all 5 metrics for the Hadoop→CloudStack project pair for the ADT and NB classifiers. Figure 4b shows that NB provides the highest average *LR* and *LF* values, whereas ADT provides the highest average *LP*, *ACC*, and *RA* values (refer to Table 6 for detailed results).

The above two examples indicate that different classifiers provide complementary information for cross-project catch-block logging prediction and, hence, their ensemble can be beneficial for improving the results of the prediction model [72].

> The results indicate that the different classifiers are complementary to each other for cross-project catch-block logging prediction.

6.2.4. RQ4: Are the algorithms that perform best for within-project and cross-project catch-block logging predictions identical?

Motivation: In a related work, Zhu et al. [11] used the same algorithm (J48) for both within-project and cross-project logging predictions. However, there is a possibility that the same algorithm is not suitable for both within-project and cross-project logging predictions. In RQ4, the performances of different classifiers for within-project and cross-project logging predictions are compared. The Authors believe that the results of this investigation will provide us with important insights regarding algorithm selection for ensemble creation.

Approach: To answer RQ4, we compare the performances of different classifiers for within-project and cross-project logging predictions.

Results: Figure 5 presents the histogram of the average *LF* values of the RF, ADT and NB classifiers for within-project and cross-project logging predictions for the CloudStack project. Figure 5 shows that the RF classifier provides the highest average *LF* of 88.47% for within-project logging prediction. The ADT and NB models provide considerably lower average *LF* of 85.54% and 82.81%, respectively, compared to the RF classifier for within-project logging prediction. However, for cross-project logging prediction, the ADT and NB classifiers provide better average *LF* compared to that of the RF classifier. For example, for the Hadoop→CloudStack project pair, NB provides an average *LF* of 70.42%, which is considerably higher than the average *LF* of the RF classifier (50.48%). Similar results are observed for other classifiers on other source and target project pairs (refer to Table 4, Table 5, Table 6 and Table 7 for detailed results). This result shows that algorithms that perform best for within-project and cross-project catch-block logging predictions are different. These results reveal the weakness of the cross-project logging prediction experiment performed by Zhu et al. [11], where the authors perform within-project and cross-project logging predictions using the same algorithm (J48). Hence, in this work, the Authors explore multiple classifiers for cross-project catch-block logging prediction model building.

> Classifiers that provide the best results for within-project and cross-project logging predictions are different.

Performance summary of base classifiers (RQ1–RQ4): In RO1, 4 investigations are performed in the context of cross-project logging prediction. RQ1 indicates that the results of single classifiers are considerably lower for cross-project logging prediction compared to the results for within-project logging prediction. Hence, more advanced methods are required for building the cross-project logging prediction model. RQ2 indicates that multi-project training does not improve the performance of cross-project logging prediction on all source and target project pairs. Hence, to improve the model building time, only a single project for training the cross-project logging prediction model is considered. RQ3 indicates that the classifiers provide complementary information for the task of cross-project logging prediction. Hence, the Authors believe that an ensemble of different classifiers may be beneficial in improving the performance of cross-project logging prediction. RQ4 indicates that the classifiers which provide good results for within-project logging prediction are different from the classifiers which provide good results for cross-project logging prediction. Hence, to build an ensemble of classifiers to improve the performance of cross-project logging prediction, it is necessary to conduct experiments on a wide range of classifiers to find the best set of classifiers. The first four RQs derive the motivation for constructing the ECLogger model, i.e. an ensemble of classifiers-based model which uses a single project for training the model.

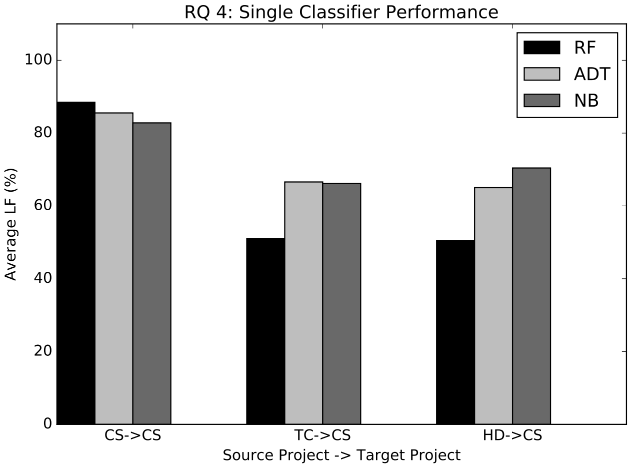


Figure 5. Performance of RF, ADT, and NB classifiers for within-project and cross-project catch-block logging predictions

6.3. RO2: Performance of ensemble-based classifiers for cross-project catch-block logging prediction

In this subsection, the performances of ensemble-based classifiers are investigated and compared with the performances of single classifiers for cross-project logging prediction (refer to RQ5–RQ8). For each source and target project pair, the single classifier that provides the best results (in terms of average *LF*) becomes the baseline classifier. For example, for the CloudStack→Tomcat project pair, the BN classifier provides the highest average *LF* and is hence considered to be a baseline classifier (refer to Table 5).

6.3.1. RQ5: What is the performance of ECLogger$_{\text{Bagging}}$ for cross-project catch-block logging prediction?

Motivation: In RQ5, the performances of 8 ensemble classifiers, created using the bagging technique, are investigated and compared with the performance of the baseline classifier. The answer to this research question can provide important insights regarding whether bagging is useful in improving the performance of cross-project catch-block logging prediction.

Approach: To answer this research question, the average *LF* and average *ACC* of 8 ensemble classifiers generated by applying bagging on the base classifiers, i.e. ECLogger$_{\text{Bagging}}$ models, are computed. For each source→target project pair, the bagging model which provides the best average *LF* is reported. Then the results obtained by the best bagging model is compared with the baseline classifier for each source→target project pair.

Results: Table 8 presents the average *LF* and average *ACC* of the baseline classifier and the best ECLogger$_{\text{Bagging}}$ model for each source→target project pair. Table 8 shows that ECLogger$_{\text{Bagging}}$ considerably improves (more than 1% improvement) the average *LF* and average *ACC* for three and two source→target project pairs, respectively. It improves the average *LF* and average *ACC* by 4.6% and 5.57% (CloudStack→Tomcat) and by 3.96% and 2.44% (CloudStack→Hadoop) when Bagging$_{ADT}$ is used. For the Tomcat→CloudStack project pair, project pair, a considerable improvement (1.03%) is observed only in the average *LF*. For all other source and target project pairs, no considerable difference in the performance of ECLogger$_{\text{Bagging}}$ was observed, compared to the performance of the baseline classifier. Overall, the Bagging$_{ADT}$ model performs better than the other bagging models and gives the highest average *LF* for three source and target project pairs.

ECLogger$_{\text{Bagging}}$ shows a considerable improvement in the average *LF* in 3 out of 6

Table 8. Results of ECLogger$_{\text{Bagging}}$. NA: Not Applicable and IMP: Improvement

Source Project →Target Project	Algorithm	Avg. *LF* (%)	%IMP	Avg. *ACC* (%)	%IMP
CloudStack→Tomcat	Baseline (BN)	73.65 ± 0.35	NA	70.39 ± 0.54	NA
	Bagging$_{\text{ADT}}$	78.25 ± 0.22	**4.6**	75.96 ± 0.32	**5.57**
Hadoop→Tomcat	Baseline (NB)	73.66 ± 0.31	NA	74.04 ± 0.41	NA
	Bagging$_{\text{NB}}$	73.54 ± 0.29	-0.12	73.96 ± 0.39	-0.08
Tomcat→CloudStack	Baseline (ADT)	66.58 ± 0.11	NA	73.5 ± 0.13	NA
	Bagging$_{\text{ADT}}$	67.61 ± 0.14	**1.03**	73.92 ± 0.17	0.42
Hadoop→CloudStack	Baseline (NB)	70.42 ± 0.17	NA	67.27 ± 0.27	NA
	Bagging$_{\text{BN}}$	70.49 ± 0.19	0.07	67.56 ± 0.29	0.29
Tomcat→Hadoop	Baseline (NB)	56.5 ± 0.2	NA	64.69 ± 0.29	NA
	Bagging$_{\text{NB}}$	56.43 ± 0.2	-0.07	64.7 ± 0.29	0.01
CloudStack→Hadoop	Baseline (DT)	68.62 ± 0.14	NA	71.99 ± 0.18	NA
	Bagging$_{\text{ADT}}$	72.58 ± 0.3	**3.96**	74.43 ± 0.39	**2.44**

source→target project pairs with a maximum improvement of 4.6% (average *LF*) and 5.57% (average *ACC*) for the CloudStack→Tomcat project pair.

6.3.2. RQ6: What is the performance of ECLogger$_{\text{AverageVote}}$ for cross-project catch-block logging prediction?

Motivation: In RQ6, the performances of 466 ECLoggerAverageV ote models are investigated and compared with the performances of baseline classifiers. The answer to this research question can provide important insights about the effectiveness of the average vote ensemble technique for cross-project catch-block logging prediction.
Approach: To answer this research question, the average *LF* and average *ACC* of 466 ECLogger$_{\text{AverageVote}}$ models, generated using the average voting ensemble technique are computed. For each source→target project project pair, the ECLogger$_{\text{AverageVote}}$ model which provides the best average *LF* is reported. We then compare the results obtained by the best ECLogger$_{\text{AverageVote}}$ models with the baseline classifier for each source→target project pair.
Results: Table 9 presents the average *LF* and average *ACC* of the baseline classifier and the best average voting technique for each source→target project pair. Table 9 shows that the ECLogger$_{\text{AverageVote}}$ technique considerably improves the average *LF* and average *ACC* on 5 source→target project pairs. ECLogger$_{\text{AverageVote}}$ improves the average *LF* and average *ACC* by 3.78% and 5.92% (CloudStack→Tomcat), 2.3% and 3.01% (Hadoop→Tomcat), 7.04% and 2.45% (Tomcat→CloudStack), 6.9% and 11.43% (Hadoop→CloudStack) and 3.57% and 1.35% (CloudStack→Hadoop). For the 6th source→ →target project pair, i.e. Tomcat→Hadoop, ECLogger$_{\text{AverageVote}}$ shows a considerable improvement in the average *ACC* (1.74%), whereas for the average *LF*, it provides results comparable to the baseline classifier. No particular group of classifiers whose average vote provides the best results on each source and target project pair was observed. However, it was observed that the ADT, DT, BN, and NB classifiers are present in most of the ECLogger$_{\text{AverageVote}}$ models which provide the best results.

ECLogger$_{\text{AverageVote}}$ shows a considerable improvement in the average *LF* in 5 out of 6 source→target project pairs with a maximum improvement of 7.04% in the average *LF* (for the Tomcat→CloudStack project pair) and 11.43% in the average *ACC* (for the Hadoop→CloudStack project pair).

6.3.3. RQ7: What is the performance of ECLogger$_{\text{MajorityVote}}$ for cross-project catch-block logging prediction?

Motivation: In RQ7, the performances of 466 ECLogger$_{\text{MajorityVote}}$ models are investigated and compared with the performances of baseline classifiers. The answer to this research question can

Table 9. Results of ECLogger$_{\text{AverageVote}}$. IMP: Improvement, NA: Not Applicable, and AV: Average Vote

Source Project →Target Project	Algorithm	Avg. *LF* (%)	%IMP	Avg *ACC* (%)	%IMP
CloudStack→Tomcat	Baseline (BN)	73.65 ± 0.35	NA	70.39 ± 0.54	NA
	AV (ADT-BN-DT-LR)	77.43 ± 0.47	**3.78**	76.31 ± 0.64	**5.92**
Hadoop→Tomcat	Baseline (NB)	73.66 ± 0.31	NA	74.04 ± 0.41	NA
	AV (ADT-BN-DT-J48-NB)	75.96 ± 0.41	**2.3**	77.05 ± 0.51	**3.01**
Tomcat→CloudStack	Baseline (ADT)	66.58 ± 0.11	NA	73.5 ± 0.13	NA
	AV (ADT-BN-DT-J48-NB)	73.62 ± 0.1	**7.04**	75.95 ± 0.13	**2.45**
Hadoop→CloudStack	Baseline (NB)	70.42 ± 0.17	NA	67.27 ± 0.27	NA
	AV (ADA-ADT-BN-DT-LR-NB)	77.32 ± 0.15	**6.9**	78.7 ± 0.18	**11.43**
Tomcat→Hadoop	Baseline (NB)	56.5 ± 0.2	NA	64.69 ± 0.29	NA
	AV (ADT-BN-DT-J48-NB)	56.77 ± 0.16	**0.27**	66.43 ± 0.22	**1.74**
CloudStack→Hadoop	Baseline (DT)	68.62 ± 0.14	NA	71.99 ± 0.18	NA
	AV (ADA-ADT-DT-J48-NB)	72.19 ± 0.26	**3.57**	73.34 ± 0.34	**1.35**

Table 10. Results of ECLogger$_{\text{MajorityVote}}$. IMP: Improvement, NA: Not Applicable, and MV: Majority Vote

Source Project→Target Project	Algorithm	Avg. *LF* (%)	%IMP	Avg. *ACC* (%)	**%IMP**
CloudStack→Tomcat	Baseline (BN)	73.65 ± 0.35	NA	70.39 ± 0.54	NA
	MV (ADT-BN-DT)	77.85 ± 0.3	**4.2**	76.78 ± 0.41	**6.39**
Hadoop→Tomcat	Baseline (NB)	73.66 ± 0.31	NA	74.04 ± 0.41	NA
	MV (BN-NB-RF)	73.7 ± 0.32	0.04	74.09 ± 0.43	0.05
Tomcat→CloudStack	Baseline (ADT)	66.58 ± 0.11	NA	73.5 ± 0.13	NA
	MV (ADA-BN-J48- LR-NB)	71.12 ± 0.13	**4.54**	74.45 ± 0.16	**0.95**
Hadoop→CloudStack	Baseline (NB)	70.42 ± 0.17	NA	67.27 ± 0.27	NA
	MV (ADA-BN-DT- LR-NB)	74.17 ± 0.18	**3.75**	76.74 ± 0.22	**9.47**
Tomcat→Hadoop	Baseline (NB)	56.5 ± 0.2	NA	64.69 ± 0.29	NA
	MV (ADT-BN-NB)	56.49 ± 0.2	-0.01	64.75 ± 0.28	0.06
CloudStack→Hadoop	Baseline (DT)	68.62 ± 0.14	NA	71.99 ± 0.18	NA
	MV (ADA-ADT- BN-DT-J48)	74.01 ± 0.23	**5.39**	73.78 ± 0.31	**1.79**

provide important insights about the effectiveness of majority vote models for cross-project catch-block logging prediction.

Approach: To answer this research question, the average *LF* and average *ACC* of 466 ECLogger$_{\text{MajorityVote}}$ models generated using the majority vote ensemble technique are computed. For each source→target project pair, the ECLogger$_{\text{MajorityVote}}$ model which provides the best average *LF* is reported. Then the results obtained by the best ECLogger$_{\text{MajorityVote}}$ models are compared with the baseline classifier for each source→target project pair.

Results: Table 10 shows the average *LF* and average *ACC* of the baseline classifier and the best majority vote classifier for each source→target project pair. Table 10 shows that the ECLogger$_{\text{MajorityVote}}$ classifier improves the average *LF* and average *ACC* for 4 source→target project pairs. ECLogger$_{\text{MajorityVote}}$ improves the average *LF* and average *ACC* by 4.2% and 6.39% (CloudStack→Tomcat), 4.54% and 0.95% (Tomcat→CloudStack), 3.75% and 9.47% (Hadoop→CloudStack) and 5.39% and 1.79% (CloudStack→Hadoop). In other cases, no considerable difference in the performances of the baseline classifier and ECLogger$_{\text{MajorityVote}}$ classifier was observed. No particular group of classifiers whose majority vote provides the best results on each source and target project pair was observed. However, it was observed that the BN classifier was present in all of the ECLogger$_{\text{MajorityVote}}$ models that provide the best results.

ECLogger$_{\text{MajorityVote}}$ improved the cross-project catch-block logging prediction in 4 out of 6 source→target project pairs with

Table 11. Average performance of ECLogger models on all source→target projects pairs. IMP: Improvement, NA: Not Applicable, AV: Average Vote, and MV: Majority Vote

Source→Target	Approach	Algorithm	Avg. LF (%)	% IMP	Avg. ACC (%)	%IMP
All	Baseline	NB	67.825	NA	66.85	NA
	ECLogger$_{\text{Bagging}}$	Bagging$_{\text{BN}}$	67.8	−0.025	66.89	0.04
	ECLogger$_{\text{AverageVote}}$	AV (ADA-ADT-BN-DT-LR-NB)	70.95	**3.12**	72.93	**6.08**
	ECLogger$_{\text{MajorityVote}}$	MV (ADT-BN-DT-LR-NB)	68.775	0.95	72.84	**5.99**

> a maximum improvement of 5.39% in the average LF (for the CloudStack→Hadoop project pair) and 9.47% in the average ACC (for the Hadoop→CloudStack project pair).

6.3.4. RQ8: What is the average performance of the baseline classifier and ECLogger$_{\text{Models}}$ over all the source and target project pairs?

Motivation: In RQ 8, the performances of all 9 base classifiers and 940 ensemble models are examined over all 6 source and target project pairs. The answer to this RQ can be beneficial in identifying a dominant approach that is suitable for all types of source and target project pairs.

Approach: To answer RQ8, performances of all 9 single classifiers and 940 ensemble classifiers on all the source and target project pairs are computed. Then the average performance of all the 9 + 940 classifiers is computed. For example, to compute the average performance of a model $\mathcal{M}$, its performance on all 6 source→target project pairs is computed. The summation of these 6 values is divided by 6.

Results: Table 11 shows the results of the average performances of the best classifiers in each category, i.e. single classifier, ECLogger$_{\text{Bagging}}$, ECLogger$_{\text{MajorityVote}}$ and ECLogger$_{\text{AverageVote}}$. The results show that for individual classifiers, NB provides the best results and provides on average a 67.82% average LF and 66.85% average ACC on all source and target project pairs and, hence it is considered to be a baseline classifier for this experiment. The results of the best bagging technique are comparable with the baseline classifier, i.e., best individual classifier. ECLogger$_{\text{MajorityVote}}$ provides an average ACC of 72.84%, i.e. a 5.99% improvement in the average ACC of baseline classifier. ECLogger$_{\text{MajorityVote}}$ provides an average LF comparable to that of the baseline classifier. ECLogger$_{\text{AverageVote}}$ performs best and provides an average LF and average ACC of 70.95% and 70.93%, respectively. ECLogger$_{\text{AverageVote}}$ results in 3.12% and 6.08% improvements in the average LF and average ACC, respectively, compared to the baseline classifier.

> ECLogger$_{\text{AverageVote}}$ performs best and shows improvements of 3.12% (average LF) and 6.08% (average ACC) compared to the baseline classifier.

Overall performance summary of the three proposed approaches (RQ5–RQ8): Table 12 presents the overall performance summary of the ECLogger$_{\text{Bagging}}$, ECLogger$_{\text{AverageVote}}$ and ECLogger$_{\text{MajorityVote}}$ models. The model that provides the highest improvement (measured in average LF and average ACC) by each approach is selected for each source and target project pair. In Table 12, cells containing $\surd$ indicate that the respective model improved the results of the baseline classifier and those containing ☆ indicate the model providing the best result for the respective source and target project pair. Table 12 shows that for three source and target project pairs, i.e. Hadoop→Tomcat, Tomcat→CloudStack and Hadoop→CloudStack, the ECLogger$_{\text{AverageVote}}$ model provides the best results in terms of both average LF and average ACC. Table 12 shows that the cells associated with the ECLogger$_{\text{AverageVote}}$ model consist of a higher number of ☆ compared to those associated with ECLogger$_{\text{MajorityVote}}$ and ECLogger$_{\text{Bagging}}$. This result shows that the ECLogger$_{\text{AverageVote}}$ model results in better improvements compared to the other two approaches, i.e. ECLogger$_{\text{Bagging}}$ and ECLogger$_{\text{MajorityVote}}$. Table 12 shows that the rows consisting of CloudStack as the source

Table 12. Performance summary of $\text{ECLogger}_{\text{Bagging}}$, $\text{ECLogger}_{\text{AverageVote}}$ and $\text{ECLogger}_{\text{MajorityVote}}$. IMP: Improvement and ALGO: Algorithm

Source→Target	$\text{ECLogger}_{\text{Bagging}}$		$\text{ECLogger}_{\text{AverageVote}}$		$\text{ECLogger}_{\text{MajorityVote}}$	
	IMP. in Avg. *LF*	IMP. in Avg. *ACC*	IMP. in Avg. *LF*	IMP. in Avg. *ACC*	IMP. in Avg. *LF*	IMP. in Avg. *ACC*
CloudStack→Tomcat	√ ☆	√	√	√	√	√ ☆
Hadoop→Tomcat			√ ☆	√ ☆		
Tomcat→CloudStack	√		√ ☆	√ ☆	√	
Hadoop→CloudStack			√ ☆	√ ☆	√	√
Tomcat→Hadoop				√ ☆		
CloudStack→Hadoop	√	√ ☆	√	√	√ ☆	√

project consist of the largest number of √ symbols . This result indicates that all three proposed models provide better performance (improve the results provided by the baseline classifier) when the CloudStack project (used as the source project) is used to train the cross-project logging prediction model compared to when Tomcat or Hadoop is used to train the model. This reveals that the overall CloudStack project is more generalizable compared to the Hadoop and Tomcat projects for cross-project logging prediction. In all cases, the proposed models provide better or comparable results analogous with baseline baseline classifier.

7. Discussion

Performance of single classifiers: The log context can be viewed from two perspectives. One perspective is the log level such as debug, fatal, error, info, trace and warn. The other perspective is the programming construct and the block in which the log statements are used such as try-catch or if-else. The static code features used as predictive variables to estimate the position of the log statement depends on the log context. This study focused on a specific context such as catch-block, and in future a study will be conducted on a larger context. It was observed that classifiers LR, RF, and J48 provide better performance than the other classifiers for within-project catch-block logging prediction. The results are in compliance with results of previous studies which report RF [10] and J48 [11] as the best performing classifier for within-project logging prediction. It was observed that RF algorithm not only gives decent performance for logging prediction but it is also one of the fastest algorithms and, hence, is suitable for large datasets or time constrained environments. For cross-project catch-block logging prediction, ADT, BN, NB, and DT provide better results. The results reveal that NB performs best in three out of the six cross-project logging prediction experiments, whereas ADT, DT and BN perform best for one cross-project logging prediction experiment. Logging prediction is essentially a text classification problem and NB has shown good results for text classification problems [58]. The simplistic learning approach of NB makes is suitable for cross-project learning. NB gives good results for other cross-project prediction studies, such as cross-project defect prediction [73]. In addition to this, NB does not need a large training dataset and can handle missing data and uncorrelated features [74]. The performance of RBF is worst for cross-project catch-block logging prediction. RBF consistently provides poor performance across all the projects. RBF is a neural network-based approach [59]. RBF is known to be highly sensitive towards the training data and the dimensionality of the training data and, hence, it cannot extrapolate beyond the training data [75].

Performance of ensemble techniques: The investigated problem was the concept of integrating and combining multiple models to obtain a more accurate global predictive model in comparison to its constituent models for cross-project logging prediction [76–78]. The objective of using ensemble methods was to develop a more reliable and robust global model by reducing the generalization error [76–78]. Reducing generalization error is particularly important in this study because the experimental dataset consisting of three open-source software projects can naturally have biases due to specific logging practice guidelines and characteristics. Although there are several types of ensemble methods available, three methods were used in the experiments: bagging, average vote and majority vote. It was observed that Bagging$_{ADT}$ provided better results for cross-project catch-block logging prediction compared to other bagging combinations. The Authors believe that the better performance of bagging is due to the decreased variance of the base model [76–78]. Bagging$_{ADT}$ model was also found to be useful in other applications such as the detection of Single Nucleotide Polymorphism (SNP) associated with diseases [79]. However, a considerable increase in the model building time was observed when the bagging technique was applied and, therefore, it was not possible to build the Bagging$_{\mathrm{DT}}$ model because of its high time complexity. Hence, bagging may not be a good option for large datasets. It was also observed that for the ADA algorithms, bagging results in a considerable drop in the prediction performance, i.e. upto 28.1% in average LF on CloudStack→Hadoop project pair, as compared to the results of the ADA algorithm. The Authors believe this happened because ADA is an ensemble based algorithm and its learning method is quite different from bagging [80].

The results show that the average vote and majority vote ensemble techniques give better results as compared to that of the bagging ensemble technique. It was also observed that the time complexity of the average vote and the majority vote is considerably lower as compared to that of bagging. The reason for the higher time complexity of bagging is that the bagging technique assigns weight to the source instances by running multiple iterations of the algorithm on a sub-sampled dataset to generate the learner whereas the average vote and the majority vote generate a multiple learner in one iteration [38, 49]. However, it was observed that for majority weight ensemble technique, the majority vote of a different classifier gives the best results (measured in average LF) for different source and target project pairs. For example, classifier set ADT-BN-DT gives the highest average LF for the CloudStack→Tomcat project pair. No dominating set of classifiers which gave the best average LF for all source and target project pairs was found. For the majority vote technique classifier set, ADT-BN-DT-LR-NB gives overall 0.95% and 5.99% improvement (average over all source and target project pairs) in average LF and average ACC respectively, as compared to the results of baseline classifier. Similarly to majority vote for the average vote ensemble technique, the average vote of different set of classifiers gives the best results (measured in average LF) for different source and target project pairs. However, for the average vote technique classifier set ADA-ADT-BN-DT-LR-NB gives overall 3.12% and 6.08% improvement (average over all source and target project pairs) in average LF and average ACC respectively, as compared to the results of the baseline classifier. Comparing the improvements percentage of the majority vote and the average vote we can infer that the average vote gives better prediction results. Moreover, in the case of the equal number of votes to both the positive and negative class, the majority vote ensemble technique does random prediction of the class, whereas the average vote ensemble technique makes fair decisions. As it considers classification score for decision making. However, the Authors observed that in the literature, there are several studies which analyse the performance of the majority vote ensemble technique in different contexts, such as effect of the small-world and various diversities [81–83]. There has been much less emphasis given to the average vote ensemble technique. The Authors believe that the results of this paper mean that an indepth study of the average vote ensemble technique is necessary.

Project generalizability: During the experiments it was observed that the CloudStack project was more generalizable compared to Hadoop and Tomcat for cross-project catch-block logging prediction. Some traces of the association of the CloudStack project with both the Tomcat and Hadoop projects were found. Cloud-

Stack is the first cloud platform to join ASF[2] and is quite popular in organizations which prefer an open-source option for their cloud and big data infrastructure. It was found out that Hortonworks and the CloudStack project team were working on identifying opportunities where Hadoop components could be used to back Cloud APIs and also where Cloud APIs could be used to deploy Hadoop [84]. In addition, CloudStack uses Tomcat as its servlet container [85]. To find some examples of association in the source code of these three projects,a simple experiment was performed. For each source and target project pair, the count of logged catch-blocks that were common (i.e. have same exception types) was computed. It was observed that Tomcat→CloudStack, Tomcat→Hadoop, CloudStack→Tomcat, CloudStack→Hadoop, Hadoop→Tomcat, and Hadoop→CloudStack have 38, 44, 38, 41, 41 and 44 common unique exception types, respectively. The frequency of each of these exception types in each source project was computed. It was found out that the CloudStack→Tomcat and CloudStack→Hadoop projects have the highest frequency of these exception types, i.e.1852 and 1934, respectively. This provides an indication that the CloudStack project has some similarities in its code with both the Tomcat and Hadoop projects. The Authors believe that the CloudStack project is more generalizable than the Tomcat and Hadoop projects because it provides some support to both of these projects.

8. Threats to validity

Number and type of projects: The Tomcat, CloudStack and Hadoop projects were selected for the study. All three projects are open-source Java-based projects. However, the results may not be generalizable for all Java projects or projects written in other programing languages. Additional studies are required for other Java projects or projects written in other languages (e.g. C#, python). Only open-source projects are considered in this study; hence, the results cannot be generalized to closed-source projects. Overall, no general conclusion which would be applicable to logging prediction in all types of software applications can be drawn.

Quality of ground truth: It was assumed that logging statements inserted by the software developers of the Apache Tomcat, CloudStack and Hadoop projects were optimal. There is a possibility of errors or non-optimal logging in the code by the developers, which can affect the results of this study. However, all three of the projects are long lived and are actively maintained; hence, it can be assumed that most of the code constructs follow good logging (if not optimal). In the study 26 regular expressions[3] were used to extract the logging statements from the source code. Manual analysis reveals that all the logging statements were extracted (to the best of the Authors' knowledge). However, there is still a possibility that the regular expressions missed some types of logging statements in the source code.

Algorithm parameters: Default parameters for all the algorithms were used. Tuning classification parameters is important and can help improve the classification results. However, the Authors considered default parameters for all the algorithms as the initial step towards cross-project logging prediction. Some planned future work will encompass finding optimal parameters for each of the classification algorithms.

Sampling bias: The under-sampling of majority class instances was performed to balance the datasets. This can lead a sampling bias in the results. However, to reduce the sampling bias, 10 datasets were created and the average results over these 10 datasets were reported.

Classifier set: In this work, we explored 9 base classifiers and 3 ensemble techniques. However, there are many other classifiers (such as genetic algorithms [86]) and many other ensemble techniques (such as stacking [39] and boosting [39]), which have not been explored in this work. It is possible that a different set of algorithms would provide better results for cross-project

[2] http://nosql.mypopescu.com/post/20461845393/cloudstack-and-hadoop-a-match-made-in-the-cloud

[3] https://dl.dropboxusercontent.com/u/48972351/RegExLoggingStudy.txt

catch-block logging prediction compared to the set of algorithms explored in this work.

Computation of *LF* metric: Equation (4) was used to compute the *LF* metric. In Equation (4), parameter β can take any value ranging from 0 to ∞. If the value of β is less than 1, Equation (4) gives more weightage to precision. Similarly, the value of β greater than 1 gives more weightage to recall. A system with high precision but low recall returns few results, but most of its predicted labels are correct when compared to the training labels. A system with high recall but low precision returns many results, but most of its predicted labels are incorrect when compared to the training labels. At the time of logging prediction, a high recall and low precision system can cause the excess of log statements in the source code. However, a high precision and low recall system can cause less than required number of log statements. In this system, both excess or sparse logging in the source code is problematic. Hence, the value of β as 1, i.e. assigned equal weightage to both precision and recall, was used. Hence, this study gives preference to the classifier which optimizes both precision and recall. There are certain application domains which prefer either high-precision (low recall) or high recall (low precision) system [87]. Hence, depending upon the application domain either high precision (low recall) or high recall (low precision) system may be more suitable. In such cases the value of β needs to be adjusted accordingly. In this work, we have not compared the performance of different classifiers for different values of β.

9. Conclusion and future work

In this paper, the Authors propose ***ECLogger***, an ensemble-based, cross-project, catch-block logging prediction framework. ECLogger uses 9 base classifiers (AdaBoostM1, ADTree, Bayesian network, decision table, J48, logistic regression, Naïve Bayes, random forest and radial basis function network). ECLogger combines these algorithms with three ensemble techniques, i.e. bagging, average vote and majority vote. In the study 8 $\text{ECLogger}_{\text{Bagging}}$, 466 $\text{ECLogger}_{\text{AverageVote}}$ and 466 $\text{ECLogger}_{\text{MajorityVote}}$ models were created. The performance of ECLogger on three open-source Java projects: Tomcat, CloudStack and Hadoop was evaluated. The results of the comparison of $\text{ECLogger}_{\text{Bagging}}$, $\text{ECLogger}_{\text{AverageVote}}$ and $\text{ECLogger}_{\text{MajorityVote}}$ with baseline classifiers were presented. $\text{ECLogger}_{\text{Bagging}}$, $\text{ECLogger}_{\text{AverageVote}}$ and $\text{ECLogger}_{\text{MajorityVote}}$ show maximum improvements of 4.6%, 7.04% and 5.39% in average *LF*, respectively, in comparison to the baseline classifier. Overall, the $\text{ECLogger}_{\text{AverageVote}}$ model performs better than $\text{ECLogger}_{\text{Bagging}}$ and $\text{ECLogger}_{\text{MajorityVote}}$. The experimental results show that the CloudStack project is more generalizable for cross-project catch-block logging prediction than the Tomcat and Hadoop projects.

In the future, there are plans to evaluate to evaluate ECLogger on datasets from more software projects, i.e. closed-source applications and projects from other programing languages (i.e. C, C++, and C#). There are also plans to extend the functionality of ECLogger for other types of code constructs, such as if-blocks. The Authors will also work to improve the performance of ECLogger using other ensemble techniques, such as stacking, which is found to be useful in bug assignment problem [88]. Apart from this work will be conducted on tuning various classifier parameters to obtain the optimal classification performance [89]. In addition to this, the identification of the most productive feature will be examined using various feature selection techniques to reduce the model building time and to further improve the performance. The Authors believe that their research can be applied or transferred into practice by building a development environment tool, such as an Eclipse or Visual Studio plug-in. Future plans encompass the development of an ECLogger plug-in for Eclipse IDE which will give a logging suggestion to software developers. at the time of coding. The advantage of a plug-in based implementation is that the developers can use the tool as part of their existing infrastructure and process and do not need to learn or install a completely new tool.

References

[1] D. Yuan, S. Park, and Y. Zhou, "Characterizing logging practices in open-source software," in *Proceedings of the 34th International Conference on Software Engineering*, 2012, pp. 102–112.

[2] B. Sharma, V. Chudnovsky, J.L. Hellerstein, R. Rifaat, and C.R. Das, "Modeling and synthesizing task placement constraints in Google compute clusters," in *Proceedings of the 2Nd ACM Symposium on Cloud Computing*. New York: ACM, 2011, pp. 3:1–3:14.

[3] K. Nagaraj, C. Killian, and J. Neville, "Structured comparative analysis of systems logs to diagnose performance problems," in *Proceedings of the 9th USENIX Conference on Networked Systems Design and Implementation*, 2012, pp. 353–366. [Online]. https://www.usenix.org/system/files/conference/nsdi12/nsdi12-final61.pdf

[4] Q. Fu, J.G. Lou, Y. Wang, and J. Li, "Execution anomaly detection in distributed systems through unstructured log analysis," in *Proceedings of the 2009 Ninth IEEE International Conference on Data Mining*. Washington: IEEE Computer Society, 2009, pp. 149–158.

[5] Z.M. Jiang, A.E. Hassan, G. Hamann, and P. Flora, "Automatic identification of load testing problems," in *IEEE International Conference on Software Maintenance*, Sep. 2008, pp. 307–316.

[6] Z.M. Jiang, A.E. Hassan, G. Hamann, and P. Flora, "Automated performance analysis of load tests," in *IEEE International Conference on Software Maintenance*, Sep. 2009, pp. 125–134.

[7] Blackberry enterprise server logs submission, BlackBerry Limited. [Online]. https://salesforce.services.blackberry.com/webforms/beslogs [Accessed 4 June 2016].

[8] Q. Fu, J. Zhu, W. Hu, J.G. Lou, R. Ding, Q. Lin, D. Zhang, and T. Xie, "Where do developers log? An empirical study on logging practices in industry," in *Companion Proceedings of the 36th International Conference on Software Engineering*, 2014, pp. 24–33.

[9] S. Lal, N. Sardana, and A. Sureka, "LogOptPlus: Learning to optimize logging in catch and if programming constructs," in *40th Annual Computer Software and Applications Conference COMPSAC*, Vol. 1, Jun. 2016, pp. 215–220.

[10] S. Lal and A. Sureka, "LogOpt: Static feature extraction from source code for automated catch block logging prediction," in *9th India Software Engineering Conference ISEC*, 2016, pp. 151–155.

[11] J. Zhu, P. He, Q. Fu, H. Zhang, M. Lyu, and D. Zhang, "Learning to log: Helping developers make informed logging decisions," in *37th IEEE International Conference on Software Engineering ICSE*, Vol. 1, May 2015, pp. 415–425.

[12] A. Grabner, Top Tomcat performance problems part 2: Bad coding, inefficient logging and exceptions. [Online]. http://apmblog.dynatrace.com/2016/03/08/top-tomcat-performance-problems-part-2-bad-coding-inefficient-logging-exceptions/ [Accessed 31 May 2015].

[13] W. Shang, M. Nagappan, and A.E. Hassan, "Studying the relationship between logging characteristics and the code quality of platform software," *Empirical Software Engineering*, Vol. 20, No. 1, 2015, pp. 1–27.

[14] J. Nam, S.J. Pan, and S. Kim, "Transfer defect learning," in *35th International Conference on Software Engineering ICSE*, May 2013, pp. 382–391.

[15] A.T. Mısırlı, A.B. Bener, and B. Turhan, "An industrial case study of classifier ensembles for locating software defects," *Software Quality Journal*, Vol. 19, No. 3, 2011, pp. 515–536.

[16] L. Mariani and F. Pastore, "Automated identification of failure causes in system logs," in *Software Reliability Engineering, 2008. ISSRE 2008. 19th International Symposium on*, Nov. 2008, pp. 117–126.

[17] D. Yuan, H. Mai, W. Xiong, L. Tan, Y. Zhou, and S. Pasupathy, "SherLog: error diagnosis by connecting clues from run-time logs," in *Proceedings of the Fifteenth Edition of ASPLOS on Architectural Support for Programming Languages and Operating Systems*. New York: ACM, 2010, pp. 143–154.

[18] W. Xu, L. Huang, A. Fox, D. Patterson, and M.I. Jordan, "Detecting large-scale system problems by mining console logs," in *Proceedings of the ACM SIGOPS 22nd Symposium on Operating Systems Principles*, 2009, pp. 117–132.

[19] M. Montanari, J.H. Huh, D. Dagit, R. Bobba, and R.H. Campbell, "Evidence of log integrity in policy-based security monitoring," in *IEEE/IFIP 42nd International Conference on Dependable Systems and Networks Workshops DSN-W*. IEEE, 2012, pp. 1–6.

[20] G. Lee, J. Lin, C. Liu, A. Lorek, and D. Ryaboy, "The unified logging infrastructure for data analytics at Twitter," *Proc. VLDB Endow.*, Vol. 5, No. 12, Aug. 2012, pp. 1771–1780.

[21] Logstash homepage, Elasticsearch. [Online]. https://www.elastic.co/products/logstash/ [Accessed 27 July 2016].

[22] Splunk homepage, Splunk, Inc. [Online]. http://www.splunk.com/ [Accessed 27 July 2016].

[23] S. Kabinna, C.P. Bezemer, W. Shang, and A.E. Hassan, "Examining the stability of logging statements," in *The 23rd IEEE International Conference on Software Analysis, Evolution, and Reengineering SANER*, 2016.

[24] D. Yuan, J. Zheng, S. Park, Y. Zhou, and S. Savage, "Improving software diagnosability via log enhancement," in *Proceedings of the Sixteenth International Conference on Architectural Support for Programming Languages and Operating Systems*. New York: ACM, 2011, pp. 3–14.

[25] D. Yuan, S. Park, P. Huang, Y. Liu, M.M. Lee, X. Tang, Y. Zhou, and S. Savage, "Be conservative: Enhancing failure diagnosis with proactive logging," in *Proceedings of the 10th USENIX Conference on Operating Systems Design and Implementation*, 2012, pp. 293–306.

[26] T. Nurkiewicz, 10 tips for proper application logging. [Online]. http://www.javacodegeeks.com/2011/01/10-tips-proper-application-logging.html [Accessed 19 Oct 2015].

[27] Why does the TRACE level exists, and when should I use it rather than DEBUG? [Online]. http://programmers.stackexchange.com/questions/279690/why-does-the-trace-level-exists-and-when-should-i-use-it-rather-than-debug [Accessed 22 Oct 2015].

[28] T. Menzies, J. Greenwald, and A. Frank, "Data mining static code attributes to learn defect predictors," *IEEE Transactions on Software Engineering*, Vol. 33, No. 1, Jan. 2007, pp. 2–13.

[29] S. Kim, E.J.W. Jr., and Y. Zhang, "Classifying software changes: Clean or buggy?" *IEEE Transactions on Software Engineering*, Vol. 34, No. 2, Mar. 2008, pp. 181–196.

[30] Y. Zhang, D. Lo, X. Xia, and J. Sun, "An empirical study of classifier combination for cross-project defect prediction," in *39th Annual Computer Software and Applications Conference COMPSAC*, Vol. 2, Jul. 2015, pp. 264–269.

[31] Y. Hu, X. Zhang, E. Ngai, R. Cai, and M. Liu, "Software project risk analysis using Bayesian networks with causality constraints," *Decision Support Systems*, Vol. 56, 2013, pp. 439–449.

[32] X. Xia, D. Lo, X. Wang, X. Yang, S. Li, and J. Sun, "A comparative study of supervised learning algorithms for re-opened bug prediction," in *17th European Conference on Software Maintenance and Reengineering CSMR*. IEEE, 2013, pp. 331–334.

[33] T.G. Dietterich, "Ensemble learning," in *The handbook of brain theory and neural networks*, 2nd ed., M.A. Arbib, Ed. MIT Press: Cambridge, MA, 2002, pp. 405–408.

[34] Z.H. Zhou, "Ensemble learning," *Encyclopedia of Biometrics*, 2015, pp. 411–416.

[35] L. Breiman, "Bagging predictors," *Machine Learning*, Vol. 24, No. 2, 1996, pp. 123–140.

[36] L. Breiman, "Random forests," *Mach. Learn.*, Vol. 45, No. 1, Oct. 2001, pp. 5–32.

[37] Y. Freund and R.E. Schapire, "A decision-theoretic generalization of on-line learning and an application to boosting," *Journal of Computer and System Sciences*, Vol. 55, No. 1, Aug. 1997, pp. 119–139.

[38] J.R. Quinlan, "Bagging, boosting, and C4.S," in *Proceedings of the Thirteenth National Conference on Artificial Intelligence – Volume 1*. AAAI Press, 1996, pp. 725–730.

[39] J. Han, M. Kamber, and J. Pei, *Data Mining: Concepts and Techniques*, 3rd ed. San Francisco: Morgan Kaufmann Publishers Inc., 2011.

[40] D.H. Wolpert, "Stacked generalization," *Neural networks*, Vol. 5, No. 2, 1992, pp. 241–259.

[41] A. Panichella, R. Oliveto, and A.D. Lucia, "Cross-project defect prediction models: L'union fait la force," in *IEEE Conference on Software Maintenance, Reengineering and Reverse Engineering CSMR-WCRE*, Feb. 2014, pp. 164–173.

[42] X. Xia, D. Lo, E. Shihab, X. Wang, and X. Yang, "ELBlocker: Predicting blocking bugs with ensemble imbalance learning," *Information and Software Technology*, Vol. 61, 2015, pp. 93–106.

[43] W. Dai, Q. Yang, G.R. Xue, and Y. Yu, "Boosting for transfer learning," in *Proceedings of the 24th International Conference on Machine Learning*. New York: ACM, 2007, pp. 193–200.

[44] S.J. Pan, I.W. Tsang, J.T. Kwok, and Q. Yang, "Domain adaptation via transfer component analysis," *IEEE Transactions on Neural Networks*, Vol. 22, No. 2, Feb. 2011, pp. 199–210.

[45] X. Xia, D. Lo, S. McIntosh, E. Shihab, and A.E. Hassan, "Cross-project build co-change prediction," in *22nd International Conference on Software Analysis, Evolution, and Reengineering SANER*, Mar. 2015, pp. 311–320.

[46] S.J. Pan, X. Ni, J.T. Sun, Q. Yang, and Z. Chen, "Cross-domain sentiment classification via spectral feature alignment," in *Proceedings of the 19th international conference on World wide web*. ACM, 2010, pp. 751–760.

[47] Y. Freund and R.E. Schapire, Experiments with a new boosting algorithm, (1996).[Online].

http://www.public.asu.edu/~jye02/CLASSES/Fall-2005/PAPERS/boosting-icml.pdf

[48] M. Hall, E. Frank, G. Holmes, B. Pfahringer, P. Reutemann, and I.H. Witten, "The WEKA data mining software: An update," *SIGKDD Explor. Newsl.*, Vol. 11, No. 1, Nov. 2009, pp. 10–18.

[49] M. Sewell, "Ensemble learning," *RN*, Vol. 11, No. 02, 2008.

[50] Y. Freund and L. Mason, "The alternating decision tree learning algorithm," in *Proceedings of the Sixteenth International Conference on Machine Learning*, 1999, pp. 124–133.

[51] K. Murphy, A brief introduction to graphical models and Bayesian networks. [Online]. http://www.cs.ubc.ca/~murphyk/Bayes/bnintro.html [Accessed 20 March 2016].

[52] T.D. Nielsen and F.V. Jensen, *Bayesian networks and decision graphs.* Springer Science & Business Media, 2009.

[53] R. Kohavi, "The power of decision tables," in *Machine Learning: ECML-95*, ser. Lecture Notes in Computer Science (Lecture Notes in Artificial Intelligence), N. Lavrac and S. Wrobel, Eds. Springer, 1995, Vol. 912, pp. 174–189.

[54] G.H. John, R. Kohavi, K. Pfleger *et al.*, "Irrelevant features and the subset selection problem," in *Machine Learning: Proceedings of the Eleventh International Chonference*, 1994, pp. 121–129.

[55] A. Padhye, Classification methods. [Online]. http://www.d.umn.edu/~padhy005/Chapter5.html [Accessed 20 March 2016].

[56] D.W. Hosmer and S. Lemeshow, "Introduction to the logistic regression model," *Applied Logistic Regression, Second Edition*, 2000, pp. 1–30.

[57] D.D. Lewis, "Naive (Bayes) at forty: The independence assumption in information retrieval," in *Proceedings of the 10th European Conference on Machine Learning.* London, UK, UK: Springer-Verlag, 1998, pp. 4–15.

[58] S. Shivaji, E.J. Whitehead, R. Akella, and S. Kim, "Reducing features to improve code change-based bug prediction," *IEEE Transactions on Software Engineering*, Vol. 39, No. 4, 2013, pp. 552–569.

[59] M.D. Buhmann and M.D. Buhmann, *Radial Basis Functions.* New York: Cambridge University Press, 2003.

[60] Python NLTK library, NLTK Project. [Online]. http://www.nltk.org/ [Accessed 19 March 2016].

[61] P. Krill, Java regains spot as most popular language in developer index. [Online]. http://www.infoworld.com/article/2909894/application-development/java-back-at-1-in-language-popularity-assessment.html [Accessed 19 March 2016].

[62] Apache project homepage, The Apache Software Foundation. [Online]. https://commons.apache.org/proper/commons-logging/ [Accessed 18 March 2016].

[63] Cloudstack project homepage, The Apache Software Foundation. [Online]. https://cloudstack.apache.org/ [Accessed 18 March 2016].

[64] Hadoop project homepage, The Apache Software Foundation. [Online]. http://hadoop.apache.org/ [Accessed 18 March 2016].

[65] B. Chen and Z.M. (Jack) Jiang, "Characterizing logging practices in Java-based open source software projects – a replication study in Apache Software Foundation," *Empirical Software Engineering*, 2016, pp. 1–45.

[66] D. Correa and A. Sureka, "Chaff from the wheat: Characterization and modeling of deleted questions on Stack Overflow," in *Proceedings of the 23rd International Conference on World Wide Web.* New York: ACM, 2014, pp. 631–642.

[67] S. Shivaji, E.J.W. Jr., R. Akella, and S. Kim, "Reducing features to improve bug prediction," in *Proceedings of the 2009 IEEE/ACM International Conference on Automated Software Engineering.* Washington: IEEE Computer Society, 2009, pp. 600–604.

[68] C.D. Manning, P. Raghavan, and H. Schütze, *Introduction to Information Retrieval.* New York: Cambridge University Press, 2008.

[69] Y. Tian, J. Lawall, and D. Lo, "Identifying Linux bug fixing patches," in *Proceedings of the 34th International Conference on Software Engineering.* Piscataway: IEEE Press, 2012, pp. 386–396.

[70] H. Valdivia Garcia and E. Shihab, "Characterizing and predicting blocking bugs in open source projects," in *Proceedings of the 11th Working Conference on Mining Software Repositories.* New York: ACM, 2014, pp. 72–81.

[71] F. Zhang, Q. Zheng, Y. Zou, and A.E. Hassan, "Cross-project defect prediction using a connectivity-based unsupervised classifier," in *Proceedings of the 38th International Conference on Software Engineering.* New York: ACM, 2016, pp. 309–320.

[72] G. Zhou, D. Shen, J. Zhang, J. Su, and S. Tan, "Recognition of protein/gene names from text using an ensemble of classifiers," *BMC bioinformatics*, Vol. 6, No. 1, 2005, p. 1.

[73] R.F. Satin, I.S. Wiese, and R. Ré, "An exploratory study about the cross-project defect prediction: Impact of using different classification algorithms and a measure of performance in

building predictive models," in *Latin American Computing Conference CLEI*. IEEE, 2015, pp. 1–12.

[74] A. Jordan, "On discriminative vs. generative classifiers: A comparison of logistic regression and Naive Bayes," *Advances in neural information processing systems*, Vol. 14, 2002, p. 841.

[75] Neural networks, StatSoft, Inc. [Online]. http://www.fmi.uni-sofia.bg/fmi/statist/education/textbook/eng/stneunet.html [Accessed 30 July 2016].

[76] T.G. Dietterich, "Ensemble methods in machine learning," in *Proceedings of the First International Workshop on Multiple Classifier Systems*. London, UK, UK: Springer-Verlag, 2000, pp. 1–15.

[77] S.B. Kotsiantis, "Supervised machine learning: A review of classification techniques," in *Proceedings of the 2007 Conference on Emerging Artificial Intelligence Applications in Computer Engineering: Real Word AI Systems with Applications in eHealth, HCI, Information Retrieval and Pervasive Technologies*. Amsterdam, The Netherlands, The Netherlands: IOS Press, 2007, pp. 3–24.

[78] Z.H. Zhou, *Ensemble methods: foundations and algorithms*. CRC Press, 2012.

[79] R.T. Guy, P. Santago, and C.D. Langefeld, "Bootstrap aggregating of alternating decision trees to detect sets of SNPs that associate with disease," *Genetic epidemiology*, Vol. 36, No. 2, 2012, pp. 99–106.

[80] E. Bauer and R. Kohavi, "An empirical comparison of voting classification algorithms: Bagging, boosting, and variants," *Machine Learning*, Vol. 36, No. 1-2, 1999, pp. 105–139.

[81] G. Brown and L.I. Kuncheva, "Good and bad diversity in majority vote ensembles," in *International Workshop on Multiple Classifier Systems*. Springer, 2010, pp. 124–133.

[82] P.R. Campos, V.M. de Oliveira, and F.B. Moreira, "Small-world effects in the majority-vote model," *Physical Review E*, Vol. 67, No. 2, 2003, p. 026104.

[83] L.I. Kuncheva, C.J. Whitaker, C.A. Shipp, and R.P. Duin, "Limits on the majority vote accuracy in classifier fusion," *Pattern Analysis & Applications*, Vol. 6, No. 1, 2003, pp. 22–31.

[84] Sheng, Cloudstack and Hadoop: A match made in the cloud. [Online]. http://nosql.mypopescu.com/post/20461845393/cloudstack-and-hadoop-a-match-made-in-the-cloud#fn:2-fn-Sheng/ [Accessed 27 July 2016].

[85] Additional installation options, The Apache Software Foundation. [Online]. http://docs.cloudstack.apache.org/projects/cloudstack-installation/en/4.9/optional_installation.html [Accessed 27 July 2016].

[86] M. Mitchell, *An introduction to genetic algorithms*. MIT Press, 1998.

[87] C. Zhai and S. Massung, *Text Data Management and Analysis: A Practical Introduction to Information Retrieval and Text Mining*. Association for Computing Machinery and Morgan & Claypool Publishers, 2016.

[88] L. Jonsson, M. Borg, D. Broman, K. Sandahl, S. Eldh, and P. Runeson, "Automated bug assignment: Ensemble-based machine learning in large scale industrial contexts," *Empirical Software Engineering*, 2015, pp. 1–46.

[89] M. Borg, "TuneR: a framework for tuning software engineering tools with hands-on instructions in R," *Journal of Software: Evolution and Process*, Vol. 28, No. 6, 2016, pp. 427–459.