e-Informatica

software engineering journal

2017 volume 11 issue 1

e-Informatica

Wrocław University
of Science and Technology

Editors
Zbigniew Huzar (*Zbigniew.Huzar@pwr.edu.pl*)
Lech Madeyski (*Lech.Madeyski@pwr.edu.pl, http://madeyski.e-informatyka.pl*)

Department of Software Engineering, Faculty of Computer Science and Management
Wrocław University of Science and Technology, 50-370 Wrocław, Wybrzeże Wyspiańskiego 27, Poland

e-Informatica Software Engineering Journal
www.e-informatyka.pl, DOI: 10.5277/e-informatica

Editorial Office Manager: Wojciech Thomas

Proofreader: Anna Tyszkiewicz

Typeset by Wojciech Myszka with the $\LaTeX\,2_\varepsilon$ Documentation Preparation System

OFICYNA WYDAWNICZA POLITECHNIKI WROCŁAWSKIEJ
Wybrzeże Wyspiańskiego 27, 50-370 Wrocław
www.oficyna.pwr.edu.pl;
e-mail: oficwyd@pwr.edu.pl; zamawianie.ksiazek@pwr.edu.pl

ISSN 1897-7979

Print and binding: beta-druk, www.betadruk.pl

Editorial Board

Contents

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: ECLogger$_{\text{Bagging}}$, ECLogger$_{\text{AverageVote}}$, and 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. 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 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), $\alpha, w_1, w_2, \ldots w_n$ represent the linear combination coefficients, and $x_1, x_2, \ldots, x_n$ 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.

$$P(d_i) = \frac{e^{\alpha + w_1 x_1 + w_2 x_2 + \cdots + w_n x_n}}{1 + e^{w_1 x_1 + w_2 x_2 + \cdots + w_n x_n}} \tag{1}$$

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

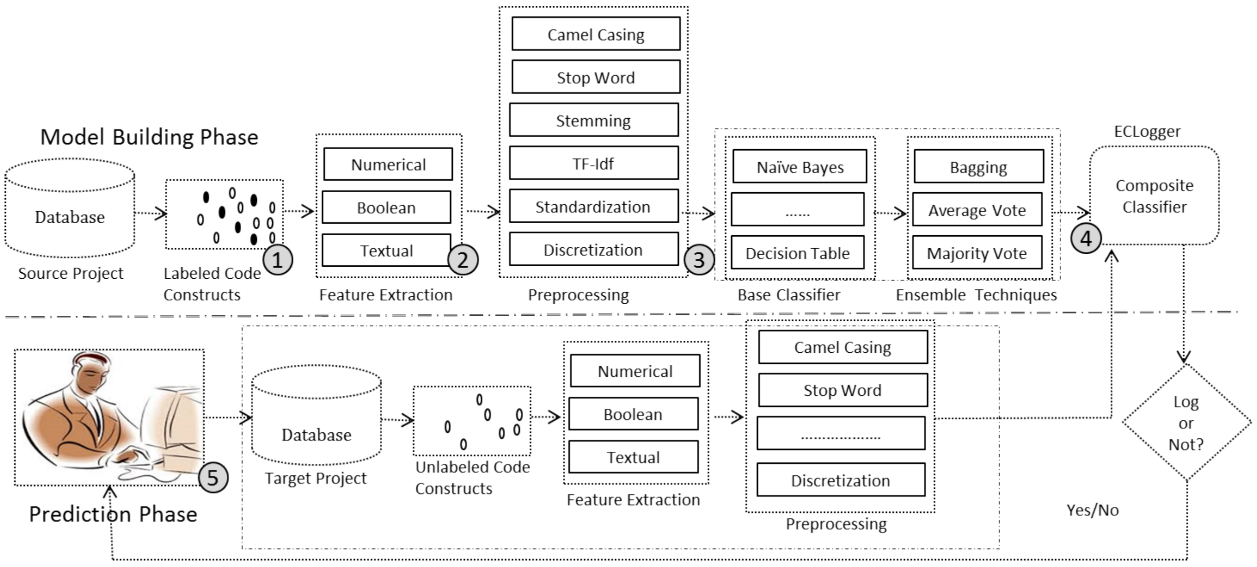


Figure 1. Overview of the proposed ECLogger framework

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

```
 1: procedure ECLogger
 2:     $\mathcal{P} = \{\mathcal{P}_{\mathrm{T}}, \mathcal{P}_{\mathrm{C}}, \mathcal{P}_{\mathrm{H}}\}$
 3:     $\mathcal{A} = \{\mathcal{A}_{\mathrm{ADA}}, \mathcal{A}_{\mathrm{ADT}}, \mathcal{A}_{\mathrm{BN}}, \mathcal{A}_{\mathrm{DT}}, \mathcal{A}_{\mathrm{J48}}, \mathcal{A}_{\mathrm{LR}}, \mathcal{A}_{\mathrm{NB}}, \mathcal{A}_{\mathrm{RF}}, \mathcal{A}_{\mathrm{RBN}}\}$
 4:     $\mathcal{EA} = \{\mathcal{EA}_{\mathrm{BA}}, \mathcal{EA}_{\mathrm{MV}}, \mathcal{EA}_{\mathrm{AV}}\}$
 5:     M=10
 6:     $\mathcal{CS} = \{3, 4, 5, 6, 7, 8, 9\}$
 7:     for all $S \in \mathcal{P}$ do
 8:         for all $T \in \mathcal{P}$ do
 9:             if $S \neq T$ then
10:                 $\mathcal{SP} = S$, $\mathcal{TP} = T$
11:                 $\mathcal{CB}_{\mathcal{SP}}$ = ReadCompleteData($\mathcal{SP}$)
12:                 $\mathcal{FV}^{I}_{\mathcal{SP}}$ = ExtractFeatures($\mathcal{CB}_{\mathcal{SP}}$)
13:                 $\mathcal{FV}^{F}_{\mathcal{SP}}$ = Preprocess($\mathcal{FV}^{I}_{\mathcal{SP}}$)
14:                 ECLogger$_{\mathrm{Model}}$[] = BuildModel($\mathcal{FV}^{F}_{\mathcal{SP}}, \mathcal{A}, \mathcal{EA}, \mathcal{CS}$)
15:                 $\mathcal{CB}_{\mathcal{TP}}[M]$ = ReadBalanceData($\mathcal{TP}$)
16:                 for $i = 1$ to size(ECLogger$_{\mathrm{Model}}$) do
17:                     for $j = 1$ to $M$ do
18:                         $\mathcal{FV}^{I}_{\mathcal{TP}}$ = ExtractFeatures($(\mathcal{CB})_{\mathcal{TP}}[j]$)
19:                         $\mathcal{FV}^{F}_{\mathcal{TP}}$ = Preprocess($\mathcal{FV}^{I}_{\mathcal{TP}}$)
20:                         $\mathcal{PD}[i][j]$ = ApplyModel($\mathcal{FV}^{F}_{\mathcal{TP}}$, ECLogger$_{\mathrm{Models}}[j]$)
21:                     $\mathcal{AR}[i] = \frac{\sum_{j=1}^{M} \mathcal{PD}[i][j]}{M}$
22:                 $\mathcal{BM}_{\mathcal{SP}\to\mathcal{TP}}$ = FindBestModel(AR, ECLogger$_{\mathrm{Models}}$)
23: procedure ReadCompleteData($P$)
24:     $\mathcal{CB}$ = ReadCatchBlocks($P$)
25:     return $\mathcal{CB}$
26: procedure ReadBalanceData($P, M$)
27:     $\mathcal{CB}$ = ReadCatchBlocks($P$)
28:     $\widehat{\mathcal{CB}}$ = Randomize($\mathcal{CB}$)
29:     $\mathcal{BS}$[] = Generate_M_BalanceSamples($\widehat{\mathcal{CB}}$)
30:     return $\mathcal{BS}$
31: procedure ExtractFeatures($\mathcal{CB}$)
32:     ${}_{T}\mathcal{FV}$ = getTextualFeatures($\mathcal{CB}$)
33:     ${}_{N}\mathcal{FV}$ = getNumericFeatures($\mathcal{CB}$)
34:     ${}_{B}\mathcal{FV}$ = getBooleanFeatures($\mathcal{CB}$)
35:     $\mathcal{FV} = \{{}_{T}\mathcal{FV},\ {}_{N}\mathcal{FV},\ {}_{B}\mathcal{FV}\}$
36:     return $\mathcal{FV}$
37: procedure Preprocess($\mathcal{FV}$)
38:     ${}_{T}\hat{\mathcal{FV}}$ = TF_IDFConversion(Stemming(StopWordRemoval(CamelCaseSeparation(${}_{T}\mathcal{FV}$))))
39:     if It is Test Data then
40:         ${}_{T}\widetilde{\mathcal{FV}}$ = FilterFeatureNotTrainData(${}_{T}\widehat{\mathcal{FV}}$)
41:     else
42:         ${}_{T}\widetilde{\mathcal{FV}} = {}_{T}\widehat{\mathcal{FV}}$
43:     $\mathcal{FV}^{F}$ = Discretization(Standardization(Combine(${}_{T}\widetilde{\mathcal{FV}},\ {}_{B}\mathcal{FV},\ {}_{N}\mathcal{FV}$)))
44:     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	$\text{ECLogger}_{\text{Models}}$	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 $\text{ECLogger}_{\text{Bagging}}$ 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. $\text{Bagging}_{\text{ADA}}$, $\text{Bagging}_{\text{ADT}}$, $\text{Bagging}_{\text{BN}}$, $\text{Bagging}_{\text{J48}}$, $\text{Bagging}_{\text{LR}}$, $\text{Bagging}_{\text{NB}}$, $\text{Bagging}_{\text{RF}}$ and $\text{Bagging}_{\text{RBF}}$. $\text{Bagging}_{\text{ADA}}$ 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 $\text{ECLogger}_{\text{AverageVote}}$ models are created. Similarly to $\text{ECLogger}_{\text{AverageVote}}$, 466 $\text{ECLogger}_{\text{MajorityVote}}$ models are created. 940 distinct ECLogger models ($\text{ECLogger}_{\text{Models}}[]$) are created for cross-project logging prediction (line 14 in Algorithm 1).

4.2. Phase 2: (prediction)

Prediction (step 5): In the prediction phase, $\text{ECLogger}_{\text{Models}}$ 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 ($\mathcal{FV}^F_{SP}$) 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 ($\mathcal{FV}^F_{TP}$) 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$_{\text{Model}}$($\mathcal{BM}_{SP\rightarrow TP}$) 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, $l \to l$ (true positive).
2. Predicting logged code construct as non-logged, $l \to n$ (false negative).
3. Predicting non-logged code constructs as non-logged, $n \to n$ (true negative).
4. Predicting non-logged code constructs as logged, $n \to l$ (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 ($C_{l\to l}$), logged code constructs falsely predicted as non-logged ($C_{l\to n}$), non-logged code constructs predicted as non-logged ($C_{n\to n}$), and non-logged code constructs predicted and logged ($C_{n\to l}$) 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.

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

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?

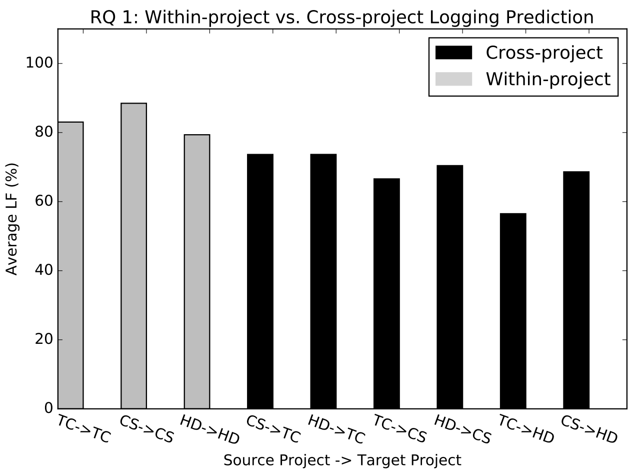


Figure 2. The highest average *LF* of within-project and cross-project logging predictions. CS: CloudStack, TC: Tomcat, and HD: Hadoop

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).

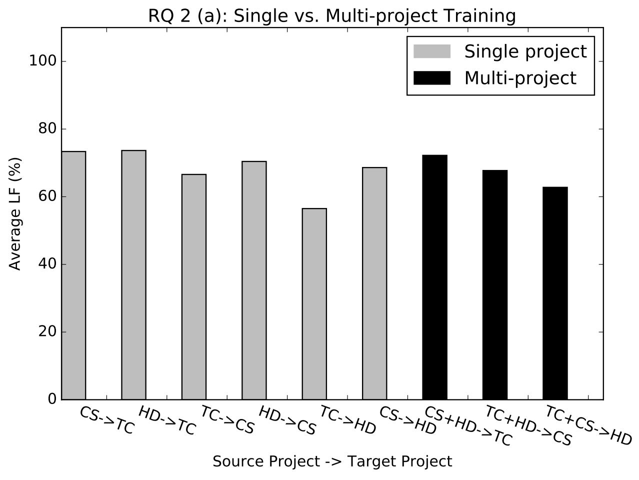
There is no dominant approach among the single-project and multi-project cross-project catch-block prediction models.

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

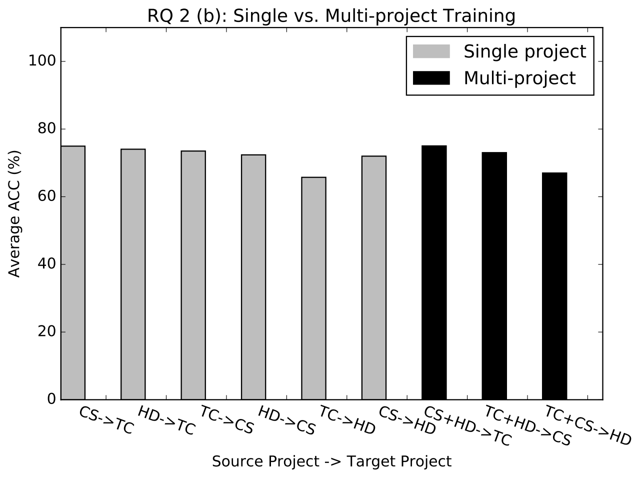
Motivation: In RQ3, the objective is to examine the performance of individual classifiers on multiple evaluation metrics. The evaluation of a predictive model or a classifier can be performed using several metrics or measures, and the selected set of metrics depends on the classification task and problem. 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

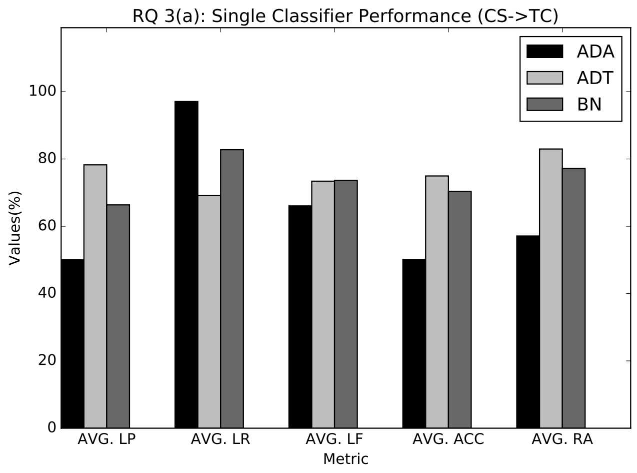


(a) Average *LF* (%)

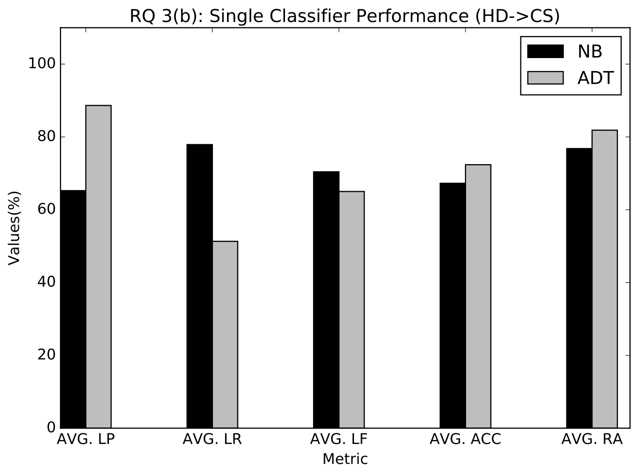


(b) Average *ACC* (%)

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.

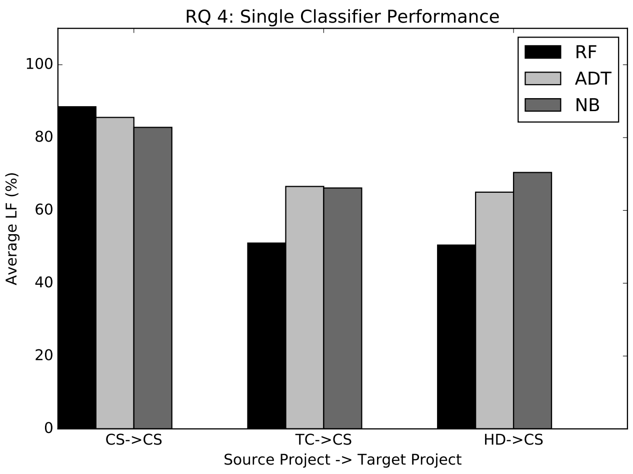


Figure 5. Performance of RF, ADT, and NB classifiers for within-project and cross-project catch-block logging predictions

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 √ 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.

e-Informatica Software Engineering Journal, Volume 11, Issue 1, 2017, pages: 39–57, DOI 10.5277/e-Inf170102

Experience Report: Introducing Kanban into Automotive Software Project[‡]

Marek Majchrzak[*], Łukasz Stilger[**]
[*]*Faculty of Computer Science and Management, Wroclaw University of Science and Technology*
[**]*Capgemini Polska*
marek.majchrzak@pwr.edu.pl, lukasz.stilger@capgemini.com

Abstract

The boundaries between traditional and agile approach methods are disappearing. A significant number of software projects require a continuous implementation of tasks without dividing them into sprints or strict project phases. Customers expect more flexibility and responsiveness from software vendors in response to the ever-changing business environment. To achieve better results in this field, Capgemini has begun using the Lean philosophy and Kanban techniques.
The following article illustrates examples of different uses of Kanban and the main stakeholder of the process. The article presents the main advantages of transparency and ways to improve the customer co-operation as well as stakeholder relationships. The Authors try to visualise all of the elements in the context of the project.
There is also a discussion of different approaches in two software projects. The article focuses on the main challenges and the evolutionary approach used. An attempt is made to answer the question how to convince both the team as well as the customer, and how to optimise ways to achieve great results.

Keywords: kanban, lean, automotive, scrum, agile

1. Introduction

Lean thinking is important because it can dramatically reduce waste and introduce built-in quality in every process step. It has been shown that when applying this approach in the manufacturing or service organisation, the productivity has at least doubled. Moreover, this method also significantly reduces delivery time for new products and decreases overall costs [1, 2].

The paper also describes two software projects in the automotive industry, which have employed the Kanban technique in an evolutionary way. In each of these cases, Kanban was used to optimise a different process and was motivated by other business problems. However, the mutual characteristic was the simplification of processes and evolutionary adaptation of both the developer teams and the collaborating client teams.

The idea was to make the communication more efficient, and thanks to that do a project more efficiently. It is necessary to continuously identify bottlenecks and wastes. With the lean principles and Kanban specific practices it is possible to visualise the state of the project and focus on the process.

This paper is organised as follows: an introduction to Lean Software Development and Kanban technique in Sections 2 and 3. Section 4 presents the related work. Section 5 describes projects and their Kanban technique adoption history. General results and conclusions are discussed in Section 6.

[‡]This paper was originally published in the KKIO 2015 proceedings P. Kosiuczenko and M. Śmiałek (Eds.), "From Requirements to Software: Research and Practice".

2. Lean software development

The principles of Lean thinking focus on value added for the customer [3]. By removing the unnecessary processes, activities and artefacts, and on the other hand organising work as a continuous flow, which recombines labour into cross-functional teams dedicated to that activity and constant improvements across the entire company, we have been able to develop, fabricate and sell with half or less of the human effort, tools and overall costs. By introducing Lean thinking and its associated style of operation, we have been able to react faster and more flexibly to the ever-changing needs of our Clients and the modern market. Lean thinking requires continuous learning, growth and most importantly, commitment and understanding from the personnel of any level including management.

Lean Software Development is the application of Lean Thinking to the software development process. The Poppendieck and Poppendieck [4] illustrated how many of the Lean principles and practices could be used in the software engineering context. They proposed seven principles eliminating and managing the waste in software development:

– Eliminate Waste – Do only what adds value for a customer, and do it without delay.
– Amplify Learning – Use frequent iterations and regular releases to provide feedback.
– Delay Commitment – Make decisions at the last responsible moment.
– Deliver Fast – The measure of the maturity of an organisation is the speed at which it can repeatedly and reliably respond to customer needs.
– Empower the Team – Assemble an expert workforce, provide technical leadership and delegate the responsibility to the workers.
– Build Integrity In – Have the disciplines in place to assure that a system will delight customers both upon initial delivery and over an extended period.
– See the Whole – Use measurements and incentives focused on achieving the overall goal.

The automotive software projects use proven technologies, mostly not the latest development techniques. It is because the business functionality is more complex than the other software projects. There are many external system interfaces which extend at the same time. Lean principles offer support to optimise the processes with focus on following aspects:

– Time to market – it is crucial for the software used in the automotive sector to keep stability and compatibility. However, the rapid development of industry requires also more flexibility of software. Automotive companies constantly release new car models or versions of the same car.
– Stakeholder management – at the same time, stakeholder structure was extended in big organisations. Each stakeholder (or group of interested parties) has their goals which need to converge.
– Domain knowledge – another characteristic dimension in software for the automotive sector is domain knowledge which is based on experienced subject matter experts.

Lean principles support the optimisation of the processes with a focus on these three aspects: time to market, stakeholder management and domain knowledge. The main advantages of lean principles for automotive software projects is a fast visualisation of the executed processes and based on it, improve our efficiency.

3. Kanban in software engineering

The name “Kanban” originates from Japanese and could be translated as “signboard” or “billboard”. It is a flow-control mechanism for pull-driven “just-in-time” production. The idea behind Kanban is to execute Lean principles in practice.

David J. Anderson defined 5 Kanban core principles which to agreat extent overlap with Lean principles [5].

– Visualise the workflow – one has to understand the route covered by an item between a request and its completion.

- Limit WIP – limiting work-in-progress implies that a pull system is implemented on parts or on the whole workflow. New work is "pulled" into the new activity when there is available capacity within the local WIP limit.
- Manage Flow – the flow of work items through each state in the workflow should be monitored and reported.
- Make Process Policies Explicit – the process needs to be defined, published and socialised explicitly and concisely.
- Improve Collaboratively (using models & the scientific method) – the use of models allows a team to make a prediction about the effect of change (or intervention).

In software projects, using "Kanban" is becoming increasingly popular regardless of the project stages or production methods. An interesting aspect of this technique is that it is becoming an inside tool in both waterfall and agile processes.

Kniberg [6] points out that Kanban is less prescriptive than other agile methods like RUP, XP or even SCRUM.

Scrum, XP and RUP are highly adaptive while Kanban leaves almost everything open. The only constraints are *Visualize Your Workflow* and *Limit WIP*, which makes it a great tool for quick and efficient workflow and a process management tool. Especially, in the case when the prescribed rules and artefacts do not fit project needs. Scrum prescribes the use of timeboxed methods, but in the case of a support team or a firefighting team, it is hard to plan tasks in a sprint timebox.

3.1. Metrics – a way of observing facts and finding bottlenecks

To make decisions, the management of a given project requires capabilities for adequate situation analysis [7, 8]. This role is served by project metrics, correctly selected criteria according to which the defined parameters can be observed.

A simple visualisation is a great way of investigating the work of a team and the current state of progress. However, it is mainly employed in the day-to-day planning. When more accurate analysis based on a larger volume of data is needed, it is crucial for creating metrics or information-gathering schemes. Metrics are collections of updated and adequately represented data used for problem identification and decision-making.

The value of a good metric is to find a proper way to monitor bottlenecks in the whole processes or at its stages. For instance, one can measure the development processes in detail, i.e the devlopemnt team, integration, defect handling, system tests and network integration (target environment). One of the key findings is to focus on measuring the flow and not the constraints, which means it is better to identify and then remove organisational impediments instead of measuring it. Another interesting aspect is not focusing on speed measurements but on monitoring capacity. Speed is usually related to the human aspect and could foster unwanted competition instead of the collective responsibility for the project [9]. However, the case presented in this paper depends more on dynamics development analysis and not always on sustainable development of new features. The key concepts in measuring work efficiency in this specific case are as follows:

- Reaction time: it is the interval time between reporting an issue and starting an analysis of the problem. A reaction can be defined in a number of ways, however, according to the most common definition the reaction means delegating a task to a team.
- Lead time: a total time measured from task creation until its c ompletion. Lead time takes into consideration all of similar events between these two points, both predictable and unpredictable.
- Cycle time: the correct volume of work.

Figure 1 is used to portray these two concepts. It must be noted here that the entry and exit points for work units, as well as the in-between points, are defined in each project. The purpose of both of these metrics is to show the current work efficiency and the potential decrease in the time and costs of delivering a valuable work unit. More practical details and the corresponding results are described in Section 5.2.

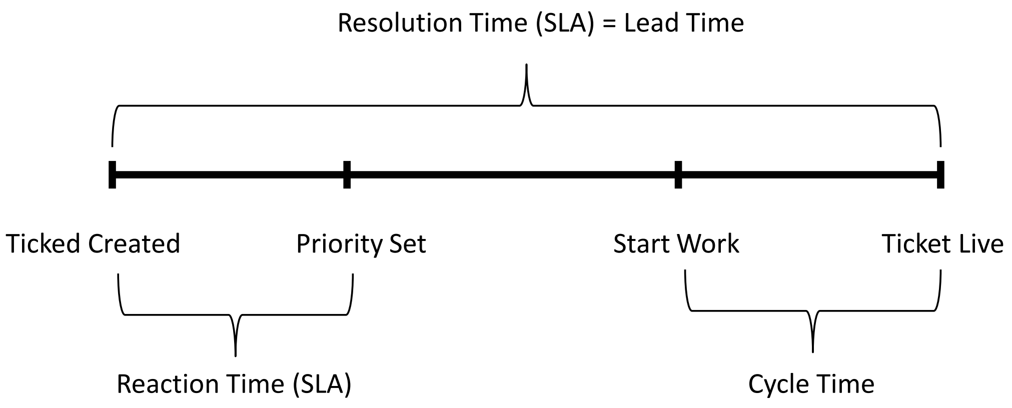


Figure 1. Lead time and cycle time

4. Related work

Mattias Jansson, Operations Engineer at Spotify[1], introduces [10] Kanban in the operations team as the answer to the growing number of different kinds of tasks. Before testing Kanban, the team noticed that although they were quite efficient, they were not able to plan far in advance. The problem was that they were reactive and not pro-active. The growing number of "urgent" jobs from other departments was always more important that the internal tasks and the context switching decreased the team's effectiveness. They realised that the company was growing too fast for them to accommodate.

Soon after Kanban's introduction, they noticed that their lead times became shorter, more internal tasks were done, and the departments they interfaced with were more satified.

In his book *Lean from the Trenches* [11] Kniberg described PUST – a digital investigation system for the Swedish national police authority. Due to the project scale, the teams, as well as the Kanban boards, were divided into subsystems. Besides having WIP limits in regular tasks, they also decided to restrict the number of bugs reported in the bug tracker. In the case of blocker priority, the bug had to be fixed immediately or if it was less important, it had to be replaced with an existing one from the top thirty. Otherwise, it would be ignored. He claimed that such an approach not only allowed for effective communication (lower number of bugs, highly prioritised bugs were immediately fixed), but also avoided continued change control meetings to manage long lists of bugs which would probably never be fixed.

Ikonen et al. [12] conducted a study in a medium-sized project (13 developers) in the R&D field. The investigation focused on the following project work aspects: *documentation, problem-solving, visualisation, understanding the whole, communication, embracing the method, feedback, approval process, selecting work assignment.* The presented results indicated considerable benefits of the Kanban technique including team motivation and control over project activities. Most of the work aspects were positively supported by Kanban techniques inside the team.

Middleton and Joyce in their BBC Worldwide case study [13] showed that as a result of the introduction of the Kanban technique, the lead time to deliver the software improved by over 37% and the number of defects reported by customers decreased by 24%. They observed very similar obstacles to those which may occur after the introduction of the lean principles connected with the environment and workspace, such as the tension within the existing corporate standards and processes. Some especially common obstacles encompass, e.g. office space designed inappropriately for Kanban boards, Kanban and reduction of WIP inability to work with milestones and Gant charts, close team co-operation with the customer may be seen as working beyond the

[1]Spotify is a music streaming service for desktops and smartphones, which aims to provide a wide-ranging music collection.

remit, and the self-managing team of specialist may be challenging to the managers.

They observed that the Agile approach, especially Scrum, has some similarities. However, they also noticed that the Kanban technique and Lean have several advantages over the Agile/Scrum approach. They claim that WIP limits the pull work model compared to the Scrum Push and timeboxed approach, it reduces delivery time and allows to develop better quality software. They also noticed that the ownership and responsibility of the Scrum "impediments list" are diffused. On the other hand, the Lean team must solve the problem immediately if they are blocked, because of limited WIP and visualisation on the Kanban boards. In this case, all staff members are obligated to eliminate the bottlenecks.

In another case study by Middleton et al. [14] the Timberland Company was analysed while practising Lean thinking for two years. They noticed many steps without any aded value in their processes. A survey amongst company staff showed that the majority of them supported lean ideas and thought they could be applied to software engineering. Interestingly only a small minority (10%) was not convinced of the benefits of lean software development. The company showed a 25% gain in productivity and time for defect fixing was reduced by 65–80%. The response on the product released using lean development from customer site was overall positive.

The authors of each study emphasise the benefits of of the introduction of Kanban and collaboration both in the team and with the client. The presented work was mainly conducted as internal projects. Moreover, most of the described projects were relatively small. Hence, they did not require the governance of the customer collaboration process. Additionally, they did not show how to evolve and build the lean values in the team and with the client to establish and use the Kanban technique effectively. Thus we want to present the Kanban technique in the extensive project setting and the way of collaboration with the external customer.

5. Discovering Kanban

This section presents two different approaches and two different perspectives of Kanban introduction. In Project A[2] Kanban was introduced as a tool for dealing with unplanned tasks in Sprint. In Project B the main goal was to unblock communication in the extensive stakeholders structure.

5.1. Project A

5.1.1. Background

The system under investigation covers all aspects of car purchasing in one of the premium car manufacturers in Germany. The system was designed for experts and is used internally by the customer. It allows to buy, lease or rent cars by the clients' employees, institutions or VIPs, car fleet management and used car dealers.

The system consists of two main components. One is a new version of the system developed as a modern web-based application. The second component written in COBOL is the old system's version, which is to be replaced by a new version step by step. Thus, both systems are available to the end users, and this, in turn, requires data synchronisation in real time.

The project uses the Scrum framework with certain small additional procedures, like the additional Scrum of Scrums meeting and the PO-Team meeting. A typical Sprint takes three weeks, some of the user stories (US) are approved during the sprint, some at the end during the demo. The majority of US are confirmed and tested by the PO-Team, however, in the case of larger epics, there are more people involved, including many external IT specialists.

5.1.2. Timeline

The old system version has been being developed since 1990 using the waterfall software development model. At the beginning the new version of the system was also developed using the waterfall software development model. The devel-

[2]Due to a commercial agreement, the project names have been anonymous.

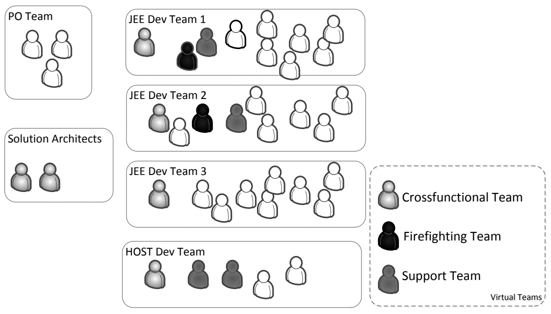


Figure 2. Project A – team structure

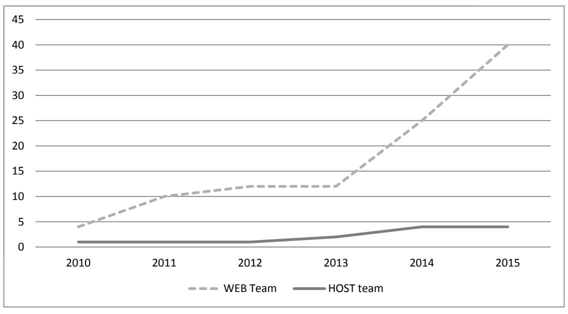


Figure 3. Project A – team grow

opment started in 2010. The first release after 16 months showed that it was not possible to integrate with the old system and that the minimal end user needs were not covered. In 2012 it was decided that to improve co-operation between both systems and ensure faster delivery, new requirements for the whole project will be developed as one Scrum project. After the final transition in 2013, as it was mentioned above, the entire team and the customer used Scrum. Currently, a new version is being released at least quarterly. In the case of urgent requirements, we will provide minor releases extending the latest production version.

It is important to note that the team started to expand rapidly in 2014 (Figure 3). Until mid-2013 no particular need of improvement had been noticed. Support and bug fixing were performed on a daily basis by one or two experienced developers. Also, issues were stored in several systems or delivered via email, hence developers responsible for support and bug fixing had a detailed overview of pending issues.

A dynamic increase in the number of team members forced a change of the project processes. A growing number of developers caused code integration problems. Instead of one Scrum Team the Scrum of Scrums concept was introduced and a different project governance model (3 Scrum Masters, Project Manager, PMO[3] support). The response time and cycle time became longer, mainly due to the insufficient expertise in the automotive industry among new developers and new Product Owners. Even though the developers mentioned above have been previously involved in different automotive ventures, domain knowledge is often one of the main obstacles in a software project. For example, the automotive domain consists of several specific sub-domains and a vast number of process details, – which is one of the main reasons for building custom software. In general, team members do not have all the required knowledge, and the project must acquire additional information before accomplishing productive work. The sources of this information can be relevant documentation, formal training sessions, meetings with domain experts and key users.

Additionally, several new business components started to be delivered, which, in turn, resulted in a growing number of support requests and end-user bugs.

5.1.3. Team composition

The Team consists of 45 people. Around one-third of them are connected with the project from the initial stage. Approximately half of the workers had several years of experience in enterprise projects. The entire team is divided into seven sub-teams (see Figure 2), two of them are virtual. A team member could be assigned to more than one team because of his/her function.

– *JEE Development Teams* (x3, DT) are responsible for the new system version, they use Scrum. Each team has about six members and the Scrum Master.
– *Host Team* is responsible for developing the old system in a Cobol technology. The team consists of 5 members and the Scrum Master.
– *Fire Fighting and Support Team* (FT and ST) consists of 6 members. In most cases, they are nominated in the sprint beginning from

[3]PMO stands for Project Management Office.

each development team. The team is responsible for the integration and production of bug fixing and for providing 3rd line support. The members of the team change every Sprint session.
- *Cross-Functional Team* (CT) consists of technical leaders from each development team and solution architects (SA). It focuses on long-term technical and business decisions and designs new components and supports customer.
- *Product Owners* (PO) *Team* consists of 3 business architects focused on the new User Story development. They work and agree on a new functionality directly with end users and major stakeholders within the organisation.

Up to a 70% of the capacity of people who lack industry experience are used in the initial few months of the Sprint. The slack time is used for internal project training provided by experienced software developers and architects.

Project teams are located in 3 different cities. This type of work is organised in accordance with the Distributed Scrum concept as described by Majchrzak et al. [15].

The development team and project management are located in two cities, the stakeholders and PO work in the third location. The main Scrum meetings were conducted in the customer's office. These meetings were attended by several team members only; the remaining ones participated in them via video calls.

Regarding daily contact with the PO team as well as the major stakeholders, these are managed mostly by means of email communication and video calls.

5.1.4. Engineering practices

From the very beginning, we were focused on XP techniques [16] which could be applied in both the waterfall and then in Scrum framework:
- test-driven development (unit tests);
- clean code [17] instead of code documentation;
- automated end-2-end (E2E) testing covering the user stories;
- continuous integration after each source code change, nightly build includes long-running E2E regression tests;
- source control software and rigorous configuration management;
- bug-tracking software (JIRA [18]);
- documentation in wiki (Confluence [19]).

This results in high test coverage. The team can provide a new release after each sprint. Due to E2E testing, business and technical complexity, the results are not always stable. About 5% of the tests fail regularly. The problems appearing before the release are checked manually to ensure that the reported ticket occurs because of new functionality or because of bugs. Every time a bug is found, it is promptly reported in the issue tracking system. Another aspect which needs to be included in agile projects is the branching strategy. Similar to Shabib et al. [20] we have found that a complex branching strategy could impact the quality of software. Despite the fact that the project consists of several teams, a decision is made to reduce the number of branches to the minimum, and to use the simplest branching strategy possible (Figure 4). The reason behind that was the need of frequent merging (even several times per day) as well as the choice to use one branch for the sprint, maintenance branch and release branch. This was only possible provided that the team decide to use rigorous code-change rules, such as frequent commit and integration (several times per day), and quality rules such as no failing JUnit tests (immediate fix or revert) static code analysis before commit, clean code and alike.

The branching and quality assurance rules, such as XP practices, clean code and fully automated E2E, were also valid for each team working in the Kanban process, hence the changes in Sprint and Kanban were visible immediately for every team member in the project.

5.1.5. Kanban introduction stages

The main impulses for employing Kanban were the doubts expressed by the team with regard to bug correction and new feature implementation, which had not been explicitly defined during the

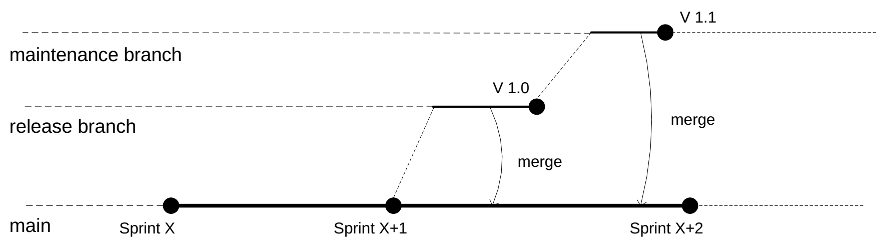


Figure 4. Project A – branching strategy

sprint planning. The change requests were often reported by end end users as tickets sent directly to the support team instead of PO. It is worth noting that the problem was not only which bug and in what order a given task or bug was to be corrected, but also the decision determining the incorrect workings of an application and the decision about who would be the sponsor of a given change.

The main stimulus to implement the improved process of error and support request management originated from the developers' team. The team initiated efforts to improve the already existing processes due to the growing amount of correspondence (delays and handoffs), new team members scattered across localities (delays), domain knowledge (waiting for solution approvals) and some restrictions stemming from the Scrum framework such as timeboxing and the sprint target (switching tasks and relearning – in the case of a new sprint).

Step 1: Identify work to be done. The first step aimed at systemizing the volume of work being done outside the sprint was to create one list of errors and problems from various sources and then organise them in the JIRA system. In the project, because of diverse users and conditioning, the above mentioned errors and queries could be called in using many ways, i.e. by email, by telephone but also with the help of HPQC and Peregrine. From the developer's perspective, many sources of those were impossible to accommodate and respond to, similarly to prioritising decisions.

The unified bug list was not the right solution as the following drawbacks were found very quickly:

1. The developer had to arrange who would be the sponsor of a change or error – fixing.
2. In the case of the lack of symptoms, many tasks had the In Progress status. It is worth noting that the developer was usually assigned to the FT team for approximately only two weeks. As a result, the said developer would begin many tasks (the work in progress was not defined or limited) and practically would not complete any in real life. Then the tasks would be returned to the "To Do list" and the whole procedure would be repeated. Consequently, some errors would wait for weeks before being solved or rejected.
3. All of the agreements and contacts with PO and stakeholders were chaotic. From the point of view of each user group their bugs or tasks had the top priority. Undoubtedly, it resulted in misunderstandings and also caused additional arduous communication by email.

Step 2: Identify workstreams. The next step was to identify the workstream and WIP arrangement (Figure 5). For instance, people working on subjects connected with support received their boards or were able to use the common one, but their tasks were marked with different colours. Similarly, people working on errors (FT) owned their boards. Very quickly another problem emerged, it was connected with the project characteristics. Some of the bugs had to be fixed using a different budget, which meant for example that only 1 FTE[4] could be assigned. Another problem was the fact that many errors were marked as a new feature (CR) and from the project standpoint, they were then investigated in a different budget and with the use of various resources. The constraints mentioned above required the introduction of additional Kanban boards. Then the question concerning the person who would make a choice between

[4]FTE – Full-time equivalent is a unit that indicates the workload of an employed person.

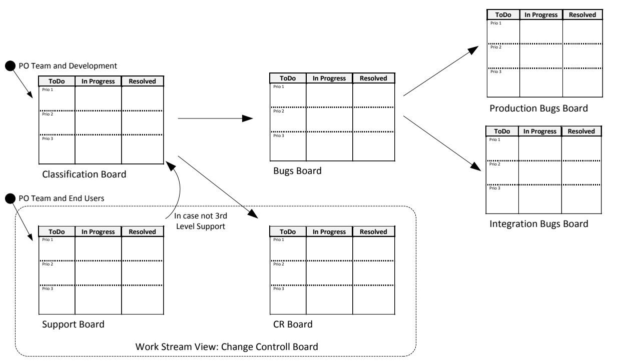


Figure 5. Project A – Kanban boards – work streams

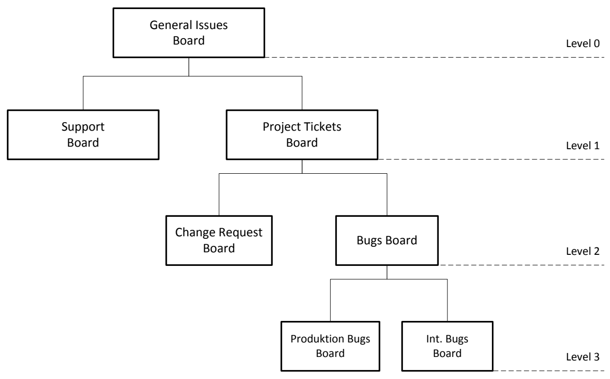


Figure 6. Project A – Kanban processes view

workstreams arose. Certainly, more boards were added and experienced developers or solution architects used them to investigate the issue and decide where it belonged.

Because of plenty of boards, this approach might seem very complicated. However, from the FT point of view, the "To Do Lists" were shortened and the focus was only on bug fixing.

Step 3: Improve the communication. The introduced division between work streams was optimal from the FT and ST point of view. On the other hand, from the management's and the client's perspectives (PO-Team), the existing work streams did not always meet their needs Because of this, to improve communication the efficient conduct of prioritisation meetings, we defined many options which grouped the chosen work streams but still retained simplicity. For instance in Figure 5 Change Control Board formed from Support Board and Change Request Board was identified.

5.1.6. Results

After nearly a year since the introduction of the above process, process, an interview similar to the one suggested by Ikonen et. all [12] was conducted. Opinions concerning the high complexity of the project structure and different expectations were collected from each of the teams. Since the main work stream was done in sprints, we have focused only on selected work aspects.

The hierarchy of the Kanban processes was built (Figure 6), it provides information for particular team members depending on their level. Each project member can find the right perspective and reduce the amount of information depending on their needs. For instance Level 3 suits the FT as they will concentrate solely on selected and initially analysed bugs. On the other hand Level 0 or Level 1 would meet the needs of the project management team in terms of the general project state and SLA.

Documentation. Dispersed exchange of information by email and arrangements during several meetings have been replaced by cohesive comments within the scope of a given task. They allowed us to understand each given problem and the process in which a decision was made. To a developer, it become evident what and why needs to be done. A member of the ST additionally states:

> "Documentation has been improved, there is no worry about losing parts of data. Once something has appeared on a Kanban board, it will not be forgotten or omitted, and it will be equipped with the correct commentary serving afterwards as a source of knowledge in similar problems." [ST]

The error log and new feature request stored in one place allow for a significantly more straightforward analysis of the changelog. It consequently helps to understand who made a decision to introduce a change, and what were their reasons as well as motivations for such an alteration.

> "Due to the significant number of tasks regarding various components and business processes, we have made a decision for all the information to be included in the comments of a given bug or task. This simplifies the

> correlation detection between tasks and bugs and the root cause analysis." [CT]
>
> "Bugs are well commented on, hence we can reuse historical information during later stages." [SM]

From the developer team's point of view, the key aspect is the task status and in particular the information which may impact the current tasks in the sprint.

> "We are aware of errors and how they are managed; earlier on we used to lose such data, whereas at present we can obtain statuses of errors which are of interest to us." [DT]

Problem solving. The introduction of Kanban allowed for easy task assignment to suitable people in the correct order. In the case when a developer or a customer finds a bug or requests a new feature, he or she can easily issue a new ticket without the need for consultation with the Scrum work model and budget, which made it possible to continue work without delays:

> "It facilitates work and provides a structure error correction." [FT]
>
> "Developers are not blocked and know that the reported problem will be properly classified and solved. They are not blocked by unplanned tasks and can develop new user stories without changing the context." [CT]

Even though most of the team thinks that certain progress has been made, ST and CT still see some room for improvement:

> "Using Kanban has not solved all of our problems. Too much of a mess occasionally still happens." [ST]
>
> "Still, a large number of tasks, e.g. copying emails and HPQC Bugs into JIRA require additional time and should be automated." [ST]

Also, one of the FT members pointed out that there are still problems due to a largely rotating team:

> "On the other hand, we have to improve the process of bug fixing itself. A task once started does not always get completed before it is time for a developer to leave the FT team." [FT]

Visualization. The process of error fixing and CR management is much simpler and more transparent from the team's point of view. Bug statuses and workloads are always visible. Each member can easily select a given board and then the related task:

> "Kanban boards look much better and provide more information than a long bug list." [SM]

The Team and stakeholders understand what the support and bug fixing process looks like:

> "Once upon a time, I found it difficult to describe the way in which we worked. A project seems to be much more mature, once it it becomes clear how a given process functions." [ST]
>
> "Above all priorities are error statuses and developers who make such errors. Such information is indeed favourable in case of a significant number of various errors." [FT]

Visualisation also helps people working on regular Sprint tasks, and they claim that:

> "It is easy to observe whether a person is working to solve a problem which has blocked us and what is its priority." [DT]
>
> "When we need to have a general overview, Kanban boards offer a great deal of assistance." [CT]

Communication. Internal communication between teams and the PO has improved dramatically. Instead of having dedicated meetings to discuss each critical error, regular meetings for the selected work streams have been initiated:

> "The number of meetings has grown, however, they have become shorter and allowed us to manage the tasks on given boards effectively." [CT]
>
> "The number of meetings has increased, which could be considered a major improvement. Instead the inefficient information flow via emails, it is much faster to conduct a meeting drawing on the Kanban board and thanks to this keeping the changes up-to-date." [SM]

From the team's point of view, the information concerning prioritising error-correction and the details concerning them have been set in place. On the other hand, from the PO's point of view, communication with the development team has been improved:

"The client knows who to speak to with regard to a given task and when to expect the solution to a particular error." [FT]

Another equally important aspect is the option of regular progress monitoring.

"We can see the status of work immediately. I don't have to interrupt people and ask what they are doing." [SM]

A different point of view is presented by developers working on tasks in Sprints. As far as they are concerned, communication has been reduced to the minimum.

"Not relevant in the case of a development team. We do not have to cope with bugs because we know that the right people from the Firefighting Team will support the process, and we do not have to be involved." [DT]

Approval process. The basic advantage of the defined process was the improvement of work stream choices for new tasks:

"In the case of fixed bugs, our process still needs improving. Despite the fact that developers know that they should fix and test the bug, we additionally need a *Verify* column in order to explicitly emphasise the need for a test and verification." [DT]

"It is easier to approve bugs and to assign them to the proper work stream." [SA]

Additionally, the introduction of rules concerning the flow and identification of the sponsors responsible for error-correcting has improved support work:

"In the case when a task cannot be easily solved, because we have to deal with a non-trivial bug or simply with a new feature, we can easily move it to another board." [ST]

"The PO Team checks the Kanban boards, provides some additional information and, most importantly, sets the right priority." [FT]

Despite the above improvements, the team has still seen the need for enhancing the process:

"Unfortunately, as of now, we have not been able to establish a correctly-functioning approval process. We need another state (column) – Verified. Fortunately we have a proper release process, we can see on the Kanban board what got released and what didn't." [SM]

Selecting work assignment. Through close interaction with the Customer and PO, we have been able to set our priorities right, which in turn has allowed for optimal task accomplishment by the team:

"You simply take the first task from the first column. You don't have to search for tasks or ask others." [SM]

"Together with the PO-Team, we conduct the prioritisation, thus the most important tasks are at the top of the To Do column." [SA]

Taking into account the aspect of a budget for particular error types, the choice of a task is clearly sensible from the Team member's point of view:

"Different boards help us find bugs from different work streams." [FT]

"Tasks are split into work streams and could be easily selected based on priority order." [DT]

"We use issue priority in order to select work assignments. If the tasks have the same priority, we simply select the oldest ones." [ST]

5.1.7. General problems and future work

One of the most difficult challenges found in the processes described earlier is the need to perform numerous activities manually. Clearly, using extra human resources, e.g. in order to copy emails to Jira, would be considered a waste in the process. The next step should be the introduction of such systems as Jira Service Desk [21], and e.g. ConnectAll [22] to integrate HPQC and Jira.

It was also observed that assigning new people to a virtual team at the beginning of each Sprint may result in wasting time and and resources. Certain tasks sometimes require several days to analyse, fix and test. If the members of virtual teams are changed while tasks are still in progress, new developers have to start them almost from the beginning. Thus, it was decided to allow people to work on a given task, even though they are assigned only to Sprint development.

5.2. Project B

The project involves the manufacturing of production software in a large automotive concern. A part of this software supports the direct steering of car production in three separate stages: body construction, paint shop and assembly. The steering systems are critical because each potential software error generates relatively expensive problems. The said software is employed in several dozen factories belonging to the aforementioned automotive concern. The goal of this project is the delivery of services within a specified time frame and with specified availability, namely the development of new functions and the support of current and existing functions in the production environment.The support is limited to the most difficult problems requiring changes in the software or specific changes in the system configuration. In Project B, the main challenge in the introduction of the lean philosophy with Kanban techniques was combining transparency principles and contractual issues. Many contractual constraints originate from the extensive structure of stakeholders on the customer's side.

In general, it is possible to visualise many processes on a Kanban board, e.g. governance, transition, staffing, knowledge management, technology and infrastructure, financial and contractual elements.

5.2.1. Extensive structure of stakeholders on customer side

In this case, the customer is one of the biggest world manufacturing consortiums with many layers of interests. On the one hand, there is a need for simplicity, however, however, on the hand the goals is to deliver the production of software – a crucial part of the customer's business. To meet these contradictory expectations a set of stakeholders was identified, however, this article focuss on the following groups:

– The IT department which is the main stakeholder from the contractual point of view.
– Factories which are most important in case of the continuity of the project.
– Quality assurance which is most important to evidence our quality.

5.2.2. Team composition

Due to the massive system function complexity, the team was extended. Taking into account various functions and tasks, the project was divided into the following teams (see Figure 7):

– *Feature Team* (x4) – these are people directly responsible for software manufacturing. The team consists mainly of Programmers and Testers. Each Team is responsible for specific business components.
– *Project Support* (*cross-functional team 1*) – responsible for the infrastructure and continuity of project functioning in relation to technical data, namely integrating both Client's and contractor's networks as well as supporting the build and configuration of management processes.
– *Governance* (*cross-functional team 2*) – responsible for the management and client co-operation takes key decisions concerning the project. It is involved in all the existing aspects of project management including change and risk management.

5.2.3. Kanban introduction stages

The deployment of the agile approach is much more challenging within the realms of a large organisation and an extensive stakeholder structure. The above project description does not focus on organisational or business limitations, it focuses on the employment of the Kanban techniques instead. Because of critical and limited functionality, all of the process changes had to be introduced carefully, i.e. with risk management, which is an indispensable element of the empirical project approach.

Step 1: Establishment of the common workflow. At the initial stages, the arrangement meant that each of the Teams functioned according to their rules and used their individual workflow. The following issues caused difficulties pertaining to the correct definition of the general state of work: defining a completed task (Defini-

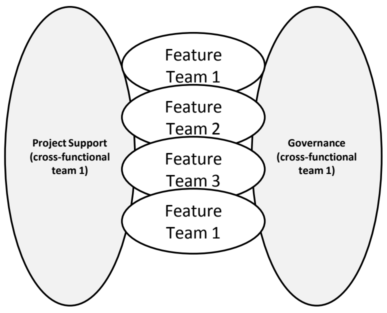

Project Support (cross-functional team 1)
Feature Team 1
Feature Team 2
Feature Team 3
Feature Team 1
Governance (cross-functional team 1)

Figure 7. Project B – team structure

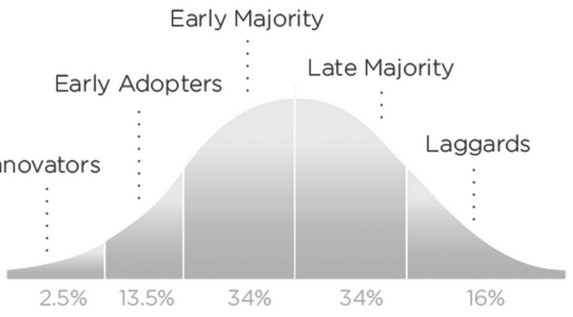


Figure 8. Innovation adoption lifecycle

tion of Done), reporting on critical productive errors (escalations) and seeing the fully complete picture of work in the entire project. After standardising the workflow, it was possible to create the root for visualising the Kanban board. In its initial stages, it comprises everyday work (daily business), meaning current tasks. It consists of the following stages:

– T-Shirt sizing: an initial assessment of a task, which is a relative description of the size of a task resulting from its complexity, uncertainty and repeatability [23, Chapter 7, 16]. At this stage, the estimates are not precise, and the analysis itself should not exceed 4 hours. The t-shirt sizing technique is similar to Planning Poker [24]. However instead of using the Fibonacci sequence, t-shirt sizes are used (XS, S, M, L, XL).
– Problem analysis: at this stage, a detailed analysis is conducted based on the earlier estimate. The purpose of this stage is the definition of the scope of work and its costs.
– Development: at this stage the earlier analysis is used to perform the task. The purpose of this stage is the engineering of a registrable change in software.
– Deployment: the final stage is the employment of software change and in the majority of cases this is the most complex process. The goal is the delivery of the change in the production environment.

It is possible that a problem is solved at each of these levels, which then completes the process.

Step 2: Visualisation processes. Visualisation is the best way to achieve a common understanding of the state of the project, the best way to keep a shared vision. It is possible to find the bottlenecks only when everything is measured and visualised to the whole team.

The reality of communication is that every stakeholder can have different interests. At this phase of introducing Kanban, it became crucial to start collaborating in the same "language". A Kanban board was created on the basis of the earlier study of the said workflow (see Figure 9). The workflow of problem management is described in Paragraph 5.2.3. Visualisation is not only communication improvement, but it is also a major factor in achieving the shared vision and promoting it in the whole project. After the introduction of the visualisations, the following observations were made in the teams:

– the processes were described and changes were continuously supplemented;
– the board was continuously adapted;
– the processes were always visible to all members of the team, and they were proposing improvements (feedback loop).

Step 3: Introducing the culture of self-improvement. The project approaches based on nimble philosophy are tough to implement for multiple reasons, such as the requirements for experience and courage. A given situation can be much simpler if there is an environment open to the Agile and Lean thinking. It is fair to say that their deployment is not possible without the culture of change and constant improvement in place.

In the process of change within a large organisation, one must not forget about sociological processes, an example of which can be the Adoption Curve (see Figure 8) [25]. It is precisely this model that became used in the process of employing change to the project and its close

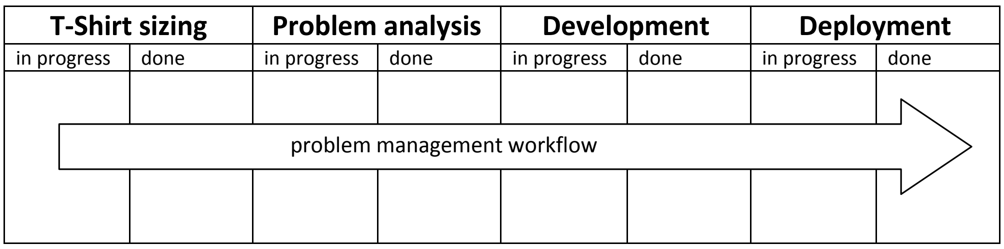


Figure 9. Project B – Kanban board – problem management workflow

environment. The technique used in the project was, among others, the selection of the "so-called" Change Ambassadors (early adopters), who were recruited from the management of selected feature teams. It was this group to be the main communication target in relation to Kanban deployment techniques. In the aforementioned "Early Majority" means the Team.

Instilling the Lean culture allows the use of techniques such as Kanban. Simultaneously, an organisation promotes an adaptive approach on a wider scale, moving far beyond the scope of this project.

Step 4: Managing improvement from the team. The Coach is a crucial role in this operation, and their position is not to be underestimated. However, their role is to guide the Team towards learning the process of coming to correct conclusions. Just as a parent bringing up a child teaches it to walk and then allows it to reach full independence, so does the Team Coach by pointing out specific problems and then teaching the Team members a lesson on independence.

The first dilemma, observed thanks to visualisation and the common workflow, was the lack of comprehensible understanding of the Definition of Done (DoD). At the beginning, each Team defined their DoD in their own way. Unsurprisingly, it invariably led to serious misunderstandings during the execution of the said agreement, especially at the final stages of the project.

Second of all, certain knowledge limitations became apparent within the Team. The Kanban board immediately made the team painfully aware which module lacked the necessary knowledge, where fewer tasks existed and where there was a potential for certain key moves. Through the act of standardising the workflow and programming it correctly in the JIRA, both the executive documentation procedure and the communication regarding production difficulties were successfully improved:

– documentation concerning current problems consists of the necessary meta information, i.e. contact persons, references to other existing documents (i.e. change request);
– summary of existing problems is documented in a uniform manner.

Step 5: Introducing the processes to the Customer. In the described here case, being able to implement the process of improvements was a direct result of the steps taken at the previous stages. One of the most efficient ways to achieve lean principles is visualising a given processes. Together with identified stakeholders (see Section 5.2.1) we decided to start with three working areas: problem management, governance processes and release management

Problem management board. This visualisation shows the whole scope and the parameters of the daily state of work. The content of the board consists of a set of tickets (problems) which were sent to the development team. The goal of the problem management board is to simplify the feedback loop with the factories – one of the crucial stakeholders identified for the project.

The problems (tickets) are prioritised and delegated to the appropriate team member. They can proceed with a particular case of the Kanban board relatively fast, aligning work to their processes and also completing the gaps in the specification.

Governance workflow. The work with the governance processes was dynamic, which was possible thanks to a frame contract joining two companies by agreements that set out the terms and conditions for delivery services. In this project the goal was the delivery of the 3rd level development support. The frame contract allows to adjust the financial part of the delivery – each service can be negotiated separately. For instance, the workflow of offering (see Figure 10) consists of following steps:

1. *Service* request: the customer requests a specific offer.
2. *Capacity*: project management checks the capacity of the team, inclusive of the know-how in other projects (if needed).
3. *Offer*: a full offer is made to the customer.
4. *Confirmation*: the customer accepts or rejects the offer.

The real workflow is more complex than described here. However, this example shows how the crucial part of the processes can also be involved in the Kanban visualisation. The project management team and the customer's IT department work jointly on the governance board. As a result, a faster "one-piece flow" is achieved. It is the crucial part of Lean Manufacturing [26] and also works well with software development.

Release management processes. Besides ensuring the quality of the software solution, it is necessary to deliver software packages to the factories. The development team delivers various types of ensembles: release, service pack, fix pack and hotfix. The roll-out team in the factory installs the corresponding package and ensures the continuity of production. The development team supports packages in case of emergency. Quality assurance is the most important stakeholder in this area. With the visualisation of the release management processes, the delivery can be prioritised more easily and additionally, the steps of the processes can be adapted relatively fast. The workflow of release management (see Figure 11) consists of the following steps:

1. *Development*: preparation of delivery.
2. *Ready for tests*: finishing delivery and releasing it for the customer.
3. *Tests*: the customer conducts acceptance tests.
4. *Ready for roll-out*: finishing the delivery and releasing it for roll-out (installation).

5.2.4. Results

One of the most significant consequences of the introduction to Kanban is the ability to measure a process, for example by quoting such defined metrics as:

– an increase in the number of created tickets relative to the closed cases (see Figure 12);
– a possibility to measure the average time for closure and the costs of fabrication (Lead Time and Cycle Time);
– cyclical report of shifts in the original estimate, which meant a comparison of adequate work estimation in the T-Shirt sizing phase to the actual volume of work being done. The report made the early detection of the most incorrect estimates and their causes possible;
– the quality of task documentation coming from the Client is also measured, which in turn allowed for the introduction of multiple improvements. The final effect was improved communication with respective Client departments.

Additionally, SLA values (Service Level Agreement) are measured within the scope of the said project based on the previously agreed parameters. The achieved SLA may shift up to a certain extent. The following SLA indicators are used in the project:

– Reaction time from the moment a work unit was created to the beginning of actual work (early analysis),
– Time of the initial analysis (T-Shirt sizing).

In the analysis phase (Problem Analysis, Development and Deployment), the high level of vagueness made it impossible to introduce the SLA which would measure the end of work. An sample cumulative chart (Figure 13) highlighted the piling up of tickets in the analysis phase for the final period between September to November. Such visualisation enhanced the credibility of the reports for the Client. The goal of metrics is to monitor the state of the project and react when problems occur. In this case most valuable metrics are: Reaction Time, Lead Time and Cycle Time. In practical terms, the results of the metrics are not always easy to understand, which was also the experinece gained in this project. Matters which need to be interpreted separately are described below.

– Incompletion of the data – there was sometimes a gap between real processes and the

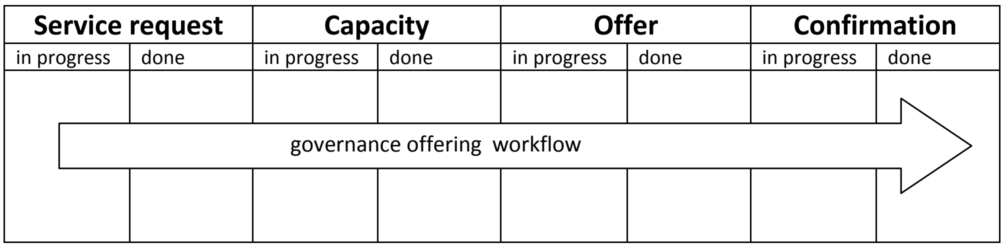


Figure 10. Project B – Kanban board – governance workflow

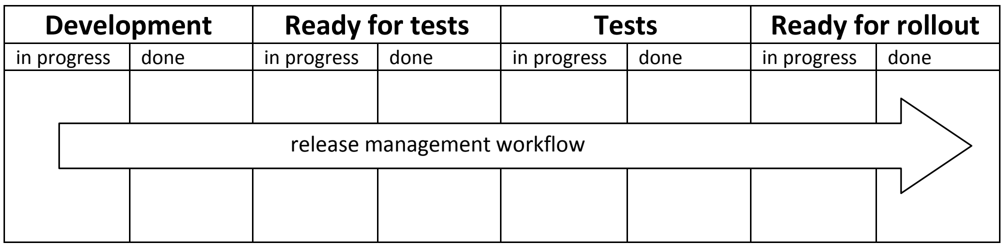


Figure 11. Project B – Kanban board – release management

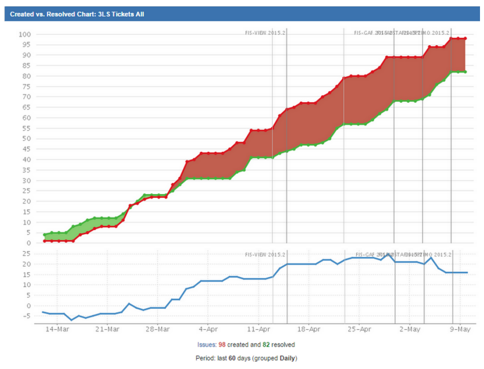


Figure 12. Created vs. resolved chart

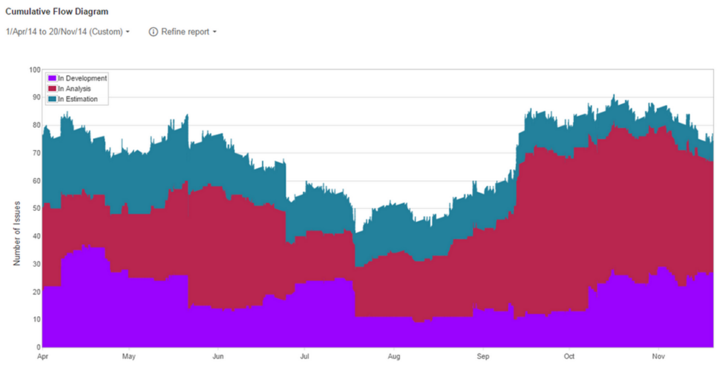


Figure 13. Cumulative flow diagram

documented data; for example, a request which was not registered in the system. It was assumed that these data were not crucial for this metric.

– The learning curve while introducing changes to the processes – the whole process of introducing lean changes with Kanban techniques is relatively time-consuming. Establishing the efficient workflow of tickets lasted more than six months, and the following six months were required to teach the entire team. It was assumed that there always was a learning curve and the measured data were calibrated with time. Hence, it effectively corresponded with the reality.

– Team rotation – the real problem was when the capacity of the team varied. All projects need to deal with such issues not only because of the growing numbers in teams on the team, but also because of technical experience and domain knowledge, which should be taken into account. Unfortunately, the impact of this issue on the presented results cannot be accurately compared.

– Bad multitasking – the more tasks there are in the progress, the less efficient the working time is. The Kanban visualisation allows us to minimise this problem. However, this aspect needs considering when interpreting the final results.

– The complexity of business knowledge – it is a well-known fact that the software for production systems is complicated because of various elements, such as interface systems, real-time constraints, security aspects and production continuity. There is no direct relation between the level of business complexity and the quality of the project; yet in metrics, it was observed that in the tickets, comparison to the others, this adverse effect can be eliminated by using medians instead of averages.

Reaction time. The total time from reporting a problem to the moment the development team can start dealing with it. Table 1 and Figure 14 show the effects of the described aspect of the described aspect “the learning curve during the introduction of changes to the processes”. The year 2014 was the learning phase. From 2015 Q1 tuning the reaction time through minimizing “bad multitasking” was started.

Lead time and cycle time. The lead time is the total time required to develop a solution to the problem including corresponding activities, both the predicted as well as the unpredicted ones. It is the time from task creation until its completion. Cycle time is the correct volume of work.

In both tables (Table 2, 3) one can observe the difference between averages and medians. The cause of the difference is that a small amount of tickets was extremely complex. It corresponds to the described issue “complexity of business knowledge”. In Figure 15 one can observe how the cost of the delivery was optimised. The period between 2013 Q4 and 2014 Q2 was the time when measuring data was not complete. From the 2014 Q3 team rotation begins. In the first quarter of 2015, we finished installing all modules of the software.

6. Summary and key observations

In a progressively larger number of IT projects, one can easily notice a trend towards process and tools optimisation. The software companies and its customers have spotted flaws in the current perspective based on waterfall approaches. There is a big potential in creating waste, e.g. through administrative behaviour. Moreover, frequently chosen software development methodologies do not encompass certain much-needed processes.

Although the Kanban technique is not the subject of many analyses and was not promoted as much as the Scrum or XP ones, it is more and more frequently used in software projects as one of the tools of Lean thinking. It can be used with positive results in each project type regardless if it is an “Agile” or “Waterfall” style operation.

It should be noted that this basic and simultaneously intuitive mechanism is a powerful tool allowing for the easy optimisation of nearly every activity and process within software projects. In both cases, substantial profits were observed both on the side of our Team as well as on the Client’s side over a relatively short period.

As mentioned above, the most important aspects of those undoubtedly are visualisations, process regular order and the creation of a cooperative platform, which can be easily modified and adapted to any given target group.

During the analysis of the consequences of Kanban deployment another perspective emerged. Taking into account the human factor, it can be observed that Kanban uses triggers as a tool for gradual self-improvement of each team, a sort of evolutionary step towards the reform of documentation and fabricating processes. Hence, unlike something forced upon the staff by the management or outside specialists, Kanban results in an all-natural, symbiotic and adaptive process.

7. About Capgemini and Software Solutions Center Wroclaw

With 180,000 people in over 40 countries, Capgemini is one of the world’s foremost providers of consulting, technology and outsourcing services. The Group reported 2014 global revenues of EUR 10.573 billion. Together with its clients, Capgemini creates and delivers business, technology and digital solutions that fit their needs, enabling them to achieve innovation and competitiveness. A deeply multicultural organisation, Capgemini has developed its own way of working, the Collaborative Business

Table 1. Project B – reaction time

Period		Reaction Time		
year	**quarter**	**average**	**median**	**issues**
2013	Q4	6d 8h	1h 21m	73
2014	Q1	2w 5d 6h	3d 3h	156
2014	Q2	1w 4d 23h	19h 14m	195
2014	Q3	3w 14h 21m	16h 52m	200
2014	Q4	2w 8h 38m	20h 42m	171
2015	Q1	3w 7h 41m	6h 31m	176
2015	Q2	1d 15h	5h 30m	161
2015	Q3	2d 6h	4h 31m	179
2015	Q4	1d 15h	3h 2m	223

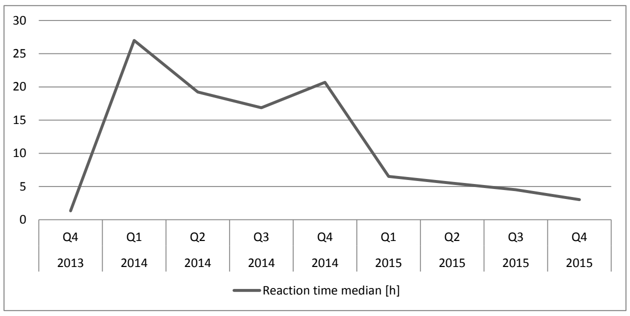


Figure 14. Project B – median of reaction time

Table 2. Project B – lead time (time line)

Period		Reaction Time		
year	**quarter**	**average**	**median**	**issues**
2013	Q4	6w 5d 20h	7w 3h 26m	25
2014	Q1	4w 3d 19h	2w 1d 10h	76
2014	Q2	8w 4d 4h	6w 2d 2h	97
2014	Q3	9w 6d 5h	5w 6d 4h	166
2014	Q4	16w 5d 22h	11w 6d 4h	173
2015	Q1	22w 1d 7h	15w 1d 19h	231
2015	Q2	15w 5d 16h	10w 6d 22h	148
2015	Q3	17w 5d 8h	10w 3d 6h	205
2015	Q4	16w 3d 5h	6w 1d 6h	427

Table 3. Project B – cycle time (in progress)

Period		Reaction Time		
year	**quarter**	**average**	**median**	**issues**
2013	Q4	3w 1d	2w 3d 1h	41
2014	Q1	3w 5d 5h	2w 3d 22h	62
2014	Q2	4w 6d 3h	3w 1d 5h	107
2014	Q3	4w 6d 12h	3w 5h 44m	130
2014	Q4	9w 21h 6m	5w 3d 2h	177
2015	Q1	7w 6d 14h	5w 10h 20m	200
2015	Q2	8w 1d 16h	3w 3d 20h	171
2015	Q3	6w 6d 8h	2w 6d 15h	191
2015	Q4	5w 1d 21h	1w 2d 23h	328

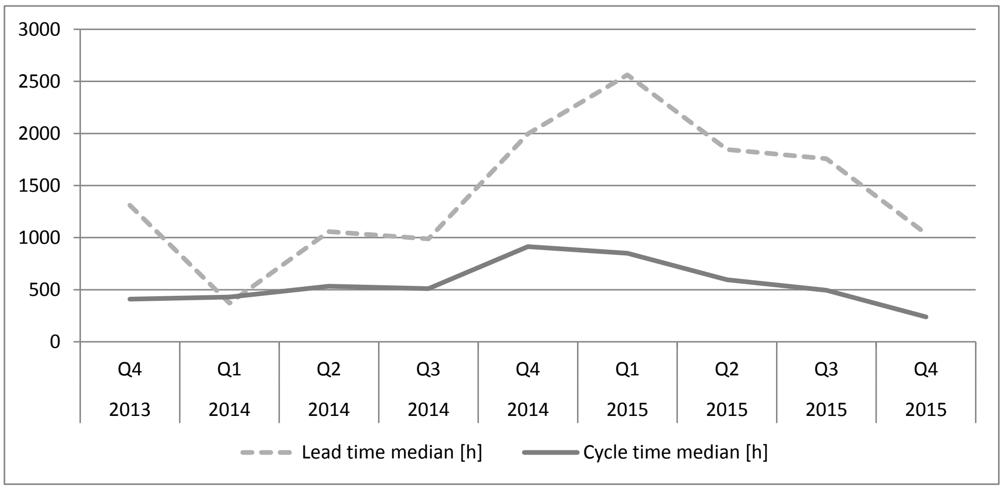


Figure 15. Project B – median of lead time and cycle time

Experience^{TM}, and draws on Rightshore®, its worldwide delivery model. Capgemini in Poland employs 6500 consultants and IT as well as business process experts. Centres for IT and business process outsourcing services has operated in Wrocław, Poznań, Kraków, Katowice and Opole with the main office serving the Polish market based in Warszawa. Capgemini Software Solutions Center exists in Wroclaw since 2004. More than 800 IT experts currently work in Wroclaw delivering high-quality services in the areas of software development, software package implementation and application lifecycle services to German-speaking clients.

References

[1] T. Ohno, *Toyota Production System: Beyond Large-Scale Production.* Cambridge, MA: Productivity, 1988.

[2] J. Koplin, S. Seuring, and M. Mesterharm, "Incorporating sustainability into supply management in the automotive industry – the case of the

Volkswagen AG," *Journal of Cleaner Production*, Vol. 15, No. 11, 2007, pp. 1053–1062.

[3] J.P. Womack and D.T. Jones, "From lean production to the lean enterprise," *Harvard Business Review*, Apr. 1994.

[4] M. Poppendieck and T. Poppendieck, *Lean Software Development: An Agile Toolkit.* Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 2003.

[5] D.J. Anderson, *Kanban: Successful Evolutionary Change for Your Technology Business.* Blue Hole Press, Apr. 2010.

[6] H. Kniberg, *Kanban and Scrum – Making the Most of Both.* Lulu.com, 2010.

[7] K. Petersen and C. Wohlin, "Measuring the flow in lean software development," *Software: Practice and Experience*, Vol. 41, No. 9, 2011, pp. 975–996.

[8] M. Host, B. Regnell, J.N. och Dag, J. Nedstam, and C. Nyberg, "Exploring bottlenecks in market-driven requirements management processes with discrete event simulation," *Journal of Systems and Software*, Vol. 59, No. 3, 2001, pp. 323–332.

[9] M. Staron and W. Meding, "Monitoring bottlenecks in agile and lean software development projects," *Product-Focused Software Process Improvement*, 2011, pp. 3–16.

[10] M.J. Michael Prokop, "Use of kanban in the operations team at spotify," *InfoQ*, Sep. 2010.

[11] H. Kniberg, *Lean from the Trenches: Managing Large-Scale Projects with Kanban.* Pragmatic Bookshelf, 2011.

[12] M. Ikonen, E. Pirinen, F. Fagerholm, P. Kettunen, and P. Abrahamsson, "On the impact of Kanban on software project work: An empirical case study investigation," in *16th IEEE International Conference on Engineering of Complex Computer Systems (ICECCS).* IEEE, 2011, pp. 305–314.

[13] P. Middleton and D. Joyce, "Lean software management: BBC worldwide case study," *IEEE Transactions on Engineering Management*, Vol. 59, No. 1, 2012, pp. 20–32.

[14] P. Middleton, A. Flaxel, and A. Cookson, "Lean software management case study: Timberline Inc." in *Extreme Programming and Agile Processes in Software Engineering.* Berlin, Heidelberg: Springer, 2005, pp. 1–9.

[15] M. Majchrzak, Ł. Stilger, and M. Matczak, "Working with agile in a distributed environment," in *Software Engineering from Research and Practice Perspective*, L. Madeyski and M. Ochodek, Eds. Polish Information Processing Society Scientific Council, 2014, pp. 41–54.

[16] K. Beck and C. Andres, *Extreme Programming Explained: Embrace Change*, 2nd ed. Addison-Wesley Professional, 2004.

[17] M. Robert C., *Clean Code: A Handbook of Agile Software Craftsmanship*, 1st ed. Upper Saddle River, NJ, USA: Prentice Hall PTR, 2008.

[18] Atlassian, JIRA documentation, (2016). [Online]. https://confluence.atlassian.com/display/JIRA/JIRA+Documentation

[19] Atlassian, Specification – confluence advanced editor, (2016). [Online]. http://confluence.atlassian.com/display/DOC/Specification+-+Confluence+Advanced+Editor

[20] E. Shihab, C. Bird, and T. Zimmermann, "The effect of branching strategies on software quality," in *2012 ACM-IEEE International Symposium on Empirical Software Engineering and Measurement, ESEM*, 2012, pp. 301–310. [Online]. http://doi.acm.org/10.1145/2372251.2372305

[21] Atlassian, JIRA service desk documentation, (2016). [Online]. https://confluence.atlassian.com/servicedeskserver030/

[22] Go2Group, Connect all, (2016). [Online]. http://www.go2group.com/connectall/

[23] K.S. Rubin, *Essential Scrum: A Practical Guide to the Most Popular Agile Process*, 1st ed. Addison-Wesley Professional, 2012.

[24] M. Cohn, *Agile Estimating and Planning.* Upper Saddle River, NJ, USA: Prentice Hall, 2005.

[25] E.M. Rogers, *Diffusion of Innovations*, 5th ed. Simon and Schuster, 2003.

[26] K.L. Jeffrey, *The Toyota Way: 14 Management Principles From The World's Greatest Manufacturer.* McGraw-Hill, 2004.

e-Informatica Software Engineering Journal, Volume 11, Issue 1, 2017, pages: 59–76, DOI 10.5277/e-Inf170103

Systematic Literature Review on Search Based Mutation Testing

Nishtha Jatana*, Bharti Suri**, Shweta Rani**

**Research Scholar, USICT and Assistant Professor, Department of Computer Science and Engineering, MSIT, New Delhi, India*

***USICT, GGS Indraprastha University, New Delhi, India*

nishtha.jatana@gmail.com, bhartisuri@gmail.com, shweta2610@gmail.com

Abstract

Search based techniques have been widely applied in the domain of software testing. This Systematic Literature Review aims to present the research carried out in the field of search based approaches applied particularly to mutation testing. During the course of literature review, renowned databases were searched for the relevant publications in the field to include relevant studies up to the year 2014. Few studies for the year 2015–2016, gathered by performing snowball search, have also been included. For reviewing the literature in the field, 43 studies were evaluated, out of which 18 studies were thoroughly studied and analysed. The result of this SLR shows that search based techniques were applied to mutation testing primarily for two purposes, either for mutant optimisation or for test case optimisation. The future directions of this SLR suggests the application of search based techniques for other issues related to mutation testing, like, solution to equivalents mutants, generation of non-trivial mutants, multi-objective test data generation and non-functional testing.

Keywords: software testing, analysis and verification, systematic reviews and mapping studies

1. Introduction

Software testing [1] is a rigorous activity which must be done to find the errors, quality assessment and to gain insight into the state of the system. The key challenge in the process of software testing is to reduce costs and maximize benefits. Since it is extremely time-consuming and requires a lot of effort; the overall testing process needs to be optimized for testing practices. Software testing involves test planning, design, execution and evaluation, reporting and closure activities. Test design is an activity that entails the major chunk of software test activities. It includes reviewing the test basis, identifying test conditions, designing tests, evaluating them and thereby designing the test environment setup. One of the critical tasks in testing is the generation of test data. Research on test case generation has become quite prevalent in the last two decades [2–5].

Search based approaches have been applied to several optimisation problems [6]. Software test design portrayed as a well formed optimisation problem has been solved using meta-heuristic techniques [7]. The generation of test data can be automated using meta-heuristics or search based techniques using a specific fitness function to guide the search towards a potentially good solution within a search space [8].

Search based mutation testing (SBMT) works by formulating the test data generation/optimization and mutant optimization problems as search problems and applying meta-heuristics techniques to solve them. Bottaci [9] introduced the fitness function to apply optimization algorithms to kill the mutants or faulty programs. This fitness function was based on three conditions: reachability, sufficiency and necessity. These three conditions served as a base for the foundation of SBMT and are are still used for this purpose [10]. Optimization techniques, such as Genetic Algorithm (GA), Hill Climbing (HC), Ant Colony Optimization (ACO), Bacteriological Algorithm (BA) and Immune Inspired Algorithm (IIA) were used together with mutation testing by researchers as an art of SBMT [11].

The aim of this systematic review is to address the search based approaches applied to mutation testing either for test case generation or for mutant optimization. The systematic literature review (SLR) is the foremost essential step in the research process conducted to rigorously collect, analyse and report the current literature in the field. An extensive survey by Jia and Harman [12] comprehensively addressed developments of mutation testing. [13] lists the various advancements in the areas of SBMT since the year 2009. A recent systematic mapping by Souza et al. [11] mentioned related work in the context of test data generation for mutation testing, whereas whereas this SLR focuses particularly on the search based techniques smeared with mutation testing for test data generation/-optimization and mutant optimization. A similar study on SBMT has been recently published by Silva et al. [14]. The authors explored the use of meta-heuristic in the context of mutation testing. The study is very rigorous as well as detailed and reviews the publications up to the year 2014. The studies included in this SLR were also been in the systematic review by Silva et al. This SLR primarily aims to review the studies in the field of SBMT up to the year 2014, however, a few relevant studies of the year 2015 and 2016 were also cited. These recent papers were collected using the snowballing approach [15] to find the papers published in the last two years which cited the studies included in present SLR. The study and analysis of various papers collected from a number of sources involves a great deal of research effort and time as it also encompasses the review and modification of the paper entailing s time after it is communicated to a journal. Review and modification of the paper also entails time after it is communicated to a journal. Therefore, in this paper the relevant studies up to the year 2014 were thoroughly analysed and included only few studies published in the year 2015–2016. The primary studies selected for review are the ones which proposed a new technique for SBMT up to the year 2014. This SLR contains many figures and ttables to offer easy access to comprehensive knowledge on the topic. The studies selected for review were used immensely by the researchers working in the field of SBMT.

The rest of the paper is systematized as follows: Section 2 differentiates between Literature Review (LR), Systematic Mapping (SM) and Systematic Literature Review (SLR). Section 3 shows the research method followed for this SLR highlighting the research question and the used search strategy. Section 4 analyses the studies in SBMT. Section 5 presents the results and then Section 6 presents conclusion.

2. Differentiating LR, SM and SLR

Reviewing the literature in the field is a fundamental process prior to any worthwhile research. This section compares and contrasts the implications of LR, SM and SLR.

Literature review – “A literature review is an objective study done to thoroughly summarize and critically analyse the available relevant research, and to enable the researcher to gather up to date information, to gain insight into the current literature that forms a basis of a goal to be achieved and also justifies the future research in that area” [16].

Systematic mapping – “A systematic mapping study provides a structure of the type of research reports and results that have been published by categorizing them. It often gives a visual summary, the map, of its results. It requires less effort while providing a more coarse-grained overview” [17].

Systematic literature review – “A systematic literature review is a means of identifying, evaluating and interpreting all available research relevant to a particular research question, or topic area, or phenomenon of interest. Individual studies contributing to a systematic review are called primary studies; a systematic review is a form a secondary study” [18].

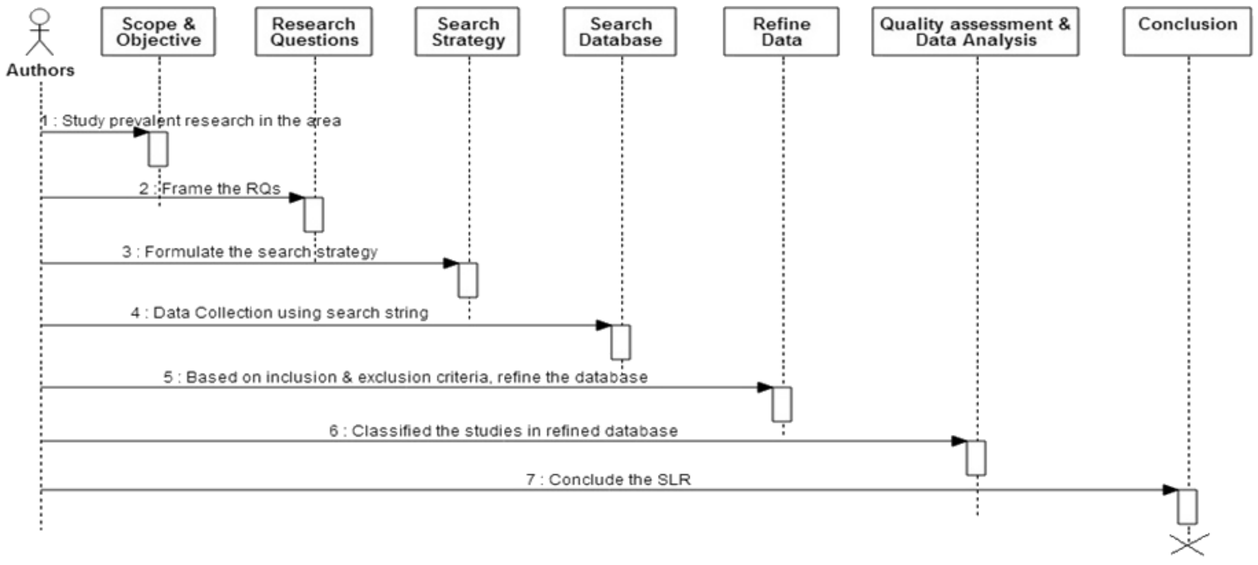

Authors
Scope & Objective
Research Questions
Search Strategy
Search Database
Refine Data
Quality assessment & Data Analysis
Conclusion
1 : Study prevalent research in the area
2 : Frame the RQs
3 : Formulate the search strategy
4 : Data Collection using search string
5 : Based on inclusion & exclusion criteria, refine the database
6 : Classified the studies in refined database
7 : Conclude the SLR

Figure 1. Undertaken course of action for SLR

SLR entails an exhaustive and comprehensive search process which adheres to the guidelines on the conduct of a review and inclusion/exclusion criteria, it is determined by the quality assessment process. However, in the case of LR the process may not be comprehensive and may or may not include the quality assessment. SM, on the other hand, has a complete search process and identifies primary and secondary research but may not include formal quality assessment process [19].

3. Research method

The aim of this study is to offer insight into various types of research carried out in the area of search based mutation testing (SBMT). The authors followed the guidelines given by Kitchenham [18]. The undertaken course of action is demonstrated using a sequence diagram (Figure 1).

Initially, primary knowledge about the subject of the study was collected. As a result of an inquisitive study of the research conducted in the field, a set of research questions (RQs) were formulated. Search strategy was then designed that forms the base for collecting the relevant research material from the data repositories. The most vital step was to refine the data gathered by defining an inclusion/exclusion criterion. After this stage, 43 relevant studies were categorized into 18 primary and 25 secondary studies (discussed in this section). The method used to extract the primary and secondary studies is explained in [20]. In order to answer the RQs, collected and segregated material was critically examined; thus, reaching the conclusion.

The steps trailed in the SLR for test data generation/optimization and mutant optimization using search based approaches were conducted in the way elaborated in the following subsections.

3.1. Research questions (RQs)

As the RQs form the foundation of the SLR, the PICOC criteria [20] (given in Table 1) were used to define the research questions. The PICOC criteria were defined as follows:

- **Population** – people, or an application area, or a group of companies or any such community affected by the research.
- **Intervention** – software methodologies/tools/technologies/procedures which are required to solve a particular issue.
- **Comparison** – software methodologies/tools/technologies/procedures with which intervention can be compared.
- **Outcomes** – factors of significance which are relevant in the study.
- **Context** – environment in which the comparison takes place.

The aim of this work is to summarize the current state of art of research in SBMT by

Table 1. PICOC criteria applied to SLR

Population	Search based Mutation Testing
Intervention	Search based/meta-heuristic techniques
Comparison	Approaches which are not meta-heuristic
Outcomes	Test data generation/optimization and mutant optimization techniques involved SBMT
Context	Within the domain of SBMT with a focus on empirical studies

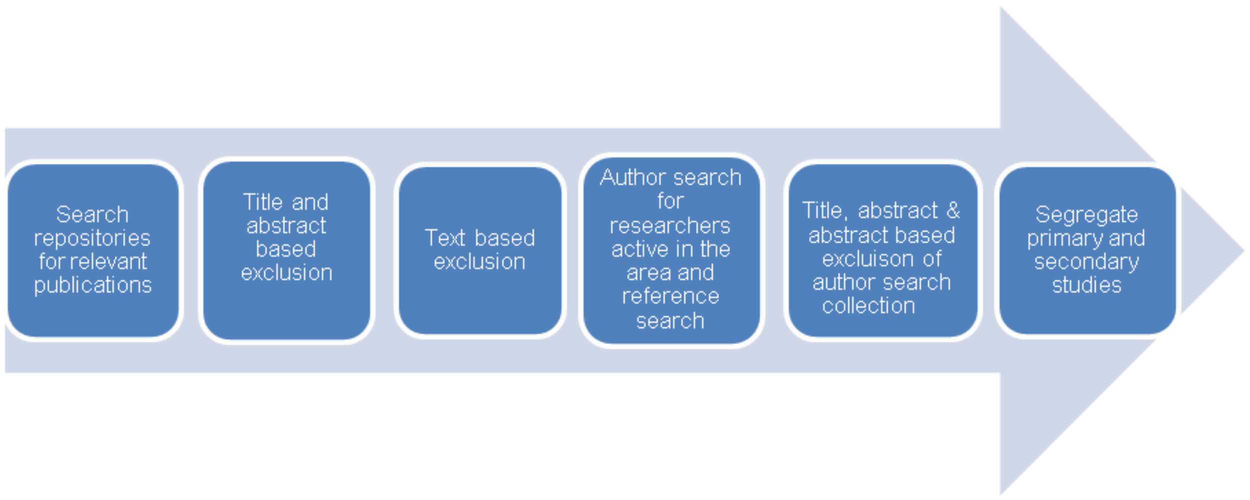

Search repositories for relevant publications
Title and abstract based exclusion
Text based exclusion
Author search for researchers active in the area and reference search
Title, abstract & abstract based excluison of author search collection
Segregate primary and secondary studies

Figure 2. Inclusion/exclusion criteria

proposing answers to the set of the following research questions(RQs).
RQ1: Which search based approaches were used in collaboration with mutation testing?
RQ2: What are the areas of application in which search based approaches were applied for mutation testing?
RQ3: What are the findings from the comparison studies of the techniques used in SBMT?
RQ4: What are the major challenges faced by the researchers in the field of SBMT?

3.2. Search process

This section describes the strategy used to mine the databases containing the research material and to extract the relevant information related to the conducted research.

3.2.1. Data sources

According to the guidelines provided by Kitchenham [20], several data sources were searched to encompass the maximum possible information. The following data repositories were explored to retrieve the relevant publications from conference proceedings, workshop proceedings, journal articles, books and theses.

- ACM digital library (www.dl.acm.org).
- IEEE Xplore Digital Library (www.ieeexplore.ieee.org).
- Science Direct (www.sciencedirect.com).
- Springer (www.springerlink.com).
- Citeseer (scholar.google.com).
- Wiley Online Library (onlinelibrary.wiley.com).

A few papers were available in more than one searched repositories and in such cases the duplicate copies were removed manually.

3.3. Inclusion and exclusion criteria

The overall process depicted in Figure 2 was followed to obtain and segregate the relevant primary and secondary studies.

3.3.1. Search strategy

Various research publication repositories provide different search options for data search. An

Table 2. Search Strings for selected data sources

Data Source	Search String
IEEE	((((Evolutionary OR heuristic OR search based OR search-based OR nature inspired OR nature-inspired OR optimization OR selection OR minimization OR prioritization)) AND (“Mutation Testing” OR “Mutation Analysis” OR “mutation operator testing” OR “fault injection” OR “fault based testing”))) in command search tab (under advanced search)
ACM	(Abstract:Mutation and Abstract:testing) and (Abstract:Evolutionary or Abstract:heuristic or Abstract:metaheuristic or Abstract:search based or Abstract:search-based or Abstract:nature or Abstract:inspired or Abstract:nature-inspired or Abstract:optimization or Abstract:optimisation or Abstract:selection or Abstract:minimization or Abstract:minimisation or Abstract:prioritisation or Abstract:prioritization)
Wiley	(Evolutionary OR heuristic OR metaheuristic OR search based OR search-based OR nature inspired OR nature-inspired OR optimization OR optimisation OR selection OR minimization OR minimisation OR prioritisation OR prioritization) in Abstract AND “Mutation Testing" OR “Mutation Analysis” OR “Mutants testing” OR “mutation operator testing” OR “fault injection” OR “fault based testing” in Abstract)
Elsevier	(Evolutionary OR heuristic OR metaheuristic OR search based OR search-based OR nature inspired OR nature-inspired OR optimization OR optimisation OR selection OR minimization OR minimisation OR prioritisation OR prioritization) and TITLE-ABSTR-KEY (“Mutation Testing” OR “Mutation Analysis” OR “mutation operator testing” OR “fault injection” OR “fault based testing”)
Springer	Mutation AND testing AND (Evolutionary OR heuristic OR metaheuristic OR search based OR search-based OR nature OR inspired OR nature-inspired OR optimization OR optimisation OR selection OR minimization OR minimsation OR prioritisation OR prioritization)
citeseer	((((Evolutionary OR heuristic OR metaheuristic OR search based OR search-based OR nature inspired OR nature-inspired OR optimization OR optimisation OR selection OR minimization OR minimisation OR prioritisation OR prioritization)) AND (“Mutation Testing” OR “Mutation Analysis” OR “Mutants testing” OR “mutation operator testing” OR “fault injection” OR “fault based testing”)))

advanced search strategy was used for mining the data sources to locate the pertinent publications. The Boolean operators “AND”, “OR” and “NOT” are used to arrange the keywords for forming the search string. As the various databases provide different search capabilities, as stated in a recent SLR [21, 22], conceptually similar search strings were used for each of the data sources listed in Table 2. To ensure the maximum retrieval of significant material, the search strategy was applied on to a title, an abstract and keywords. The earliest research in the field of search based on testing was published in the year 1976, and thus the start date for the search was established to be January, 1976 and the end date was set to December, 2014.

Initially using the search string as specified in section 3.2.2, a total of 4314 papers were found. However, many of these papers pertained to mutation and biology which were not relevant to our field. Thereafter, a multi-step process was followed to remove the irrelevant publications. The number of papers shown by the respective repositories using the aforesaid search string is listed in Table 3.

In order to exclude the irrelevant papers, title and abstract based exclusion was performed manually and then the full text was read to store the appropriate papers in our repository. After analysing the papers thus found, a list of active researchers (as listed in Table 4) in the same field was maintained. Then, to ensure the completeness of the data, another search

Table 3. Initial number of papers obtained by searching the data repositories

Data source	Initial Count
IEEE	201
ACM	448
Wiley	1844
Elsevier	20
Springer	1484
Citeseer	317

was accomplished to locate the leftover papers of the researchers working in the field. The reference section of the primary studies was also checked to extract any relevant publication that was missing in our collection. A total of 10 new papers were located out of which 6 were found by reference search and 4 were found by the author search. The total number of relevant papers thus collected is listed in Table 5.

Table 5. Number of papers after exclusion

Data source	Count after exclusion
IEEE	16
ACM	6
Wiley	1
Elsevier	4
Springer	10
Citeseer	1
Others	5

The totally pertinent publication includes all those papers which presented a new technique(s), compares existing techniques and empirically evaluated the techniques. These 43 studies were then thoroughly reviewed by each author to segregate them into primary and secondary studies. The papers with a significant contribution in the field in terms of research ideas and development were chosen as primary studies and the rest were marked as secondary ones. Eventually, 18 primary studies (as shown in Table 6) were then thoroughly, analysed separately by each author. It can be seen that 50% of the studies demonstrated their technique in a theoretical manner and the remaining 50% evaluated their technique empirically.

Table 4. List of authors actively working in the field of SBMT

Authors	Total publications in SBMT
P. May	4
K. Ayari	1
M. Harman	6
M. Papadakis	5
G. Fraser	3
B. Baudry	5
M. Rad	2

There are a few papers which were not been included in this SLR that were included in the recent SLR on SBMT [14]. The papers encompass [39–47]. These papers are relevant to the field of SBMT but their contribution to the field is not novel as the techniques they proposed are already used by researchers in the field of SBMT.

The papers published in 2015 and 2016 that are relevant to the field of SBMT were collected by snowballing. They are listed in the following section.

4. Analysis of studies in SBMT

This section presents the analysis, trend patterns and the discussion of the research done in the field of SBMT.

4.1. Trends in SBMT

Table 7 and Table 8 list the publication types for the selected studies and primary studies retrieved from the repositories. Figure 3 shows the publication trends observed from 1998 to 2014. The first publication was recorded in 1998 and after that the research was carried out continuously in this field. A decline in the research trends is observed during the year 2006–2009.

4.2. Discussion on primary studies

Table 9 summarizes the contribution of the selected primary studies and search based techniques evolved/used by them. Table 10 lists the subject programs, language and tools used by the selected primary studies. It is observed that

Table 6. Overview of the selected primary studies

Study ID	First Author	Year	Type (C/J/W/B)	Publisher	Research Category (E/T)	References
P1	L. Bottaci	2001	C	Others	T	[9]
P2	B. Baudry	2001	W	Springer	E	[23]
P3	B. Baudry	2002	C	IEEE	E	[24]
P4	P. May	2003	B	Springer	T	[25]
P5	M. C. F.P. Emer	2003	J	Others	E	[26]
P6	K. Adamopoulos	2004	C	Springer	T	[27]
P7	Md. M. Masud	2005	C	IEEE	T	[28]
P8	K. Ayari	2007	C	ACM	E	[29]
P9	Y. Jia	2008	C	IEEE	E	[30]
P10	K. K. Mishra	2010	C	IEEE	T	[31]
P11	B. Schwarz	2011	C	IEEE	T	[32]
P12	M. Harman	2011	C	ACM	T	[33]
P13	J.J. Dominguez-Jimenez	2011	J	Elsevier	E	[34]
P14	G. Fraser	2012	J	IEEE	E	[35]
P15	A. A. L. de Oliveira	2013	C	IEEE	E	[36]
P16	M. B. Bashir	2013	C	IEEE	E	[37]
P17	P. S. Yiasemis	2013	C	Springer	T	[38]
P18	M. Papadakis	2013	J	Springer	T	[10]

Type: C – Conference, W – Workshop, B – Book, J – Journal
Research Category: E – Experimental, T – Theoretical

Table 7. Publication type for 43 selected studies

Publication type	Number
Journal	13
Workshop	3
Thesis	1
Conference	23
Book	3

Table 8. Publication type for primary studies

Publication type	Number
Journal	4
Workshop	1
Book	1
Conference	12

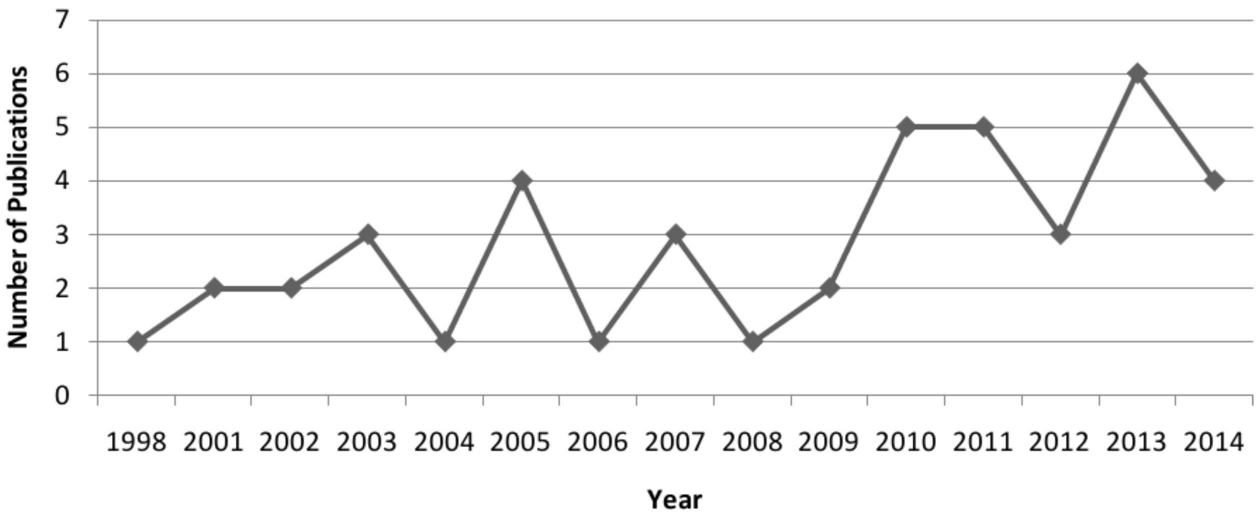


Figure 3. Publication trends of research in SBMT

Table 9. Contribution and techniques used by primary studies

Study no.	Contribution	Techniques	References
P1	Fitness function introduced based on GA for mutation testing	GA	[9]
P2	Automation of test case enhancement for object oriented software components	GA	[23]
P3	Introduced bacteriological algorithm and compared with GA	GA,BA	[24]
P4	Artificial Immune System applied for Mutation testing	AIS	[25]
P5	GP based procedure for selection and evaluation of test data	GA	[26]
P6	Dealt with equivalent mutant problem using GA	GA	[27]
P7	Test case generation for killing mutants in program units using GA	GA	[28]
P8	Test case generation using mutation testing with ACO	ACO	[29]
P9	Construction and evaluation of higher order mutants using GA, HC, Greedy	GA, HC, Greedy	[30]
P10	Elitist genetic algorithm applied with mutation testing	GA	[31]
P11	Generation of high impact mutants avoiding equivalent mutants using GA	GA	[32]
P12	Production of test input via strong higher order mutants	Search based	[33]
P13	Mutant optimization using GA for WS-BPEL	GA	[34]
P14	Test oracle generation using automated mutation testing, assertions and GA	GA	[35]
P15	Test case and mutant optimization using GA	GA	[36]
P16	Definition of a new fitness function aiming to produce test data	GA	[37]
P17	Automatically finding and correcting faults using code slices, GA and mutation testing	GA	[38]
P18	Improved fitness function for test generation using AVM	HC	[10]

Java programs are most popular for the research and there are different tools that are used by the researchers in the field of SBMT.

Figure 4 describes the yearly distribution of the techniques used by the selected primary studies. As is evident from the scatter chart, GA is the most prevalent and consistently used technique by the researchers working in the area of SBMT. Study P12 is not included in the chart, as it evaluates the application of search based techniques in general (no specific technique) for higher-order mutation testing.

4.3. Relevant publications of 2015–2016

In order to collect publications in the domain of SBMT for the years 2015–2016, the snowball approach was followed. The relevant studies collected asa result are cited here.

A few reviews relevant to the ones filed in SBMT have been published in the past two years. The literature review by Silva et al. [14] details the work carried out in the field before 2014. Other surveys include [13, 48–51].

Considerable work was done in the context of search based higher order mutation testing by eminent researchers in the field. The relevant publications include [52–56].

GA is still the most popular search algorithm used by the researchers of the domain [52, 57–59].

Hill Climbing [60] and PSO [61] have also been recently used by researchers for test data generation using SBMT. Other relevant publications include [62, 63].

5. Results (RQs)

This section presents answers to the RQs formulated above after analysing the 43 studies in SBMT, amongst which 18 studies that were identified as those stating a new technique, were thoroughly analysed.

Table 10. Language used, subject programs and tools used by (Experimental) primary studies

Study no.	Contribution	Techniques	References
P2	*	Pylon library	uSlayer
P3	C#	C# parser	*
P5	C++	cmm, fat, max, cmd	GPTesT
P8	Java	Triangle, Nextdate	muJava
P9	C	triangle, TCAS, schedule2, totinfo, printtokens, space	MILU
P13	WS-BPEL	WS-BPEL compositions (Travel Reservation, Service Extended Meta Search and Loan Approval Extended)	GAmera
P14	Java	commons CLI, commons codec, commons collections, commons logging, commons Math, commons Primitives, Google Collections, JGraphT, Joda Time, NanoXML	muTEST
P15	*	Bisect, Bub, Fourballs, Mid, trityp	*
P16	Object oriented	CGPA calculation	*
P17	Java	credit card validator, triangle classification, Base 64, Person sorted list, shapes, order set, Graph shortest Path, 3 Eclipse libraries	Kaveri/Indus
P18	Java	Triangle, Trityp, Triangle, Remainder, Calender, Fourballs, Cancel, Quadratic	*

* Indicates the data not mentioned by the authors concerned in the study.

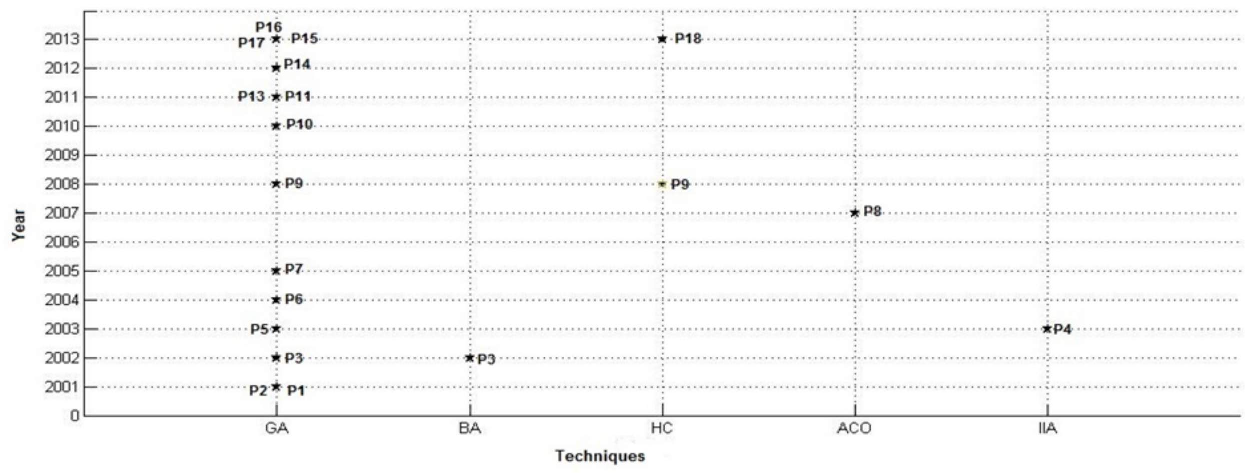


Figure 4. Corroboration of selected primary studies

5.1. Techniques of SBMT (RQ1)

In order to answer RQ1, the techniques used by the researchers working in the field are summarised below.

5.1.1. Hill climbing (HC)

It is one of the meta-heuristic search based techniques that strives to improve the current solution by exploring all its neighbours in the search space. It includes an initialization stage, where the initial candidate solution is chosen randomly. Thereafter, all its neighbours are searched and evaluated until no improved solution can be found. The major drawback of this technique is local convergence as search includes the neighbourhood space only [64]. Jia and Harman [30] applied hill climbing with GA to deal with the explosion of a large number of higher order mutants created from lower order mutants. Papadakis and Malveris [10] used Alternating Variable Method, which is a variant of Hill Climbing, for test data generation using mutation testing by using an improved fitness function.

5.1.2. Ant colony optimisation (ACO)

ACO is a metaheuristic technique proposed by Dorigo in the late 1990s [65,66] for approximately solving the combinatorial problems which are otherwise hard to solve in a reasonable amount of time. It was inspired by Ant System [67] in 1991. ACO starts by mapping the problem under study such that it simulates the stigmergic behaviour of ants. The ants start at random first to locate a food source and then they lay a chemical compound known as pheromone on the most pertinent paths. The ants thereafter follow the pheromone trails to reach the desired destination. ACO was applied to obtain approximate solutions to many optimisation problems [68]. ACO finds its application in the domain of software testing [69]. Ayari et al. [29] proposed and applied ACO for automatic test data generation in the context of mutation testing and empirically proved that ACO gives better results as compared to Hill Climbing and the random approach.

5.1.3. Genetic algorithm (GA)

The idea of GA was initially proposed by Fraser [70] and Bremermann [71] although it was popularised by Holland and his students because of applying its concepts in the field of computer science [72]. GA starts with randomly generated populations which are then evolved using genetic operators (mutation and crossover). The evolved population is evaluated for fitness and the best performing individuals are chosen. The process continues until stopping criteria are met. GA finds its application in the software testing domain for test data generation, mutant optimisation and minimisation of test data [73].

GA was used by various researchers working in the area of SBMT, but the conducted research is varied in nature. The paragraph below presents how GA was used for the optimisation of the process of search based mutation testing in various aspects.

Bottaci [9] gave a new fitness function for GA that works in collaboration with mutation testing. The fitness function was composed of three conditions: (i) Reachability: the path followed by the execution of test case must reach the mutated statement, (ii) Necessity: the condition stated by the mutated expression must be satisfied for it to be killed and (iii) Sufficiency: the difference between the subject program and the mutated program must be propagated to the final output. This fitness function was further used by Masud et al. [28], Ayari et al. [29]. Baudry et al. [23] used mutation analysis for integration testing by enhancing the test cases using GA. Baudry also applied GA for test case optimization for C# parser [24]. Emer et al. [26] developed a tool named GPTest for C++ programs for the selection and assessment of test data using fault based testing. GA was also used to address the problem of equivalent mutants in mutation testing by Adampolous et al. [27]. Masud et al. [28] proposed a model to apply GA for exposing faults by splitting the subject program into small units, hence reducing the cost and execution time of test cases. Jia and Harman [30] introduced the paradigm of higher order mutation testing and applied GA for optimization and selection amongst the huge number of higher order mutants (HOMs) from the lower order ones. Later, HOMs were extensively explored in [74]. Omar et al. [39, 40] introduced search algorithms to find subtle HOMs and concluded that local search performed better than GA and the random search in finding HOMs for Java and AspectJ subject programs. Elitist GA was used by Mishra [31] for the generation of efficient test data by selecting those test cases which have already killed a large set of mutants. Dominguez-Jimenez [34] used GA for a reduction in mutants which eventually derive test cases for improving the quality of initial test cases for WS-BPEL compositions. Fraser et al. [35] also optimised test cases using mutation analysis and GA and evaluated them on 10 open source libraries. GA has also been used by [36–38] for the optimisation of the process of mutation testing.

5.1.4. Bacteriological algorithm (BA)

BA was proposed by Baudry et al. [24, 75, 76] and is inspired by the bacteriological adaptation. This technique was particularly recommended for improving the test cases using mutation testing. It takes a set of test cases as input and makes small changes in them calling them bacteria. These bacteria are evaluated for fitness using a memorization function. The technique evolved from GA and it differs from it by two key points: introduces the memorization function for the evaluation of fitness and suppresses the crossover operator which was the cause of slow convergence in population evolution in GA.

5.1.5. Immune inspired algorithm (IIA)

Immune inspired algorithms are a class of intelligent algorithms which uses the primciples associated with the immune system of vertebrates. The characteristics exploited here are related to learning and memory and they are used to solve a problem. May et al. [25] applied these characteristics for optimization of test cases and mutants. May et al. further worked on this approach [77, 78] and successfully mapped the evolutionary approach to the process of the optimised generation of test cases and mutants. The process starts by choosing initial test cases called antibodies which are evolved through multiple iterations by seeking those which are capable of killing more mutants that are referred to as antigens. The mutation score is denoted by Affinity. Figure 5 shows the search based techniques used by researchers in SBMT. It clearly shows that the Genetic Algorithm is the one which is the most frequently used by the authors.

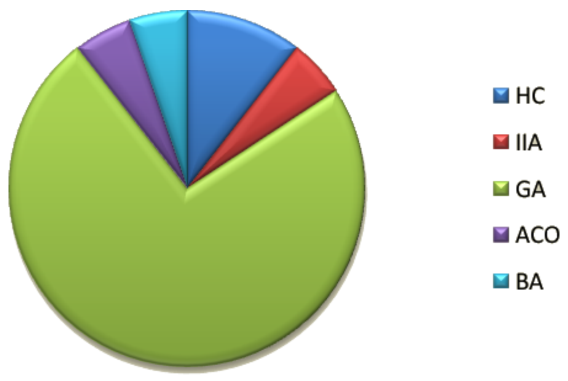


Figure 5. Usage of search based techniques

5.2. Applications of SBMT (RQ2)

After the analysis of the 18 primary studies selected for the research, it was observed that SBMT found its application as shown below in Figure 6. Amongst the 18 primary studies, 9 were identified for their work in test case generation, 3 worked on test case optimization, 5 worked on mutant optimisation and the 2 remaining ones contributed to both test case and mutant optimisation.

5.2.1. Test data generation (TDG)

The process of TDG [2,5] using mutation testing begins with the generation of a set of mutants and the execution of the test suite on them. Then the required test suite then iteratively collects those test cases which can kill the mutants. Since this process is computationally expensive and time consuming, search based approaches [7, 8] were applied to automate this process. As is evident from Figure 6, nine studies representing TDG were conducted. Out of these nine studies, six [9, 23, 28, 31, 35, 37] are based on GA. Ayari et al. [29] applied ACO and Papadakis et al. [10] used a variant of HC for generating test data using the process of mutation testing. The symbolic execution and concolic testing were also suggested for test data generation by making use of weak mutation testing [79]. Harman et al. [33] evaluates the application of the search based technique in general (not any specific technique) for higher-order mutation testing.

5.2.2. Test case optimization/minimization/prioritization (TCO)

The search based techniques were used for the optimized reduction or prioritization of test cases using mutation testing wherein the test cases that are capable of killing maximum mutants are considered at first hand, so that the maximum number of faults is covered with the minimum number of test cases. The search based techniques guide this search process in promising

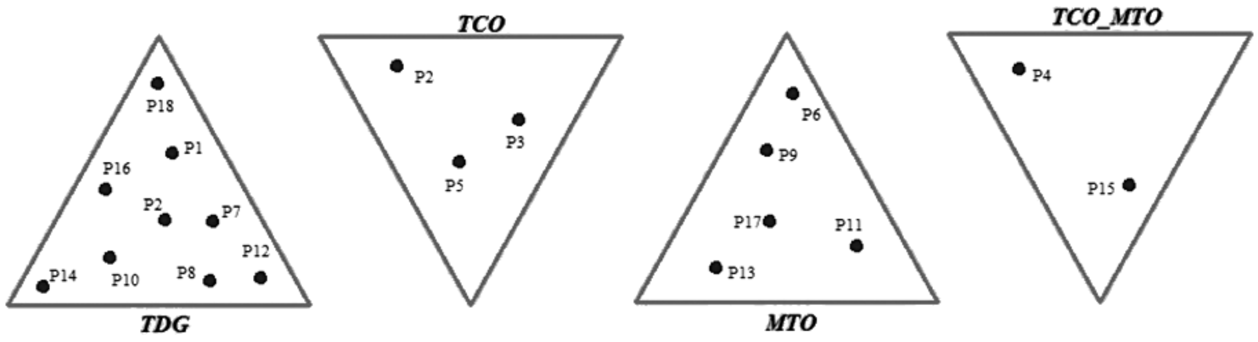


Figure 6. Categorization of primary studies according to SBMT application

directions yielding optimum results in less computational time. As shown in Figure 6, three studies [23, 24, 26] contributing towards TCO were identified. GA were used in [23,26] for test case optimisation and [76] for the comparison and evaluation of GA with BA for purpose of the optimisation of test cases.

5.2.3. Mutant optimization (MTO)

The execution of a large number of mutants requires enormous computational efforts. Owing to this, a large number of program mutants needs to be optimised to obtain a reduced set of mutants to that they are capable of detecting the majority of significant plausible faults in the program under test. Figure 6 depicts five studies [27, 30, 32, 34, 38] found to implement the techniques for MTO. GA was predominantly adopted to obtain the optimised set of mutants [27,32,34,38]. Jia et al. [30] evaluated GA with HC and the Greedy Approach for creation of higher order mutants.

5.2.4. Mutant and test case optimisation (TCO_MTO)

Some researchers proposed approaches to optimise mutants as well as test cases using search based techniques with mutation testing. May et al. [25] proposed the AIS technique and Oliveira et al. [36] used GA for the purpose of mutant and test case optimisation. Figure 6 depicts the categorization of primary studies in the domain of SBMT.

Figure 6 shows that despite the fact that test case generation is a domain containing more studies, most approaches in this domain were concentrated on applying the Genetic Algorithm (6 studies). Mutant optimization is another prominent area in which researchers were specially interested in, and here also GA was actively applied (6 studies). Basically, SBMT relies on the fitness function which searches for candidate solutions and on mutation testing which is mainly a quality assessment technique. Both the fitness function and the quality assessment criteria determine the level to which the approach satisfies the analysed problem at hand. Thus test case generation is the most worked upon area as it is test case generation is the prior requirement in the field of software testing.

5.3. Findings from comparison studies (RQ3)

Ayari et. al [29] proposed ACO and compared it with GA,HC and Random Search algorithms. Their preliminary results on two small programs (Triangle and NextDate) show that ACO outperformed the other algorithms in terms of mutation score and the convergence factor. Baudry et. al [23,24,76] introduced BA which works on a similar principle as GA, but differs from it by memorizing the efficient test cases across generations. The authors compared their work [75] with GA on 32 classes of C#. They state that BA is more stable than GA and converges faster. Jia and Harman [80] generated subsuming higher order mutants using search based techniques (GA, HC) and the greedy approach. The subsuming higher order mutants are those which are difficult to kill (detect). The authors applied the techniques (GA, HC and Greedy Approach) on 10 programs and stated that GA is the most efficient among them for the generation of subsuming higher order mutants. May et. al [77] proposed a new approach (Immune Inspired Algorithm) for evolution of test data using mutation testing adequacy criteria and evaluated the efficiency of their approach on 4 programs compared with the Genetic Algorithm. They state that their approach gives better results in comparison with the others. Researchers working in the area of SBMT proposed various approaches and most of them evaluated their approach against GA.

5.4. Major challenges faced by researchers (RQ4)

The challenges faced by researchers in the field of SBMT are significantly related to those faced by mutation testing and those in SBST in general. The challenges (i)–(iv) as listed in [81, 82] are those dealing with mutation testing in general. The challenges (v)–(vii) are in context with search based software testing [83] which is applicable to SBMT as well. The challenges (viii)–(xi) are the findings of this SLR. The following points address RQ4.
(i) Computational cost of executing a large number of mutants is very high.
(ii) Mutants generated from the traditional mutant operators may be trivial (easily killable).
(iii) Detection of equivalent mutants is a cumbersome task.
(iv) Checking the output of each test case for every mutant with that of the original program takes a significant amount of time.
(v) Work on SBMT has mainly focussed on single objective optimisation, multi-objective test data generation is yet to be achieved.
(vi) SBMT focuses on unit structural test data generation. Other areas (non-functional, etc.) of software testing remains unexplored in this field.
(vii) Tools that qualify FiFiVerify (Find Fix and Verify) challenge [83] are missing in the domain of SBMT.
(viii) Most researchers of SBMT worked on artifacts (benchmark programs) of small size and several of those are artificial examples.
(ix) Most researchers worked independently, trying to find new SBMT techniques rather than extending or enhancing the proposed approaches of other researchers.
(x) Search techniques used in SBMT were applied mostly to single order mutants, however, less work has been carried out for higher order mutants.
(xi) Updates a few tools (like MuClipse) which are still being used by researchers of SBMT are not being updated with the optimisation standards which are used nowadays in mutation testing as per latest research.

6. Conclusion

In this paper, the results from a systematic review of search based mutation testing are presented. The following are the findings in regard to the research questions:
RQ1: In the 18 primary studies identified in this work, search based techniques namely HC, ACO, GA, BA and IIA were empirically evaluated in the area of mutation testing. The other search based techniques have not yet been applied in this field.
RQ2: Search based techniques were applied to mutation testing for test data generation, selection, minimization and optimisation and also for mutant optimization.
RQ3: Most researchers who proposed a new SBMT technique empirically evaluated their technique with GA. However, there is a lot of variance in the uniqueness of the identified search based techniques. Some techniques may be treated as novel at the time of their publications, while others may be considered as slight variations of already existing techniques. No technique can be said to be distinctly better than another as the programs used for empirical evaluation may not be considered as strong evidence to prove the superiority.
RQ4: The challenges prevailing in the area of mutation testing are pertinent to the area of SBMT along with a few more challenges. The major ones include the effort and cost entailed in mutation testing, and thus limit its application

to testing real world programs. Most techniques given by researchers are either presented in a general manner or are not sufficiently empirically evaluated to serve as a base for enabling a practitioner to choose a specific SBMT technique for given software.

The analysis of the studies collected for the 2015–2016 shows that the Genetic Algorithm is still being used most frequently by the researchers in the field of SBMT. The inclination of the interest of researchers towards Higher Order Mutation testing can also be observed. Alongwith other metaheuristics, PSO is also used for test data generation using SBMT.

The findings in the area of search based mutation testing, wrapping its application domain and the used techniques are covered here. This work identified research in the field since its evolution in the year 1976. The results of this SLR show that there is a significant research gap in the area of SBMT. Despite the fact that the research has been conducted in this area for more than 40 years, only 18 studies were segregated as primary studies out of 43 relevant studies. In addition to this, the problems in the area are still prevalent and not much work has been carried out to resolve them. Few techniques applied in the area of SBMT have been applied and tested on small programs and thus may not be scalable to the industrial needs and real software. Researchers worked independently rather than working collaboratively towards the elimination of open problems in the area. As a result, a lot of research can be carried out in the area including the work on the feasibility analysis of SBMT, applying SBMT to other programming languages, effective approaches for test data generation using SBMT, complexity analysis of approaches used in SBMT and reduction in the overall cost of SBMT.

References

[1] G.J. Myers, C. Sandler, and T. Badgett, *The art of software testing*. John Wiley & Sons, 2011.

[2] J. Edvardsson, "A survey on automatic test data generation," in *Proceedings of the 2nd Conference on Computer Science and Engineering*, 1999, pp. 21–28.

[3] M. Prasanna, S. Sivanandam, R. Venkatesan, and R. Sundarrajan, "A survey on automatic test case generation," *Academic Open Internet Journal*, Vol. 15, 2005. [Online]. http://www.acadjournal.com/2005/v15/part6/p4/

[4] H. Tahbildar and B. Kalita, "Automated software test data generation: Direction of research," *International Journal of Computer Science and Engineering Survey*, Vol. 2, No. 1, 2011, pp. 99–120.

[5] S. Anand, E.K. Burke, T.Y. Chen, J. Clark, M.B. Cohen, W. Grieskamp, M. Harman, M.J. Harrold, P. McMinn *et al.*, "An orchestrated survey of methodologies for automated software test case generation," *Journal of Systems and Software*, Vol. 86, No. 8, 2013, pp. 1978–2001.

[6] C.A.C. Coello, "A comprehensive survey of evolutionary-based multiobjective optimization techniques," *Knowledge and Information systems*, Vol. 1, No. 3, 1999, pp. 269–308.

[7] P. McMinn, "Search-based software test data generation: A survey," *Software Testing Verification and Reliability*, Vol. 14, No. 2, 2004, pp. 105–156.

[8] P. McMinn, "Search-based software testing: Past, present and future," in *IEEE Fourth International Conference on Software Testing, Verification and Validation Workshops (ICSTW)*. IEEE, 2011, pp. 153–163.

[9] L. Bottaci, "A genetic algorithm fitness function for mutation testing," in *Proceedings of the SEMINALL-workshop at the 23rd International Conference on Software Engineering*, Toronto, Canada, 2001.

[10] M. Papadakis and N. Malevris, "Searching and generating test inputs for mutation testing," *SpringerPlus*, Vol. 2, No. 1, 2013.

[11] F. Souza, M. Papadakis, V.H. Durelli, and M.E. Delamaro, "Test data generation techniques for mutation testing: A systematic mapping," *Proceedings of the 11th ESELAW*, 2014, pp. 1–14.

[12] Y. Jia and M. Harman, "An analysis and survey of the development of mutation testing," *IEEE Transactions on Software Engineering*, Vol. 37, No. 5, 2011, pp. 649–678.

[13] N. Jatana, S. Rani, and B. Suri, "State of art in the field of search-based mutation testing," in *4th International Conference on Reliability, Infocom Technologies and Optimization (ICRITO) (Trends and Future Directions)*. IEEE, 2015, pp. 1–6.

[14] R.A. Silva, S. do Rocio Senger de Souza, and P.S.L. de Souza, "A systematic review on search

based mutation testing," *Information and Software Technology*, 2016.

[15] C. Wohlin, "Guidelines for snowballing in systematic literature studies and a replication in software engineering," in *Proceedings of the 18th International Conference on Evaluation and Assessment in Software Engineering.* ACM, 2014, p. 38.

[16] P. Cronin, F. Ryan, and M. Coughlan, "Undertaking a literature review: A step-by-step approach," *British Journal of Nursing*, Vol. 17, No. 1, 2008, p. 38.

[17] K. Petersen, R. Feldt, S. Mujtaba, and M. Mattsson, "Systematic mapping studies in software engineering," in *12th International Conference on Evaluation and Assessment in Software Engineering*, G. Visaggio, M.T. Baldassarre, S. Linkman, and M. Turner, Eds., 2008, pp. 68–77.

[18] B. Kitchenham, "Procedures for performing systematic reviews," Keele University, Keele University, Keele, Staffs, UK, Joint Technical Report TR/SE-0401, 2004. [Online]. http://csnotes.upm.edu.my/kelasmaya/pgkm20910.nsf/0/715071a8011d4c2f482577a700386d3a/$FILE/10.1.1.122.3308[1].pdf

[19] M.J. Grant and A. Booth, "A typology of reviews: An analysis of 14 review types and associated methodologies," *Health Information and Libraries Journal*, Vol. 26, No. 2, 2009, pp. 91–108.

[20] B. Kitchenham and S. Charters, "Guidelines for performing systematic literature reviews in software engineering," Keele University & University of Durham, EBSE Technical Report EBSE 2007-01, 2007.

[21] W. Orzeszyna, L. Madeyski, and R. Torkar, Protocol for a systematic literature review of methods dealing with equivalent mutant problem. [Online]. http://madeyski.e-informatyka.pl/download/slr/EquivalentMutantsSLRProtocol.pdf

[22] L. Madeyski, W. Orzeszyna, R. Torkar, and M. Jozala, "Overcoming the equivalent mutant problem: A systematic literature review and a comparative experiment of second order mutation," *IEEE Transactions on Software Engineering*, Vol. 40, No. 1, 2014, pp. 23–42.

[23] B. Baudry, V. Le Hanh, J.M. Jézéquel, and Y. Le Traon, "Trustable components: Yet another mutation-based approach," in *Mutation testing for the new century.* Springer, 2001, pp. 47–54.

[24] B. Baudry, F. Fleurey, J.M. Jézéquel, and Y. Le Traon, "Genes and bacteria for automatic test cases optimization in the .NET environment," in *13th International Symposium on Software Reliability Engineering.* IEEE, 2002, pp. 195–206.

[25] P. May, K. Mander, and J. Timmis, "Software vaccination: An artificial immune system approach to mutation testing," in *International Conference on Artificial Immune Systems.* Springer, 2003, pp. 81–92.

[26] M.C.F. Emer and S.R. Vergilio, "Selection and evaluation of test data based on genetic programming," *Software Quality Journal*, Vol. 11, No. 2, 2003, pp. 167–186.

[27] K. Adamopoulos, M. Harman, and R.M. Hierons, "How to overcome the equivalent mutant problem and achieve tailored selective mutation using co-evolution," in *Genetic and evolutionary computation conference.* Springer, 2004, pp. 1338–1349.

[28] M. Masud, A. Nayak, M. Zaman, and N. Bansal, "Strategy for mutation testing using genetic algorithms," in *Canadian Conference on Electrical and Computer Engineering.* IEEE, 2005, pp. 1049–1052.

[29] K. Ayari, S. Bouktif, and G. Antoniol, "Automatic mutation test input data generation via ant colony," in *Proceedings of the 9th annual conference on Genetic and evolutionary computation.* ACM, 2007, pp. 1074–1081.

[30] Y. Jia and M. Harman, "Constructing subtle faults using higher order mutation testing," in *Eighth IEEE International Working Conference on Source Code Analysis and Manipulation.* IEEE, 2008, pp. 249–258.

[31] K. Mishra, S. Tiwari, A. Kumar, and A. Misra, "An approach for mutation testing using elitist genetic algorithm," in *3rd IEEE International Conference on Computer Science and Information Technology (ICCSIT)*, Vol. 5. IEEE, 2010, pp. 426–429.

[32] B. Schwarz, D. Schuler, and A. Zeller, "Breeding high-impact mutations," in *IEEE Fourth International Conference on Software Testing, Verification and Validation Workshops (ICSTW).* IEEE, 2011, pp. 382–387.

[33] M. Harman, Y. Jia, and W.B. Langdon, "Strong higher order mutation-based test data generation," in *Proceedings of the 19th ACM SIGSOFT Symposium and the 13th European Conference on Foundations of Software Engineering.* ACM, 2011, pp. 212–222.

[34] J.J. Domínguez-Jiménez, A. Estero-Botaro, A. García-Domínguez, and I. Medina-Bulo, "Evolutionary mutation testing," *Information and*

Software Technology, Vol. 53, No. 10, 2011, pp. 1108–1123.

[35] G. Fraser and A. Zeller, "Mutation-driven generation of unit tests and oracles," *IEEE Transactions on Software Engineering*, Vol. 38, No. 2, 2012, pp. 278–292.

[36] A.A.L. de Oliveira, C.G. Camilo-Junior, and A.M. Vincenzi, "A coevolutionary algorithm to automatic test case selection and mutant in mutation testing," in *IEEE Congress on Evolutionary Computation.* IEEE, 2013, pp. 829–836.

[37] M.B. Bashir and A. Nadeem, "A fitness function for evolutionary mutation testing of object-oriented programs," in *9th International Conference on Emerging Technologies (ICET).* IEEE, 2013, pp. 1–6.

[38] P.S. Yiasemis and A.S. Andreou, "Locating and correcting software faults in executable code slices via evolutionary mutation testing," in *International Conference on Enterprise Information Systems.* Springer, 2012, pp. 207–227.

[39] E. Omar, S. Ghosh, and D. Whitley, "Comparing search techniques for finding subtle higher order mutants," in *Proceedings of the Annual Conference on Genetic and Evolutionary Computation.* ACM, 2014, pp. 1271–1278.

[40] E. Omar, S. Ghosh, and D. Whitley, "Constructing subtle higher order mutants for Java and AspectJ programs," in *IEEE 24th International Symposium on Software Reliability Engineering (ISSRE).* IEEE, 2013, pp. 340–349.

[41] Y.M.B. Ali and F. Benmaiza, "Generating test case for object-oriented software using genetic algorithm and mutation testing method," *International Journal of Applied Metaheuristic Computing (IJAMC)*, Vol. 3, No. 1, 2012, pp. 15–23.

[42] A.S. Banzi, T. Nobre, G.B. Pinheiro, J.C.G. Árias, A. Pozo, and S.R. Vergilio, "Selecting mutation operators with a multiobjective approach," *Expert Systems with Applications*, Vol. 39, No. 15, 2012, pp. 12 131–12 142.

[43] B. Baudry, V. Le Hanh, J.M. Jézéquel, and Y. Le Traon, "Building trust into oo components using a genetic analogy," in *11th International Symposium on Software Reliability Engineering.* IEEE, 2000, pp. 4–14.

[44] S. Subramanian and R. Natarajan, "A tool for generation and minimization of test suite by mutant gene algorithm," *Journal of Computer Sciences*, Vol. 7, No. 10, 2011, pp. 1581–1589.

[45] K.T. Le Thi My Hanh and N.T.B. Tung, "Mutation-based test data generation for simulink models using genetic algorithm and simulated annealing," *International Journal of Computer and Information Technology*, Vol. 3, No. 04, 2014, pp. 763–771.

[46] N.T. Binh, K.T. Tung *et al.*, "A novel test data generation approach based upon mutation testing by using artificial immune system for Simulink models," in *Knowledge and Systems Engineering.* Springer, 2015, pp. 169–181.

[47] L.T.M. Hanh, N.T. Binh, and K.T. Tung, "Applying the meta-heuristic algorithms for mutation-based test data generation for Simulink models," in *Proceedings of the Fifth Symposium on Information and Communication Technology.* ACM, 2014, pp. 102–109.

[48] B.N. Thanh and T.K. Thanh, "Survey on mutation-based test data generation," *International Journal of Electrical and Computer Engineering*, Vol. 5, No. 5, 2015.

[49] M. Patrick, "Metaheuristic optimisation and mutation-driven test data generation," in *Computational Intelligence and Quantitative Software Engineering.* Springer, 2016, pp. 89–115.

[50] F. Popentiu-Vladicescu and G. Albeanu, "Nature-inspired approaches in software faults identification and debugging," *Procedia Computer Science*, Vol. 92, 2016, pp. 6–12.

[51] M. Dave and R. Agrawal, "Search based techniques and mutation analysis in automatic test case generation: A survey," in *IEEE International Advance Computing Conference (IACC).* IEEE, 2015, pp. 795–799.

[52] Y. Jia, F. Wu, M. Harman, and J. Krinke, "Genetic improvement using higher order mutation," in *Proceedings of the Companion Publication of the Annual Conference on Genetic and Evolutionary Computation.* ACM, 2015, pp. 803–804.

[53] Q.V. Nguyen and L. Madeyski, "Searching for strongly subsuming higher order mutants by applying multi-objective optimization algorithm," in *Advanced Computational Methods for Knowledge Engineering.* Springer, 2015, pp. 391–402.

[54] Q.V. Nguyen and L. Madeyski, "Higher order mutation testing to drive development of new test cases: An empirical comparison of three strategies," in *Asian Conference on Intelligent Information and Database Systems.* Springer, 2016, pp. 235–244.

[55] Q.V. Nguyen and L. Madeyski, "Empirical evaluation of multiobjective optimization algorithms searching for higher order mutants," *Cybernetics and Systems*, 2016.

[56] F. Wu, M. Harman, Y. Jia, and J. Krinke, "HOMI: Searching higher order mutants for software improvement," in *International Symposium*

on Search Based Software Engineering. Springer, 2016, pp. 18–33.

[57] A. Estero-Botaro, A. García-Domínguez, J.J. Domínguez-Jiménez, F. Palomo-Lozano, and I. Medina-Bulo, "A framework for genetic test-case generation for WS-BPEL compositions," in *Testing Software and Systems*, ser. Lecture Notes in Computer Science, vol 8763, M. Merayo and E. de Oca, Eds. Berlin, Heidelberg: Springer, 2014, pp. 1–16.

[58] C.P. Rao and P. Govindarajulu, "Genetic algorithm for automatic generation of representative test suite for mutation testing," *International Journal of Computer Science and Network Security (IJCSNS)*, Vol. 15, No. 2, 2015, p. 11.

[59] S. Rani and B. Suri, "An approach for test data generation based on genetic algorithm and delete mutation operators," in *Second International Conference on Advances in Computing and Communication Engineering (ICACCE).* IEEE, 2015, pp. 714–718.

[60] F.C.M. Souza, M. Papadakis, Y. Le Traon, and M.E. Delamaro, "Strong mutation-based test data generation using hill climbing," in *Proceedings of the 9th International Workshop on Search-Based Software Testing.* ACM, 2016, pp. 45–54.

[61] N. Jatana, B. Suri, S. Misra, P. Kumar, and A.R. Choudhury, "Particle swarm based evolution and generation of test data using mutation testing," in *International Conference on Computational Science and its Applications.* Springer, 2016, pp. 585–594.

[62] N.T. Binh, K.T. Tung *et al.*, "A novel fitness function of metaheuristic algorithms for test data generation for Simulink models based on mutation analysis," *Journal of Systems and Software*, Vol. 120, 2016, pp. 17–30.

[63] N. Jatana, B. Suri, P. Kumar, and B. Wadhwa, "Test suite reduction by mutation testing mapped to set cover problem," in *Proceedings of the Second International Conference on Information and Communication Technology for Competitive Strategies.* ACM, 2016, p. 36.

[64] P. McMinn, M. Harman, K. Lakhotia, Y. Hassoun, and J. Wegener, "Input domain reduction through irrelevant variable removal and its effect on local, global, and hybrid search-based structural test data generation," *IEEE Transactions on Software Engineering*, Vol. 38, No. 2, 2012, pp. 453–477.

[65] M. Dorigo and G.D. Caro, "New ideas in optimization," D. Corne, M. Dorigo, F. Glover, D. Dasgupta, P. Moscato, R. Poli, and K.V. Price, Eds. McGraw-Hill Ltd., 1999, ch. The ant colony optimization meta-heuristic, pp. 11–32.

[66] M. Dorigo, G. Di Caro, and L.M. Gambardella, "Ant algorithms for discrete optimization," *Artificial life*, Vol. 5, No. 2, 1999, pp. 137–172.

[67] M. Dorigo, "Optimization, learning and natural algorithms," Ph.D. dissertation, Politecnico di Milano, 1992.

[68] T. Stützle, M. López-Ibáñez, and M. Dorigo, "A concise overview of applications of ant colony optimization," *Wiley Encyclopedia of Operations Research and Management Science*, 2011.

[69] B. Suri and S. Singhal, "Literature survey of ant colony optimization in software testing," in *CSI Sixth International Conference on Software Engineering (CONSEG).* IEEE, 2012, pp. 1–7.

[70] A.S. Fraser, "Simulation of genetic systems by automatic digital computers VI. Epistasis," *Australian Journal of Biological Sciences*, Vol. 13, No. 2, 1960, pp. 150–162.

[71] H.J. Bremermann, *The evolution of intelligence: The nervous system as a model of its environment.* University of Washington, Department of Mathematics, 1958.

[72] J.H. Holland, *Adaptation in natural and artificial systems: an introductory analysis with applications to biology, control, and artificial intelligence.* U Michigan Press, 1975.

[73] C. Sharma, S. Sabharwal, and R. Sibal, "A survey on software testing techniques using genetic algorithm," *CoRR*, 2014. [Online]. https://arxiv.org/abs/1411.1154

[74] E. Omar and S. Ghosh, "An exploratory study of higher order mutation testing in aspect-oriented programming," in *IEEE 23rd International Symposium on Software Reliability Engineering.* IEEE, 2012, pp. 1–10.

[75] B. Baudry, F. Fleurey, J.M. Jézéquel, and Y. Le Traon, "Automatic test case optimization using a bacteriological adaptation model: Application to .NET components," in *17th IEEE International Conference on Automated Software Engineering.* IEEE, 2002, pp. 253–256.

[76] B. Baudry, F. Fleurey, J.M. Jézéquel, and Y. Le Traon, "From genetic to bacteriological algorithms for mutation-based testing," *Software Testing, Verification and Reliability*, Vol. 15, No. 2, 2005, pp. 73–96.

[77] P. May, J. Timmis, and K. Mander, "Immune and evolutionary approaches to software mutation testing," in *Artificial Immune Systems*, ser. Lecture Notes in Computer Science, vol 4628, L. de Castro, F. Von Zuben, and H. Knidel, Eds. Berlin, Heidelberg: Springer, 2007, pp. 336–347.

[78] P. May, K. Mander, and J. Timmis, "Mutation testing: An artificial immune system approach," in *UK-Softest. UK Software Testing Workshop*. Citeseer, 2003.

[79] M. Papadakis and N. Malevris, "Automatically performing weak mutation with the aid of symbolic execution, concolic testing and search-based testing," *Software Quality Journal*, Vol. 19, No. 4, 2011, pp. 691–723.

[80] Y. Jia and M. Harman, "Higher order mutation testing," *Information and Software Technology*, Vol. 51, No. 10, 2009, pp. 1379–1393.

[81] Y. Jia, "Higher order mutation testing," Ph.D. dissertation, University College London, 2013. [Online]. http://discovery.ucl.ac.uk/1401264/1/YuePhDFinal2013.pdf

[82] Q.V. Nguyen and L. Madeyski, "Problems of mutation testing and higher order mutation testing," in *Advanced computational methods for knowledge engineering*. Springer, 2014, pp. 157–172.

[83] M. Harman, Y. Jia, and Y. Zhang, "Achievements, open problems and challenges for search based software testing," in *IEEE 8th International Conference on Software Testing, Verification and Validation (ICST)*. IEEE, 2015, pp. 1–12.

e-Informatica Software Engineering Journal, Volume 11, Issue 1, 2017, pages: 77–102, DOI 10.5277/e-Inf170104

Efficiency of Software Testing Techniques: A Controlled Experiment Replication and Network Meta-analysis

Omar S. Gómez[*], Karen Cortés-Verdín[**], César J. Pardo[***]

[*]*Facultad de Informática y Electrónica, Escuela Superior Politécnica de Chimborazo*
[**]*Facultad de Estadística e Informática, Universidad Veracruzana*
[***]*Electronic and Telecommunications Engineering Faculty, Information Technology Research Group (GTI), Universidad del Cauca*

ogomez@espoch.edu.ec, kcortes@uv.mx, cpardo@unicauca.edu.co

Abstract

Background. Common approaches to software verification include static testing techniques, such as code reading, and dynamic testing techniques, such as black-box and white-box testing. **Objective.** With the aim of gaining a better understanding of software testing techniques, a controlled experiment replication and the synthesis of previous experiments which examine the efficiency of code reading, black-box and white-box testing techniques were conducted. **Method.** The replication reported here is composed of four experiments in which instrumented programs were used. Participants randomly applied one of the techniques to one of the instrumented programs. The outcomes were synthesized with seven experiments using the method of network meta-analysis (NMA). **Results.** No significant differences in the efficiency of the techniques were observed. However, it was discovered the instrumented programs had a significant effect on the efficiency. The NMA results suggest that the black-box and white-box techniques behave alike; and the efficiency of code reading seems to be sensitive to other factors. **Conclusion.** Taking into account these findings, the Authors suggest that prior to carrying out software verification activities, software engineers should have a clear understanding of the software product to be verified; they can apply either black-box or white-box testing techniques as they yield similar defect detection rates.

Keywords: software verification, software testing, controlled experiment, experiment replication, meta-analysis, network meta-analysis, quantitative synthesis

1. Introduction

Currently, due to the increase in both the size and complexity of software products, verification plays an important role in the software product development (or maintenance) process. The aim of software verification is to enssure that a software product fully satisfies all the requirements defined by the customer. It typically includes such activities as code executions, reviews, walkthroughs and inspections of the artifacts produced in the development or maintenance process.

Software verification is performed at different phases of the software development (or maintenance) process by following two approaches: reviewing or inspecting artifacts, such as documents and a source code (static approach) or an executing code (dynamic approach).

In the software construction phase, common techniques used in software verification include code reading (static approach), black-box and white-box testing (dynamic approach), and various other techniques, such as regression testing [1].

With the aim of gaining a better understanding of various software testing techniques applied during the software construction phase, in this work, the authors pursue two goals: 1) running a controlled experiment replication on the efficiency of testing techniques expressed in terms of the number of defects detected per hour by each of the techniques: code reading, black-box and white-box (this carried out through the application of an experimental paradigm [2–4]), and 2) carrying out a synthesis of existing experiments which also address the efficiency of the related testing techniques.

Our replication is the extension of previous experiments reported in [5–12], where the effectiveness of the aforementioned software testing techniques was the main issue examined. In these experiments, the effectiveness was measured either as the percentage or as the number of defects observed in these testing techniques. Complementary to effectiveness, efficiency is another aspect that deserves attention. Due to the limitations of time and resources it is often raised in the software verification phase, it is worth considering which of the testing techniques behave in an optimal way (e.g. the fastest technique detecting defects). The authors have found some controlled experiments that also address the efficiency of the testing techniques [5–8, 11].

In order to corroborate the previous findings and also generate new knowledge with regard to the study of software testing techniques efficiency, this work reports the findings of a controlled experiment replication that examines efficiency in terms of the number of defects detected per hour of the following testing techniques: code reading, black-box and white-box testing. The replication results are then incorporated to existing related experiments following a quantitative synthesis approach. According to [13], this experiment can be considered as a conceptual replication of the original experiment reported in [5], only the constructs are maintained; these are the three testing techniques (causal constructs) and the efficiency (effect construct).

In science, replication is a key mechanism which allows for the verification of previous findings and for the consolidation of the body of knowledge [14, 15]. Replication is still a pending issue to be addressed in Software Engineering, since there is evidence showing a minimal amount of controlled experiments that have been replicated [16, 17]. If an experiment is not replicated or verified, there is no way to distinguish whether its outcome was produced by chance, artificially or it conforms to a reality. The results of this replication serve as a mechanism for verification, and they also contribute to the consolidation of the body of knowledge in the software verification research area. Although a number of experiments related to our replication have been conducted, it is worth to note that increasing the number of related experiments (experiments family) will allow other researchers to apply quantitative synthesis methods in a more confident way, the synthesis outcome will be strengthened by the pooled samples sizes of the related experiments.

The rest of the document is organized as follows. In Section 2, the related work is presented. In Section 3, the baseline experiment of the presented replication is described. In Section 4, the studied software testing techniques are studied. Sections 5 presents the context of our experiment replication. In Section 6,there is the statistics used for analysis and the results obtained. In Section 7, a quantitative synthesis using the obtained results and the results from related experiments is carried out. In Section 8, the findings are discussed and finally in Section 9 the conclusions are presented.

2. Related work

This section presents the summary of the empirical studies (family or series of experiments) related to the experiment replication reported here. The authors considered the controlled experiment reported in [5] as the baseline for their experiments. The aim of this experiment is to examine the effectiveness, efficiency and cost of three software testing techniques: black-box by equivalence class partitioning and boundary value analysis, white-box by sentence coverage and code reading by stepwise abstraction.

The authors of [5] carried out two replications of their experiment. Years later, the authors of [6, 7] performed the other two replications. A few years later, the authors in [8] conducted another replication. The authors of [9, 10] also carried out several replications. Recently, the authors of [11] and [12] replicated the experiment as well.

Note that these replications do not take as reference the same baseline experiment. For example, the second and third replication reported in [5] takes as baseline their first experiment. The experiment reported in [6, 7] is the replication of [5]. In the case of the experiment reported in [8], the authors used the experiment replication package of [6, 7], thus considering this experiment the replication of [6, 7]. Experiments of the authors of [9, 10] are based on the replication packages of [6, 7] and [8]. With regard to the experiment in [11], it is related to the replication package of [6, 7]. In the case of the experiment reported in [12], the authors adapted the replication package of [9,10]. Table 1 presents some characteristics of these experiments.

2.1. Constructs and operationalizations studied

2.1.1. Cause constructs and operationalizations

The cause constructs examined in these experiments are: the black-box [5–12], white-box [5–12] and code reading [5–11] techniques. Regarding black-box, it was operationalized either as equivalence class partitioning and boundary value analysis [5–8, 11] or as equivalence class partitioning [9, 10, 12]. Concerning white-box, it was operationalized either as sentence coverage [5] or as branch coverage [6–12]. In the case of code reading, in all the experiments [5–12] it was operationalized by the use of stepwise abstractions approach [18]. Secondary cause constructs, also examined, are the instrumented program (software type) [5–12], the participant expertise [5], the defect type [9, 10] and the version of the instrumented programs [9, 10].

2.1.2. Effect constructs and operationalizations

The effect constructs examined in these experiments are: effectivenesses [5–12], efficiency [5–8, 11], fault visibility [9] and cost [5–7, 11].

The **effectiveness** construct was operationalized as the number of observed defects [5, 8], the percentage of observed defects [5–7, 11, 12], the number of observable defects [5], the percentage of observable defects [5, 12], the percentage of participants who detect a given defect for each defect in the instrumented program [9, 10], the percentage of participants that are able to generate a test case that uncovers the failure associated with a given defect [9, 10], the number of isolated defects [8], and the percentage of isolated defects [6, 7, 11]. **Efficiency** was operationalized as the number of defects detected per hour (detection rate) [5–8, 11], and as the number of defects isolated per hour (isolation rate) [6, 7]. Finally **cost** was operationalized as the time spent applying the testing techniques [5–7, 11], defect isolation time [6, 7, 11], cpu-time [5], connect time [5] and number of programs runs [5].

2.2. Findings

In this section some relevant findings of these experiments are presented. These findings are organized according to the different effect constructs examined.

2.2.1. Effectiveness

Number of observed defects (operationalization o1.1). For the umd82 experiment [5], either code reading or black-box were significantly more effective than white-box. Concerning umd83 experiment [5], no significant differences were observed between the three testing techniques. In the case of umd84 [5], code reading was significantly more effective than black-box and white-box, also black-box was significantly more effective than white-box. In the case of uos97 [8], the authors observed a significant difference in the effectiveness of the techniques, however, it is not described which of the pairwise techniques was significantly different. It seems that black-box and white-box behave in a similar way and that these techniques are more effective than code reading. With regard to the studied secondary factors and interaction effects:

– Software type (instrumented programs). The effectiveness of the techniques (measured as the number of observed defects) was signifi-

Table 1. Characteristics of the aforementioned family of experiments

Experiment	Participants	Programs and number of defects	Language	Country
umd82 [5]	CS (under)graduates	p1(9), p2(6), p3(7)	Simplt	USA
umd83 [5]	CS (under)graduates	p1(9), p2(6), p4(12)	Simplt	USA
umd84 [5]	Professionals	p1(9), p3(7), p4(12)	Fortran	USA
ukl94 [6, 7]	CS undergraduates	nt(11),cm(14),na(11)	C	Germany
ukl95 [6, 7]	CS undergraduates	nt(6), cm(9), na(7)	C	Germany
uos97 [8]	CS undergraduates	nt(8), cm(9), na(8)	C	UK
upm00 [9]	CS undergraduates	nt(9), cm(9), na(9), tr(9)	C	Spain
upm01 [9, 10]	CS undergraduates	nt(7), cm(7), na(7)	C	Spain
upm02 [10]	CS undergraduates	nt(7), cm(7), na(7)	C	Spain
upm03 [10]	CS undergraduates	nt(7), cm(7), na(7)	C	Spain
upm04 [10]	CS undergraduates	nt(7), cm(7), na(7)	C	Spain
upm05 [10]	CS undergraduates	nt(7), cm(7), na(7)	C	Spain
uds05 [10]	CS undergraduates	nt(7), cm(7), na(7)	C	Spain
upv05 [10]	CS undergraduates	nt(7), cm(7), na(7)	C	Spain
ort05 [10]	CS undergraduates	cm(7), na(7)	C	Uruguay
uok11 [11]	CS graduates	nt(8), cm(9), na(8)	C	India
uady13 [12]	CS undergraduates	nt(7), cm(7)	C	Mexico

cantly affected by the instrumented programs used in umd84 [5] and uos97 [8]. On the other hand, the effectiveness is not affected by software type in umd82 and umd83 [5].

– Expertise. The effectiveness (in terms of the number of observed defects) was significantly affected by the expertise, advanced expertise participants detected more defects than either intermediates or juniors (umd84 [5]).
– Interaction effects. In umd83 [5] and uos97 [8], the authors report a significant interaction effect between the testing techniques and the instrumented programs. A three-way interaction between techniques, programs and expertise was observed in umd84 [5].

Percentage of observed defects (o1.2). Either code reading or black-box were significantly more effective than white-box (in umd82 [5]). Code reading was significantly more effective than black-box and white-box, and also black-box was significantly more effective than white-box (in umd84 [5]). There are no significant differences between the testing techniques (umd83 [5], ukl94, ukl95 [6, 7], uok11 [11] and uady13 [12]). In the case of secondary factors and interaction effects:

– Software type. The effectiveness of the techniques (measured as the percentage of observed defects) was significantly affected by the instrumented programs in umd82, umd83, umd84 [5], ukl94 [6, 7] and uok11 [11].
– Expertise. The effectiveness significantly varies with regard to the level of expertise (in umd84 [5]). The percentage of observed defects was significantly higher for the participants with advanced expertise, this difference is significant only with respect to juniors. There were not significant differences between intermediates and juniors in umd82, umd83, umd84 [5].
– Interaction effects. In umd83 [5] an interaction effect between the testing techniques and the instrumented programs was observed. A three-way interaction between techniques, programs and expertise was observed in umd84 [5].

Number of observable defects (o1.3). In the case of umd82 [5], the number of observable defects was significantly higher for black-box (in comparison to white-box). Significant differences were not found in umd84 [5].

Percentage of observable defects (o1.4). The percentage of observable defects is significantly higher for black-box than for white-box in umd82 [5]. Significant differences were not found in umd84 [5] and uady13 [12].

Percentage of participants who detect a given defect for each defect in the instru-

mented program (o1.5). The effectiveness is affected by the testing techniques. Code reading is significantly less effective than black-box and white-box, and black-box and white-box behave in a similar way (upm00 [9], upm01, upm02, upm03, upm04, upm05 and uds05 [10]). Concerning secondary factors and interaction effects:

– Software type. The effectiveness of the techniques was significantly affected by the instrumented programs in upm00 [9].
– Defect type. In upm00 [9], the effectiveness of the techniques was significantly affected by the defect types injected in the instrumented programs.
– Interaction effects. In upm00 [9] an interaction effect between the testing techniques and the instrumented programs was observed. Also an interaction effect between the instrumented programs and the defect types was observed.

Percentage of participants that are able to generate a test case that uncovers the failure associated with a given defect (o1.6). The effectiveness did not impact black-box and white-box (upm01, upm02, upm03, upm04, upm05, uds05 and upv05 [9, 10]). Black-box is significantly more effective than white-box (ort05 [10]). In the case of secondary factors and interaction effects:

– Software type. The effectiveness of the techniques was significantly affected by the instrumented programs in upm01, upm05, uds05, upv05 and ort05 [10].
– Defect type. In upm04, uds05, upv05 and ort05 [10], the effectiveness of the techniques was significantly affected by the defect types injected in the instrumented programs.
– Program version. The version of the instrumented programs was not affected in upm01, upm02, upm03, upm04, upm05, uds05, upv05 and ort05 [10].
– Interaction effects. In upm00, upm01, upm02, upm03, upm04, upm05, upv05 and ort05 [9, 10] an interaction effect between the testing techniques and the instrumented programs was observed. An interaction effect between the instrumented programs and the defect types was observed in upm00, upm02, upm04, upm05, uds05, upv05 and ort05 [9,10]. An interaction effect between techniques and defect types was observed in upm01, upm03, upm05 and uds05 [10]. An interaction effect between program version and defect types was observed in upm03, upv05 [10]. Another interaction effect between the technique and the program version was observed in uds05 and upv05 [10]. Three-way interactions between instrumented programs, techniques and defect types, and also between instrumented programs, program versions and defect types were observed in ort05 [10].

Number of isolated defects (o1.7). Although some information about this is presented in uos97 [8] neither descriptive nor inferential analysis is discussed.

Percentage of isolated defects (o1.8). The effectiveness of the testing techniques behaves in a similar way (ukl94 [6, 7] and uok11 [11]). The percentage of isolated defects is significantly affected by the testing techniques in ukl95 [6, 7], although a post-hoc is missing, it seems that black-box and code reading show better effectiveness than white-box. The findings for secondary factors and interaction effects are:

– Software type. The effectiveness of the techniques was significantly affected by the instrumented programs (in ukl94 [6, 7] and uok11 [11]).
– Technique application order (sequence). The effectiveness of the techniques is significantly affected by the order in which techniques are applied (in ukl94 [6, 7]).

Summarizing. It can be observed that the effectiveness construct has the greatest number of operationalizations. It was operationalized in several ways. It can also be seen that secondary factors such as instrumented programs and expertise may have an impact on the techniques effectiveness. It is not so clear which of the techniques is more effective due to contradictory findings.

2.2.2. Efficiency

Defects detected per hour (o2.1). The three testing techniques showed similar defect detection rates in umd82, umd83 [5] and uok11 [11].

Code reading showed the higher defect detection rate in comparison to either black-box or white-box (umd84 [5]), this difference was significant. The authors of ukl94 and ukl95 experiments [6,7] report a significant difference between the techniques, however, a post-hoc analysis did not show the pairwise significant differences, it seems that black-box shows the higher defect detection rate. In the case of uos97 [8], the authors did not report the inferential statistics for this metric, however, black-box seems to yield the higher defect detection rate, white-box appears to be the second most efficient technique. The findings from secondary factors and interaction effects are:

– Software type. The efficiency of the techniques (measured as the number of defects detected per hour) was significantly affected by the instrumented programs in umd82, umd84 [5] and uok11 [11].
– Expertise. The efficiency did not vary with regard to the level of expertise (umd83, umd84 [5]). Intermediate participants detected defects at a significantly faster rate than juniors did (umd82 [5]).
– Technique application order (sequence). The efficiency of the techniques is significantly affected by the order in which they are applied (in ukl95 [6,7]).
– Interaction effects. A two-way interaction between techniques and instrumented programs was observed in umd84 [5].

Defects isolated per hour (o2.2). The three techniques behave in a similar way (ukl94 [6,7]). However, in the case of ukl95 [6,7] and uok11 [11], the defect isolation rate is significantly affected by the techniques, although a post-hoc analysis is missing, in ukl95 [6,7] it seems that black-box shows a higher defect isolation rate. In the case of uok11 [11] it seems that white-box and black-box show higher defect isolation rates than code reading. With regard to secondary factors:

– Technique application order (sequence). The defect isolation rate is significantly affected by the order in which techniques are applied (ukl94 [6,7]).

Summarizing. Similar findings can be observed for the efficiency construct, secondary factors, such as instrumented programs, expertise and the technique application order, may have an impact on the techniques efficiency. At first sight, it is hard to conclude which of the techniques is more efficient due to some contradictory findings.

2.2.3. Cost

Time spent applying the testing techniques (o3.1). The time spent applying the three testing techniques is similar (in umd83, umd84 [5] and uok11 [11]). Applying white-box requires significantly more time than applying either code reading or black-box (umd82 [5]). Although a significant difference was observed in ukl94 and ukl95 [6,7], the authors did not present a post-hoc analysis to assess which of the techniques requires significantly less time, however, it seems that applying code-reading requires more time than applying white-box; black-box requires less time than white-box (ukl94, ukl95 [6,7]). In the case of secondary factors and interaction effects:

– Software type. The time spent applying the techniques was significantly affected by the instrumented programs in umd82, umd84 [5] and uok11 [11].
– Expertise. The time spent applying the techniques did not vary with regard to the level of expertise (umd82, umd83, umd84 [5]).
– Technique application order (sequence). The time spent applying the techniques is significantly affected by the order in which they are applied (in ukl95 [6,7]).
– Interaction effects. A two-way interaction between techniques and instrumented programs was observed in umd84 [5].

Defect isolation time (o3.2). The experiments in ukl94, ukl95 [6,7] and uok11 [11] report a significant difference between the techniques, however, a post-hoc analysis does not identify pairwise significant differences. Code reading seems to require less time for isolating defects than the other techniques.

Cpu-time (o3.3). Black-box required significantly more cpu-time than white-box (in umd84 [5]).

Connect time (o3.4). Participants applying black-box black-box spent significantly more minutes of connect time than those applying white-box (in umd84 [5]).
Number of program runs (o3.5). This metric did not show significant differences between black-box and white-box (in umd84 [5]).

Summarizing. Secondary factors, such as instrumented programs, expertise and the technique application order, may have an impact on the cost of applying the testing techniques. With regard to the the application time of these techniques, it is hard to identify which of the testing techniques incurs fewer costs. However, code reading seems to require less time for isolating defects. Concerning cpu-time and connect time, black-box seems to demand more resources.

To conclude this section, Table 2 shows the global summary of the findings found in this family of experiments.

3. Baseline experiment

Following the proposed guidelines for reporting experiment replications [19], this section describes the original experiment. In [5], the authors report results from three controlled experiments which were conducted as controlled experiments where different types of participants (undergraduate, graduate students and practitioners) applied three software testing techniques (code reading, black-box testing and white-box testing) to four instrumented programs.

The participants in these experiments were representative of three levels of computer science expertise: junior (0–2 years of experience), intermediate (2.5–6.2 years of experience) and advanced (10 years of experience). A total of 29, 13 and 32 people participated in three respective experiments. In the first two experiments, the participants were either upper-level computer science majors or graduate students. In the third experiment, the participants were programming professionals from NASA and the Computer Sciences Corporation.

The instrumented programs used in these experiments were coded in Fortran and Simpl-T. The four programs are related to a text processor (p1), a mathematical plotting routine (p2), a numeric abstract data type (p3) and a database maintainer program (p4). Table 3 shows some characteristics of the used programs, such as source lines of code (SLOC), cyclomatic complexity (VG) and the number of defects injected.

It is worth noting that the authors did not use all the programs in the three experiments. Programs p1, p2 and p3 were used in the first experiment; programs p1, p2 and p4 were used in the second experiment, and programs p1, p3, and p4 were used in the third one.

The testing techniques examined in [5] were code reading by stepwise abstraction [18], black-box testing through equivalence partitioning and boundary value analysis [20, 21] and white-box testing through statement coverage [21, 22]. Table 4 shows the efficiency observed (in terms of defects detected per hour) in the experiments and their standard deviations. The authors only report a significant difference (at $\alpha < 0.0003$) in the third experiment. This difference shows an enhanced efficiency for the code reading technique.

Regarding the defect detection rates in the instrumented programs used in the experiments, Table 5 shows the defect detection rates per program and their standard deviations. The authors report a significant difference in the first (at $\alpha < 0.01$) and third experiment (at $\alpha < 0.0001$). In both experiments, the testing techniques showed higher levels of efficiency in program p3 (Data type).

The authors also examined the efficiency of the participants according to their differing levels of expertise: junior, intermediate and advanced. Table 6 shows the efficiency rates of these types of participants and their standard deviations.

The athors report a significant difference only in the first experiment. Intermediate participants detected defects at a faster rate than junior participants. In the remaining experiments, the authors did not observe any significant difference in defect detection rates between expertise levels.

Table 2. Global summary of findings of this family

	umd82	umd83	umd84	ukl94	ukl95	uos97	upm00	upm01	upm02	upm03	upm04	upm05	uds05	upv05	ort05	uok11	uady13
								EFFECTIVENESS									
o1.1	(✠ ≡ ■) > □	✠ ≡ ■ ≡ □ I:sw	✠ > ■ > □ SF:sw,exp I:sw,exp			(■ ≡ □) > ✠ SF:sw I:sw											
o1.2	(✠ ≡ ■) > □ SF:sw	✠ ≡ ■ ≡ □ SF:sw I:sw	✠ > ■ > □ SF:sw,exp I:sw, exp	✠ ≡ ■ ≡ □ SF:sw	✠ ≡ ■ ≡ □											✠ ≡ ■ ≡ □ SF:sw	✠ ≡ ■ ≡ □
o1.3	■ > □		■ ≡ □														
o1.4	■ > □		■ ≡ □														■ ≡ □
o1.5							(■ ≡ □) > ✠ SF:sw,dt I:sw,dt	(■ ≡ □) > ✠	(■ ≡ □) > ✠	(■ ≡ □) > ✠	(■ ≡ □) > ✠	(■ ≡ □) > ✠					
o1.6								■ ≡ □ SF:sw I:sw	■ ≡ □ I:sw,dt	■ ≡ □ I:sw,v	■ ≡ □ SF:dt I:sw,dt	■ ≡ □ SF:sw I:sw,dt	■ ≡ □ SF:sw,dt I:sw,dt,v	■ ≡ □ SF:sw,dt I:sw,dt,v	■ > □ SF:sw,dt I:sw,dt,v		
o1.8				✠ ≡ ■ ≡ □ SF:sw I:seq	(✠ ≡ ■) > □ SF:sw											✠ ≡ ■ ≡ □ SF:sw	
								EFFICIENCY									
	umd82	umd83	umd84	ukl94	ukl95	uos97	upm00	upm01	upm02	upm03	upm04	upm05	uds05	upv05	ort05	uok11	uady13
o2.1	✠ ≡ ■ ≡ □ SF:sw,exp	✠ ≡ ■ ≡ □	✠ > (■ ≡ □) SF:sw I:sw	✠ ≡ ■ ≡ □	✠ ≡ ■ ≡ □ SF:seq	■ > □ > ✠										✠ ≡ ■ ≡ □ SF:sw	
o2.2				✠ ≡ ■ ≡ □ SF:seq	■ > ✠ > □											(■ ≡ □) > ✠	
								COST									
	umd82	umd83	umd84	ukl94	ukl95	uos97	upm00	upm01	upm02	upm03	upm04	upm05	uds05	upv05	ort05	uok11	uady13
o3.1	□ > (✠ ≡ ■) SF:sw	✠ ≡ ■ ≡ □	✠ ≡ ■ ≡ □ SF:sw I:sw	✠ > □ > ■	SF:seq											✠ ≡ ■ ≡ □ SF:sw	
o3.2				■ > □ > ✠	■ > □ > ✠											■ > □ > ✠	
o3.3			■ > □														
o3.4			■ > □														
o3.5			■ ≡ □														

Effectiveness operationalizations. Number of observed defects (**o1.1**); Percentage of observed defects (**o1.2**); Number of observable defects (**o1.3**); Percentage of observable defects (**o1.4**); Percentage of participants who detect a given defect for each defect in the instrumented program (**o1.5**); Percentage of participants that are able to generate a test case that uncovers the failure associated with a given defect (**o1.6**); Percentage of isolated defects (**o1.8**).
Efficiency operationalizations. Defects detected per hour (**o2.1**); Defects isolated per hour (**o2.2**).
Cost operationalizations. Time spent applying the testing technique (**o3.1**); Defect isolation time (**o3.2**); Cpu-time (**o3.3**); Connect time (**o3.4**); Number of programs runs (**o3.5**).
Software testing techniques. Code reading (✠); Black-box (■); White-box (□).
Testing techniques comparisons convention. The technique behaves significantly better (>); The technique behaves in a similar way (≡), non-significant difference.
Secondary factors (SF) studied. These secondary factors showed a significant difference in the ANOVA: Instrumented software program (**sw**); Instrumented software program version (**v**); Defect type (**dt**); Programmer expertise (**exp**); Technique application order (**seq**).
Interactions (I). In this case some secondary factors were involved in a significant interaction with the main factor (software testing technique), or interactions were found between some secondary factors.

Table 3. Characteristics of instrumented programs used in [5]

Program	SLOC	VG	Defects
Formatter (p1)	169	18	9
Plotter (p2)	145	32	6
Data type (p3)	147	18	7
Database (p4)	355	57	12

Table 4. Average and standard deviation of defect detection rates per software testing technique

Technique	umd82 [5]	umd83 [5]	umd84 [5]
Code reading	1.90 (1.83)	0.56 (0.46)	3.33 (3.42)
Black-box	1.58 (0.90)	1.22 (0.91)	1.84 (1.06)
White-box	1.40 (0.87)	1.18 (0.84)	1.82 (1.24)

Table 5. Average and standard deviation of defect detection rates per software program

Program	umd82 [5]	umd83 [5]	umd84 [5]
Formatter (p1)	1.60 (1.39)	0.98 (0.67)	2.15 (1.10)
Plotter (p2)	1.19 (0.83)	0.92 (0.71)	–
Data type (p3)	2.09 (1.42)	–	3.70 (3.26)
Database (p4)	–	1.05 (1.04)	1.14 (0.79)

Table 6. Average and standard deviation of defect detection rates according to level of expertise

Expertise	umd82 [5]	umd83 [5]	umd84 [5]
Junior	1.36 (0.97)	1.00 (0.85)	2.14 (2.48)
Intermediate	2.22 (1.66)	0.96 (0.74)	2.53 (2.48)
Advanced	–	–	2.36 (1.61)

4. Description of the studied software testing techniques

The following subsections summarize the software testing techniques known as code reading, black-box and white-box testing which were used in this experiment replication.

4.1. Code reading

The aim of code reading is to find defects in code documents without executing the code or the software (static approach).

The studied code reading technique is known as stepwise abstraction [18]. In code reading by stepwise abstraction, a software engineer identifies methods (or functions) in the source code, and then he or she abstracts from them the software program functionality. A set of abstractions builds up to other abstractions which represent modules and so forth. This process is followed until a conceptual understanding of the prime abstraction emerges and brings into view an overall picture of the examined code. This abstraction is then compared to the product specification with the aim of finding inconsistencies or defects in the source code.

4.2. Black-box testing

This type of software testing technique is based on the software product specification. Once a software engineer has the specification, he or she starts to design a set of test cases. The software to be verified is seen as a black-box whose behavior is only determined by studying its inputs and examining its outputs. Nevertheless, because examining all the possible inputs is impractical, only a subset of inputs is selected for testing during the software product verification.

The software engineer assumes that the software product to be verified contains a set of inputs that will probably cause the product to fail. As a consequence of introducing these inputs, the product yields outputs which reveals the presence of defects. Because exhaustive testing is impractical, the main goal is to find a set of data inputs whose probability of belonging to the set of inputs that produce a failure in the product is as high as possible [21, 23]. There are strategies for designing test cases to reveal these inputs. Two such strategies are known as: equivalence class partitioning (ECP) and boundary-value analysis (BVA). The authors worked with the ECP approach, where an equivalence class represents a set of valid or invalid states that are defined as input conditions. A typical input condition is a specified numerical value, a range of values, a set of related values (such as categories) or a logical condition.

4.3. White-box testing

It is also known as crystal or transparent testing, the aim of this technique is to design test cases that are able to exhaustively cover the software code, examining all aspects of the structure and logic of the software product. The main idea is to design test cases that execute all code sentences at least once and that also execute all branches of code containing conditions (evaluating both branches by using both true and false expressions) [21,23]. Because examining all paths of the software code can be impractical, various strategies exist for achieving adequate code coverage. Some of these strategies include: statement coverage, decision (or branch) coverage and condition coverage. The authors worked with the branch coverage approach where a set of test cases is designed to ensure that each control structure is executed at least once. To assess this technique, the programs with the Java JCov coverage, a tool which provides a means to measure and analyze dynamic code coverage of Java programs, were instrumented.

5. Experiment replication context

The experiment replication reported here is composed of four comparative studies (controlled experiments) carried out in December 2014 at the Technical School of Chimborazo (ESPOCH) as part of a software verification workshop. The participants were undergraduate students in their last semester of the software systems engineering bachelor degree. According to [24], the participants were categorized as advanced beginners, i.e. students having a working knowledge of the key aspects of software development practice.

The workshop was offered at no cost and it was intended for students in their last semester so as to complement their technical skills with a software verification course. Since the workshop was voluntary and free of charge, coercion was avoided. The participants were told that they could leave the workshop at any moment. Verbal consent was given from all the participants; the main goals of the experiment were explained to the participants and they were told that the experiment was part of a software verification workshop.

A differently instrumented software program was used in each experiment. The program sizes ranged between 253 and 392 SLOC. Programs were coded in the Java programming language. The average cyclomatic complexity (VG) of programs was around 40. Each program had the same type and number of defects injected (6 defects). As reference, the defect classification scheme of [25] was used, it is the same scheme as the one used in the baseline experiment [5] and also in the family discussed in Section 2. However, regarding one of the defect classification schemes, only three defect types (cosmetic, initialization and control) were used instead of the six used in [5] (cosmetic, initialization, control, data, interface and computation). The change was made to have better contol over experimental conditions, and thus havie the same number and defect types. The defects injected in each instrumented program were as follows:
- omission – cosmetic (F1),
- omission – initialization (F2),
- omission – control (F3),
- commission – cosmetic (F4),
- commission – initialization (F5),
- commission – control (F6).

It can be seen that all defect types were equally balanced in each software program, thus there was more experimental control over the instrumented programs.

The same defect counting scheme as the one used here was also applied in [6, 7], a failure is

Table 7. Characteristics of instrumented programs used in the study

Program	SLOC	VG	Defects	Session type	n
Triangle	41	1	1	Training	16
Deviation	184	14	3	Training	15
Banking	253	28	6	Experiment 1	15
Nametbl	392	43	6	Experiment 2	13
Ntree	349	46	6	Experiment 3	13
Cmdline	300	45	6	Experiment 4	12

observed if the participant applying one of the techniques records the deviate behaviour of the instrumented program with regard to its specification. In code reading, an inconsistency (analog to a failure) is observed if the participant records the inconsistency between his or her abstractions and the specification. False positives which are perceived defects reported by participants that are not in fact defects were ignored.

The experiments were run as part of a software verification workshop. This workshop consisted of ten sessions conducted on alternate days, where each session lasted between two and three hours. The first sessions were used to teach the use of the software testing techniques. Two sessions were used for training, where the participants applied the testing techniques to two instrumented programs. Table 7 shows the used program characteristics, the session type and the number of participants per session.

Regarding program functionality, the Triangle software program determines the type of triangle defined given three input values. Deviation calculates the average and standard deviation of n numbers. The banking program implements basic functions for managing bank accounts. Nametbl implements basic functions for managing a table of symbols. Ntree implements functions for managing an N-ary tree. Finally, cmdline implements the basic functionality of a command line program. All the programs were developed and instrumented by a student enrolled in his last year of the software engineering bachelor degree, he was under our supervision during a semester. The following programs were used as reference: nametbl, ntree and cmdline used in [10], these three programs were entirely rewritten to the Java programming language and instrumented with the previously mentioned defects.

5.1. Experiment replication goal

Following the GQM approach [26] this controlled experiment replication was defined as: “Analyze the testing techniques black-box, white-box and code reading for the purpose of comparison with regard to their efficiency (defects detected per hour) from the point of view of the researcher in an academic controlled context using small instrumented Java programs.”

5.2. Research questions

For this controlled experiment replication, the following main research questions were stated:

– **RQ1**. Is efficiency affected by the studied testing techniques?
– **RQ2**. Do instrumented software programs impact the efficiency of the software testing techniques?
– **RQ3**. Does the relationship between techniques and programs affect the efficiency?

With the collected data of this experiment replication it is possible to define a secondary research question linked to a secondary analysis (defect analysis). This secondary question seeks to explore a possible impact on the software testing techniques efficiency and the defect classification schemes used in the instrumented programs. This secondary research question (SRQ1) was defined as follows:

– **SRQ1**. Do defect types (according to used defect classification schemes) impact the efficiency of the studied software testing techniques?

Table 8. Factorial design structure used

Technique/ Program	Exp. 1 Banking (ba)	Exp. 2 Nametbl (na)	Exp. 3 Ntree (nt)	Exp. 4 Cmdline (cm)
Code reading (cr)	cr, ba	cr, na	cr, nt	cr, cm
Black-box (bb)	bb, ba	bb, na	bb, nt	bb, cm
White-box (wb)	wb, ba	wb, na	wb, nt	wb, cm

The efficiency construct is operationalized according to the number of defects detected per hour after applying the testing techniques. To answer the previous research questions, three hypotheses were defined. For RQ1, the null hypothesis is defined as follows: All the testing techniques studied have similar or equal levels of efficiency. For RQ2, the null hypothesis to test is as follows: The type of software program does not affect the efficiency of testing techniques. For RQ3, the null hypothesis is defined as follows: Efficiency is not affected by the relationship between testing techniques and the type of software program. With regard SRQ1 the null hypothesis is defined as: the defect classification schemes used in the instrumented programs do not affect the efficiency of the testing techniques.

5.3. Design and execution

The four experiments constitute a factorial design (3 × 4) with two factors (technique and program), where the factor technique is composed of three levels (code reading, black-box and white-box testing) and the factor program is composed of four levels (banking, nametbl, ntree and cmdline programs). A factorial design allows for the study of several factors and the interactions among them. The factorial design layout for this replication is shown in Table 8. A completely randomized design was used in each experiment. At the beginning of each session, treatments (techniques) were randomly assigned to participants. In each session, every participant applied a testing technique to an instrumented software program.

The experiments were conducted in December 2014 as part of a workshop on software verification at ESPOCH. Participants used a web application for registering information regarding the application of the software testing technique to a given instrumented program. In a non-invasive way, this web application collected the time that participants spent performing the testing techniques. Below, we provide an overview of how each testing technique is applied on an instrumented program during the training and experiment sessions.

Code reading. Participants used code reading by stepwise abstraction [18]. Each participant receives the source code of the software. Then the participant inspects the code and starts to generate abstractions in a natural language. After the participant has constructed the prime abstraction, he or she is provided with the product specification. Then the participant compares his or her abstractions with the product specification and any inconsistencies observed are registered as defects. The time elapsed for carrying out the previous activities is taken into account for computing the number of defects detected per hour.

Black-box. Participants followed the equivalence class partitioning approach. Each participant receives the software product specification and then begins to generate valid and invalid equivalence classes. Next, the participant designs test cases from the equivalence classes defined and registers the expected outputs. The participant then executes the test cases by running the software program and registers the observed output from each test case. The participant then compares the expected outputs to the observed outputs, and any inconsistencies are registered as defects.

White-box. Participants receive the source code and the instrumented software program. Each participant then starts to generate test cases with the aim of achieving 100% branch coverage of the source code. The participant then registers the observed outputs after running the program. For this testing technique, software programs were instrumented with the Java JCov coverage

Table 9. The collected defect detection rate measurements

Technique/ Program	Exp. 1 Banking (ba)	Exp. 2 Nametbl (na)	Exp. 3 Ntree (nt)	Exp. 4 Cmdline (cm)
Code reading (cr)	0, 1.56, 0.45	1.58, 0.67	0, 1.07	0, 0.71
	0, 0	0, 0	1.86, 0.6	0, 0.72
Black-box (bb)	0.55, 2.26, 0.91	0.89, 0	0.43, 1.12, 0	0, 0.5
	1.09, 1.07	0, 0	0.44, 0	0, 0.47
White-box (wb)	1.1, 0.5, 0	1.28, 1.38, 0	1.03, 0	0.47, 0
	1.1, 4.44	1.04, 0.26	0.52, 0	0, 0

tool, so the participants using this technique were able to see the percentage of coverage achieved after each test case execution. Once a participant achieves the maximum coverage level, he or she the gains access to the product specification. Next, the participant registers the expected outputs as defined by the product specification. He or she then compares the observed outputs with the expected outputs, and any inconsistencies are registered as defects.

In the case of the two dynamic techniques (black-box and white-box), the time elapsed for generating and running the test cases (which encompasses the activities previously mentioned) is taken into account for computing the number of defects detected per hour.

With the aim of striving towards better research practices in SE [27] all the collected measurements are reported. These raw data will help other researchers to verify or re-analyze [28] the experiment results presented in this work. Table 9 shows all the efficiency measurements (defect detection rates) collected during the experiment sessions (the raw data is available in Appendix). A total of 53 measurements were collected, this sample size is slightly greater than the average sample size used in software engineering experiments [16].

6. Analysis and results

This section presents both the collected descriptive and inferential statistics for the efficiency measurements. Table 10 shows the mean defect detection rates and their standard deviations for the testing techniques assessed in the four experiments.

As shown in Table 10, there is not a clear distinction between the efficiency of the different testing techniques. In the first and second experiments, white-box testing seems to be more efficient than black-box testing and code reading, however, in the third and fourth experiments, code reading performs better. With respect to the instrumented programs, Table 11 shows the mean defect detection rates and their standard deviations for the instrumented programs used in the four experiments.

As shown in Table 11, efficiency seems to vary depending on the program. The software program identified as banking, on average, yields an efficiency rate of 1 defect per hour. This program has the data point with the maximum efficiency rate. Conversely, cmdline shows the worst efficiency rate; on average, the efficiency in this program yielded 0.24 defects detected per hour.

Descriptive statistics give us an overview of basic features of the collected efficiency measurements, but at this point, it is not possible to draw any confident conclusions with respect to possible differences between treatments. Once the overview of the data is provided, it is possible to continue testing the hypotheses previously stated using inferential statistics The four experiments can be arranged in a factorial experiment design [3]. The statistical model employed according to the factorial design (3×4) is defined in Equation (1).

$$y_{ijk} = \mu + \alpha_i + \beta_j + (\alpha\beta)_{ij} + \epsilon_{ijk}\,. \qquad (1)$$

In this equation μ is the grand mean, α_i represents the effect of software testing technique i, β_j represents the effect of program j, $(\alpha\beta)_{ij}$ is the interaction effect between treatments i

Table 10. Average defect detection rates and standard deviations of the software testing techniques

Technique	Exp. 1	Exp. 2	Exp. 3	Exp. 4
Code reading	0.4 (0.68)	0.56 (0.75)	0.88 (0.79)	0.36 (0.41)
Black-box	1.18 (0.64)	0.22 (0.44)	0.4 (0.46)	0.24 (0.28)
White-box	1.43 (1.75)	0.79 (0.62)	0.39 (0.49)	0.12 (0.24)

Table 11. Average defect detection rates and standard deviations per instrumented program

Program	Exp. 1	Exp. 2	Exp. 3	Exp. 4
Banking	1 (1.15)	–	–	–
Nametbl	–	0.55 (0.62)	–	–
Ntree	–	–	0.54 (0.58)	–
Cmdline	–	–	–	0.24 (0.31)

and j, k is the number of replications in each treatment combination, and ϵ is the random error which assumes $N(0, \sigma^2)$. The analysis of variance (ANOVA) [2–4] is used to assess the components of the model (such as technique, program and the interaction between technique and program).

Before drawing any conclusions related to the components of the model, it is necessary to assess: 1) that the collected measurements are independent (independence), 2) that the variance is the same for all the measurements (homogeneity), and 3) that the measurements follow a normal distribution (normality).

The first assumption is addressed by the principle of randomization used in the four experiments; all the measurements of one sample are not related to those of the other sample. The second and third assumptions are assessed by using the estimated residuals [2, 3]. To assess the homogeneity of variances, the Levene test for homogeneity of variances was applied [29]. The Levene test allowed to obtain a p-value of 0.7043, which suggests that variance in all treatment combinations (technique and program) are equal (null hypothesis of this test). Thus, the null hypotheses in favour of homogeneity were accepted. The third assumption (normality) was evaluated by applying the Kolmogorov-Smirnov test for normality [30,31]. After applying this test, a p-value of 0.2882 was obtained, which suggests that the residuals fit a normal distribution (null hypothesis of this test). Thus, the null hypothesis in favour of normality was accepted.

Once there is a valid statistical model, it is possible to draw reliable conclusions about the model components (technique, program and the interaction or relationship between technique and program). Table 12 shows the ANOVA results of the model stated in Equation (1).

If an α level of 0.05 is set, none of the components shows a significant difference with respect to efficiency. However, if the alpha level has the value of 0.1, which represents a confidence level of 90%, a significant difference is obtained with respect to efficiency in the program component. This suggests that at least one of the programs has a different level of efficiency than the others. To determine the significant difference the Tukey test for treatment comparisons was used [32]. Table 13 shows program comparisons with respect to efficiency.

As shown in Table 13, it can be observed that there is a significant difference with respect to efficiency between the banking and cmdline programs. This difference has an estimated value of 0.76 defects detected per hour, and suggests that the software program affects, to some degree, the efficiency of the three assessed software testing techniques, as shown in Figure 1.

Since the program component showed a significant difference with respect to efficiency (at $\alpha = 0.1$), it is important to estimate the extent by which efficiency is affected by a software

Table 12. Results of the analysis of variance (ANOVA)

Component	Df	Sum Sq	Mean Sq	F-value	p-value
Technique	2	0.418	0.2089	0.359	0.7003
Program	3	4.072	1.3574	2.335	0.0879
Technique: program	6	3.933	0.6555	1.128	0.3636
Residuals	41	23.835	0.5813		

Table 13. Pairwise comparisons with respect to the defect detection rates

Program comparisons	Difference	p-value
Banking – cmdline	0.7628	0.0621
Nametbl – cmdline	0.4558	0.4022
Ntree – cmdline	0.4581	0.3978
Nametbl – banking	−0.3069	0.7469
Ntree – banking	−0.3046	0.7512
Ntree – nametbl	0.0023	1.0000

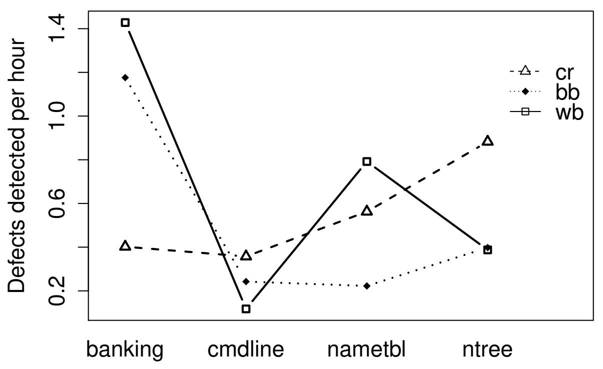


Figure 1. Interaction plot between the technique and the program

program type. Cohen's f was selected as the coefficient for assessing the average effect in the ANOVA program component across all its levels [33]. This coefficient can take values from zero to indefinitely large values. Cohen [33] suggests that values of 0.10, 0.25, and 0.40 represent small, medium, and large effect sizes, respectively. After estimating this coefficient, an effect size of $f = 0.41$ was obtained, which suggests a large effect size regarding the type of the used program.

With the effect size f estimated, it is possible to assess how sensitive (power test) any of the ANOVA components were in detecting an effect. We applied a post-hoc test to assess the degree of power achieved by the ANOVA program component. The power in a statistical test is equal to $1-\beta$, where β is the probability of making a Type II error. For program component we obtained a power of 0.79 (at α level $= 0.1$), which suggests that the acceptable level of power for the estimated effect size ($f = 0.41$) and the used sample size (53 collected measurements).

6.1. Defect analysis

In order to extend the previous efficiency analysis, the type of defects injected in the instrumented programswere scrutinized. The aim of this secondary analysis is to determine what classes of defects are detected by the studied testing techniques. As previously discussed in Section 5, defects were characterized by two classification schemes [5,25]: Scheme 1 consisting of omission and commission defects, and scheme 2 consisting of cosmetic, initialization and control defects. Figure 2 shows the percentage of observed defects of the testing techniques (black-box [bb], white-box [wb] and code reading [cr]) split into the two defect classification schemes.

Figure 2 shows that participants using the code reading technique seem to observe more initialization defects than participants using the other techniques. Conversely, code reading and white-box seem to behave worse detecting cosmetic defects than black-box.

Similar to Figure 2, Figure 3 shows the percentage of observed defects by an instrumented program and by a defect type. As shown in Figure 3, all the techniques seem to produce worse results in the detection of cosmetic defectsthan black-box. Conversely, cosmetic defects injected in nametbl (na) and cmdline (cm) programs seem to negatively impact the percentage of observed defects.

Figures 2 and 3 give us an overview of the observed defects in these two schemes, however, an inferential analysis is needed to examine pos-

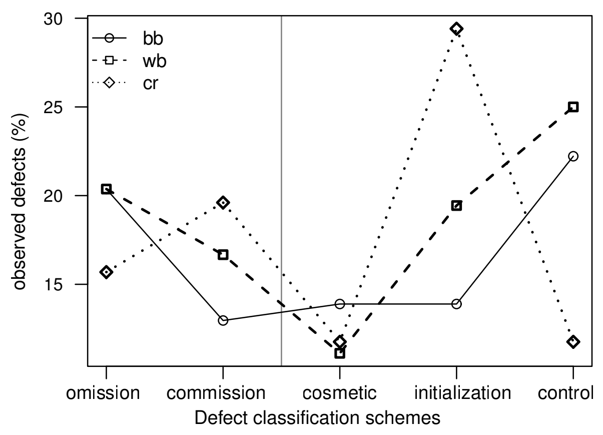


Figure 2. Types of defects observed by the testing technique

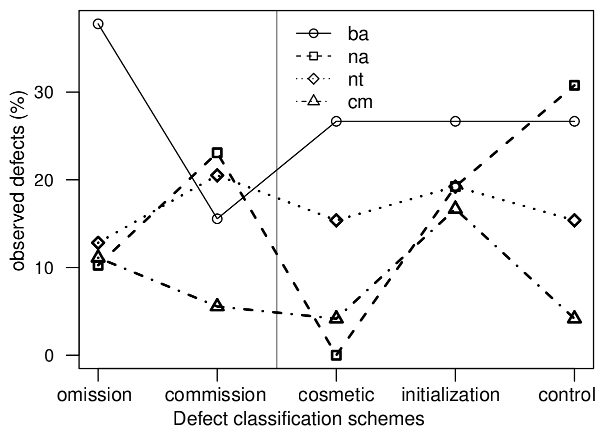


Figure 3. Types of defects observed by instrumented program

Table 14. Results of the analysis of variance (ANOVA) for defect classification scheme 1 (omission and commission defects)

Component	Df	Sum Sq	Mean Sq	F-value	p-value
Technique	2	62	30.9	0.052	0.9497
Program	3	4573	1524.2	2.548	0.0615
Scheme1	1	168	167.7	0.280	0.5979
Technique: program	6	3577	596.1	0.997	0.4334
Technique: scheme1	2	580	290.1	0.485	0.6175
Program: scheme1	3	5250	1749.8	2.925	0.0387
Tech.: prog.: scheme1	6	6086	1014.3	1.695	0.1325
Residuals	82	49056	598.2		

sible significant differences. Next the ANOVA results are presented according to the used defect classification schemes.

6.1.1. ANOVA results for defect classification scheme 1

The statistical model employed for this ANOVA is shown in Equation (2).

$$y_{ijkl} = \mu + \alpha_i + \beta_j + \gamma_k + (\alpha\beta)_{ij} + (\alpha\gamma)_{ik} + (\beta\gamma)_{jk} + (\alpha\beta\gamma)_{ijk} + \epsilon_{ijkl}\,. \tag{2}$$

In this equation μ is the grand mean; α_i represents the effect of software testing technique i, β_j represents the effect of program j, γ_k represents the effect of defect type k on the deffect classification scheme 1, $(\alpha\beta)_{ij}$ is the interaction effect between treatments i and j, $(\alpha\gamma)_{ik}$ is the interaction effect between treatments i and k, $(\beta\gamma)_{jk}$ is the interaction effect between treatments j and k, $(\alpha\beta\gamma)_{ijk}$ is the interaction effect between treatments i, j and k, l is the number of replications in each treatment combination, and ϵ is the random error which assumes $N(0, \sigma^2)$. The ANOVA results of this model are shown in Table 14.

As shown in Table 14 the program and the program:scheme1 components show a significant difference at alpha level of 0.1 and 0.05, respectively. To inspect the significant differences in these two components, the Tukey test for treatment comparisons was used [32]. Table 15 shows the pairwise comparisons of the program component and the interaction component (this between program and scheme 1).

A significant difference of 18% is observed between banking (ba) and cmdline (cm) programs. Participants applying the testing techniques observed more defects in the banking (ba) program. Another significant difference was observed between the omission defects of the banking program and the commission defects

Table 15. Significant pairwise comparisons for defect classification scheme 1

Pairwise comparisons	Difference (%)	p-value
Banking – cmdline	18.3333	0.0374
Banking: omission – cmdline: commission	32.3414	0.0213075
Nametbl: omission – banking: omission	−27.6415	0.0698521

of the cmdline program. Omission defects were the most commonly observed in these two programs. The third significant difference was observed between omission defects in the nametbl and banking programs, omission defects were the most commonly observed in the banking program.

Concerning model assumptions, the Levene test for homogeneity of variances [29] shows a non-significant p-value (0.9292), suggesting that variance in all treatment combinations (technique, program and defect classification scheme) are equal (the null hypothesis of this test). The assumption of normality was checked with the Kolmogorov-Smirnov test for normality [30, 31]. In this case the test showed a significant difference (p-value = 0.00012). Because measurements are represented as proportions (or percentages), these kinds of measurements can be prone to departures from normality, as shown in Figure 4.

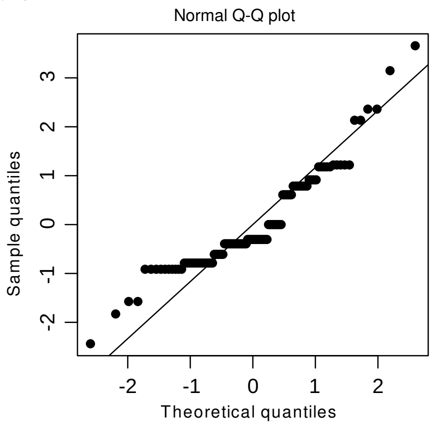


Figure 4. Normal Q-Q plot of standardized residuals given the model presented in Equation (2)

6.1.2. ANOVA results for defect classification scheme 2

Next the inferential analysis concerning the second defect classification scheme which is composed of cosmetic, initialization and control defect types is presented. Using the same statistical model as in Equation (2), but changing γ_k, representing now the effect of the defect type k of the defect classification scheme 2. Table 16 shows the results of the analysis of variance.

The results presented in Table 16 suggest a significant difference (at an alpha of 0.05) in the program component. The Tukey [32] was run to examine which of the program pairwise comparisons show significant differences. Table 17 shows the pairwise comparisons of the program component.

Table 17 suggests a significant difference of 18% between the banking (ba) and the cmdline (cm) program. Participants using the testing techniques observed a significantly larger number of defects in the banking program than in the cmdline program.

In relation to the model assumptions, the Levene test for homogeneity of variances [29] shows a non-significant p-value of 0.7501 in favor of the equality variances among treatments, however, in the same way as in the previous analysis, some departures from normality were observed with the Kolmogorov-Smirnov test [30, 31], a significant p-value of 0.00012 was observed.

7. Network meta-analysis

The results of the replication reported here can be incorporated in the existing evidence of related experiments. With the quantitative information available in similar experiments [5–8, 11] it is possible to carry out a network meta-analysis in order to offer better informed decisions on the efficiency of the testing techniques reported in previous experiments along with the one discussed in this work.

The network meta-analysis approach (NMA) [34, 35], also known as multiple treatment comparison or mixed treatment comparison, has

Table 16. Results of the analysis of variance (ANOVA) for scheme 2 (cosmetic, initialization and control defects)

Component	Df	Sum Sq	Mean Sq	F-value	p-value
Technique	2	93	46.3	0.055	0.946
Program	3	6859	2286.3	2.727	0.047
Scheme2	2	2296	1147.8	1.369	0.258
Technique: program	6	5365	894.2	1.067	0.386
Technique: scheme2	4	3826	956.6	1.141	0.340
Program: scheme2	6	5224	870.7	1.039	0.404
Tech.: prog.: scheme2	12	3904	325.3	0.388	0.966
Residuals	123	103125	838.4		

Table 17. Significant pairwise comparisons for defect classification scheme 2

Pairwise comparisons	Difference (%)	p-value
Banking – cmdline	18.3333	0.0274

been increasingly widespread in recent years in the health care arena [36–38].

The network meta-analysis approach can integrate direct and indirect evidence in a collection of studies (or experiments). This approach provides information on the relative effects of three or more treatments for the same outcome [39]. Conversely to classical meta-analysis, NMA simultaneously compares the effects of three or more treatments.

Given the evidence of the present replication (pooling together the four experiments as one experiment replication) along with the evidence of seven related experiments [5–8,11], NMA with the 'netmeta' R package [40] was performed to assess the available evidence of the efficiency of the testing techniques: black-box (bb), white-box (wb) and code reading (cr). Table 18 shows the sample sizes (n), average defect detection rates (mean) and the standard deviations (sd) of the three testing techniques examined in the aforementioned experiments.

With the information available in Table 18 it is possible to carry out NMA. Asa result of conducting all these experiments to examine the same three testing techniques it can be concluded that they conform to a single design providing only direct evidence. NMA can also be applied to estimate indirect evidence, however, not in this case. For example, suppose there are experiments examining treatments A and B and experiments examining treatments A and C (for the same outcome), these experiments can be pooled together in NMA to obtain an indirect estimate for indirect comparison between treatments B and C by means of a common comparator, that is treatment A.

Table 19 shows the resulting NMA obtained on the basis of the information of Table 18. The results are presented in the matrix of estimated overall effect sizes (with lower and upper confidence limits) belonging to all the pairwise treatment comparisons. Effect sizes were computed using the standardized mean difference (Hedges' g) [41]. The guidelines proposed by Cohen [33] suggest that effect sizes of 0.2, 0.5 and 0.8 represent small, medium and large effect sizes, respectively. Due to possible context differences in these experiments, the effect sizes shown in Table 19 were estimated according to a random effects model, assuming that the underlying effects in the experiments of the same treatment comparison come from a common normal distribution, i.e. an account for unexplained heterogeneity was assumed.

To obtain valid conclusions from NMA, the resulting network of treatments should be assessed against the transitivity and consistency assumptions [42–44]. In the case of the transitivity assumption, the network is assumed to maintain transitivity whenever pairwise treatment

Table 18. Sample sizes, average defect detection rates and standard deviations of the testing techniques examined in eight experiments

Experiment	n_{bb}	mean_{bb}	sd_{bb}	n_{wb}	mean_{wb}	sd_{wb}	n_{cr}	mean_{cr}	sd_{cr}
umd82 [5]	29	1.58	0.90	29	1.40	0.87	29	1.90	1.83
umd83 [5]	13	1.22	0.91	13	1.18	0.84	13	0.56	0.46
umd84 [5]	32	1.84	1.06	32	1.82	1.24	32	3.33	3.42
ukl94 [6, 7]	27	4.67	2.27	27	2.92	1.59	27	2.11	1.12
ukl95 [6, 7]	21	3.08	1.28	18	2.00	1.59	17	1.74	0.67
uos97 [8]	47	2.47	1.10	47	2.20	0.94	47	1.06	0.75
uok11 [11]	18	2.46	0.58	18	2.50	0.83	18	2.16	0.55
epch14	18	0.54	0.60	18	0.73	1.06	17	0.54	0.64

Table 19. Pairwise treatment overall effect size estimates, lower and upper 95% confidence limits under a random effects model

Technique	Black-box (bb)	Code reading (cr)	White-box (wb)
Black-box (bb)	–	$0.576\ (0.110, 1.042)$	$0.239\ (-0.224, 0.701)$
Code reading(cr)	$-0.576\ (-1.042, -0.110)$	–	$-0.337\ (-0.802, 0.127)$
White-box (wb)	$-0.239\ (-0.701, 0.224)$	$0.337\ (-0.127, 0.802)$	–

effects are similarly distributed across the studies (experiments). For example, suppose some studies assessing treatments A, B, C for the same outcome, if treatment A performs better than B, and treatment B performs better than C, then treatment A has to perform better than C (transitivity is met). Departures from transitivity arise when significant heterogeneity is present across one or more pairwise treatment comparisons in the network. On the other hand, the consistency assumption states that both direct and indirect evidence in a given pairwise treatment comparison (network edge) should be similar. This assumption only applies to situations where there is both direct and indirect evidence in one or more edges of the network [42].

In a similar way when the Q statistic is used in pairwise meta-analysis, a generalization of such index is used in NMA. In NMA, the Q statistic measures the deviation from heterogeneity/inconsistency. Index Q can be separated into parts for each pairwise treatment comparison and a part for the remaining inconsistency between all the treatment pairwise comparisons [43].

Given the resulting NMA in Table 19, the statistical test for assessing the heterogeneity/inconsistency of the network is run. In the same manner as in pairwise meta-analysis, in NMA the used Q statistic follows a Chi-Squared distribution. The test showed a Q-value of 74.12, corresponding with a significant p-value smaller than 0.0001, thus suggesting a significant degree of heterogeneity in the network. The I^2 index that represents the percentage of heterogeneity also showed a high value of 81.1%.

The heterogeneity found in the network suggests that at least one pairwise treatment comparison contains contradictory effect size estimates, yielding a significant heterogeneity in the network edge. Because of this situation, it was decided to assess the heterogeneity (under classical meta-analysis also using Hedges' g [41]) in each network edge, i.e. with the following pairwise comparisons: black-box (bb) vs. code reading (cr), black-box (bb) vs. white-box (wb) and code reading (cr) vs. white-box (wb). Table 20 shows the Q and I^2 coefficients in each network edge.

Table 20. Assessment of heterogeneity in each network edge

Edge	Q	p-value	I^2
bb, cr	60.81	<0.0001	88.5%
bb, wb	11.40	0.1221	38.6%
cr, wb	42.40	<0.0001	83.5%

According to Table 20 the pairwise comparison between black-box and white-box yields con-

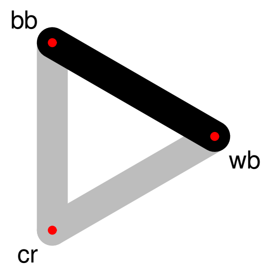


Figure 5. Network graph with a consistent edge highlighted

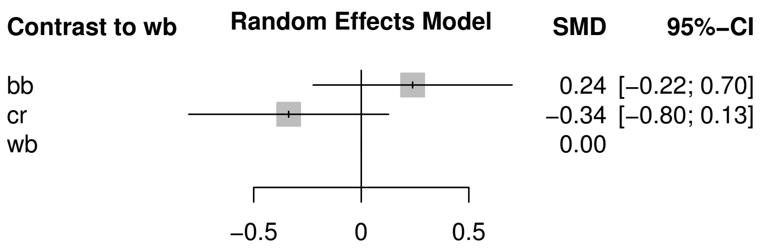


Figure 6. Forest plot with the overall effect size estimates and confidence limits of the testing techniques

sistent results (p-value is non-significant) suggesting a degree of homogeneity among effect size estimates of the eight experiments. The resulting I^2 coefficient indicates that experiments in this edge present a low level of heterogeneity which is non-significant. As observed in Table 20, the rest of pairwise comparisons show a significant difference. Figure 5 shows the resulting network graph with the pairwise treatments. The network is laid out in a plane where nodes correspond to the treatments (bb, wb and cr) whereas edges represent the pairwise treatment comparisons; the observed consistent edge is highlighted in black. The thickness of the lines represents the number of experiments available for each treatment, in this case, eight experiments.

Figure 6 displays a forest plot of pairwise overall effect size estimates using the white-box technique as the reference treatment. As it was discussed ealier, only the pairwise comparison between black-box and white-box shows homogeneity in its effect size estimates. It is visible that a small effect size of 0.24 is observed in favour of the black-box technique, however, the estimated confidence limits indicate that the overall effect size could be zero, thus suggesting that both black-box and white-box yield similar defect detection rates.

8. Discussion

Having presented the analysis, in this section the findings in reference to the research questions stated and previous work are discussed.

According to the evidence collected in the four experiments, all the testing techniques showed similar levels of efficiency (no significant difference was observed); this suggests that efficiency is not affected by the testing techniques **(RQ1)**. Evidence suggesting that the type of used software program affects the efficiency of the studied testing techniques has been found **(RQ2)**. In addition, the current evidence suggests that efficiency is not affected by the relationship between techniques and software programs, i.e. both factors are independent **(RQ3)**.

In relation to the baseline experiment, the results support the results presented in experiments umd82 and umd83 [5], the three testing techniques behave in a similar way. With regard to the efficiency of testing techniques, Table 21 shows a comparison between the reported results in [5] and the pooled results. These results indicate the average number of defects detected per hour.

As shown in Table 21, this replication supports the results of the umd82 and umd83 experiments [5]. In these experiments, the null hypothesis is accepted because the three testing techniques do not show significant differences in the defect detection rates. However, this difference is significant in umd84 [5]. One possible reason for this significant difference could be the participants' expertise. In the third experiment reported in [5], participants were programming professionals with high levels of technical skill.

As noted in Table 21, the obtained rates of efficiency were lower than those reported in [5], perhaps the number of defects injected in the instrumented programs (six per program) or the participants' expertise yielded a lower efficiency rate. One point worth noting about the umd82 and umd83 experiments in [5] is that although similar kinds of participants were used, the efficiency measurements in the first experiment are

Table 21. Average defect detection rates per a testing technique reported in [5] and in replication

Technique	umd82 [5]	umd83 [5]	umd84 [5]	epch14
Code reading	1.90	0.56	3.33	0.54
Black-box	1.58	1.22	1.84	0.54
White-box	1.40	1.18	1.82	0.73

slightly higher than those in the second experiment. One possible reason for this is that both the software type and the injected defects affected the efficiency. In the three experiments reported in [5], the authors did not use the same instrumented programs in all experiments. It is also important to note that in [5], each program had a different number of defects with respect to both defect classification schemes; i.e. the used programs were not equally balanced regarding the number of defects and defect types. It is highly probable that differences in the program type and the used defects produced an interaction effect with the testing techniques, as mentioned in [5]. With the aim of avoiding this effect interaction, the same number and the same type of defects were employed in the instrumented programs, and only the type od software program varied.

With respect to the average defect detection rate per program, the obtained results support those in umd82 and umd84 [5], the type of software impacts on the efficiency of the testing techniques. Table 22 shows the average defect detection rates of umd82, umd83, umd84 and the results obtained here, the efficiency seems to vary depending on the software type.

It is possible that the low rates of efficiency observed in the replication are due to the expertise level of the participants. According to our evidence, it seems that participants had a better understanding of the domain related to the banking instrumented program than the domains related to the other programs. Cyclomatic complexity is discarded as a possible factor that could affect efficiency. Programs nametbl, ntree and cmdline have similar levels of VG (43, 46 and 45, respectively), however, cmdline still showed the worst efficiency rate. This evidence seems to reinforce the idea that the knowledge of the program domain could affect the efficiency of software testing techniques.

Concerning the defect analysis (secondary research question, **SRQ1**), no significant differences in the two defect classification schemes were observed in the case of the testing techniques. The results in the baseline experiment [5] suggest that participants applying code reading and those applying black-box observed significantly more omission defects than those applying white-box. In the case of the conducted experiments no significant difference between omission and commission defects was observed.

With regard to the second defect classification scheme the authors in [5] observed that participants using code reading and those using black-box observed significantly more initialization defects than those using white-box. Participants using code reading observed significantly more interface defects than those using either black-box or white-box. Participants using black-box observed significantly more control defects than those using the other two techniques. Participants using code reading observed significantly more computation defects than those using white-box. With regard to data and cosmetic defects the authors in [5] did not observe significant differences. In the case of the experiments described here, it was observed that code reading seemed to detect more initialization defects than white-box and black-box but the difference was not significant. It was also found out that white-box and black-box seemed to detect more control defects than code reading, but again, this difference was not significant.

However, with respect to the instrumented programs, significant differences were observed, more omission defects were detected in the banking program compared to cmdline program (defect classification scheme 1). Similarly there were more cosmetic, initialization and control defects (defect scheme 2) observed in the banking program than in the cmdline program. Although

Table 22. Average defect detection rates per program reported in [5] and in replication

Program	umd82 [5]	umd83 [5]	umd84 [5]	epch14
Formatter	1.60	0.98	2.15	–
Plotter	1.19	0.92	–	–
Data type	2.09	–	3.70	–
Database	–	1.05	1.14	–
Banking	–	–	–	1
Nametbl	–	–	–	0.55
Ntree	–	–	–	0.54
Cmdline	–	–	–	0.24

same defect types (for both schemes) were equally injected in the four instrumented programs, they were observed in a similar way, perhaps these findings suggest that the knowledge of the domain of the program to be tested may impact the efficiency of the testing techniques.

The replication results along with the related existing ones allowed to carry out a quantitative synthesis using the network meta-analysis (NMA) approach. Taking into account the quantitative information of eight experiments run in five countries (USA, Germany, UK, India and Ecuador) only consistent results were observed among black-box and white-box techniques, both techniques yielded similar efficiency rates. The code reading technique compared with either black-box or white-box techniques showed inconsistent results (presence of heterogeneity). These results suggest that the code reading technique shows a greater level of sensitivity. In umd84 [5], code reading was significantly more efficient than black-box and white-box, probably in this experiment the expertise had an impact on this technique. In the umd84 experiment, participants were programming professionals with an overall average of ten years of professional experience. However, the expertise factor does not seem to significantly affect the efficiency of the black-box and white-box, in this treatment comparison (bb vs. wb), participants used in the pooled experiments spanned different expertise levels, such as undergraduate, graduate and professionals. In the case of the other treatment comparisons (cr vs. bb, and cr vs. wb), further subgroup analyses [45, 46] can be performed to identify moderators affecting the outcomes.

For the purpose reporting this experiment replication, the guidelines of [19] were taken as reference, however, it is not so clear how to proceed when reporting a replication in the context of a family of related experiments. In this sense, the authors propose that the family of experiments be explained in terms of the main treatments studied along with the contextual information. It is also proposed that the findings of the family be organized, first, according to the cause and effect constructs and, second, according to the cause and effect operationalizations that the related experiments address. The findings of the family can be presented as a narrative or quantitative synthesis (or both of them).

The authors also propose to describe the baseline experiment, this activity underlies the essence of the replication, which refers to the repetition of a previously run experiment [14], it is recomended to describe the main findings and contextual information of the baseline experiment. Once the family and the baseline have been described, the replication and the analysis of the results should be described. Next, the replication findings should be analyzed in relation to the baseline experiment along with the related experiments of the family. Depending on the type of analysis done, this activity can be performed as a qualitative or quantitative synthesis (or both of them). Finally, a discussion of the consistency or inconsistency of the findings should be addressed.

Regarding the limitations of the experiment replication reported here, below the strategies used to minimize the threats to validity are described [47]. With respect to conclusion validity, the measurements collected in these experiment sessions satisfy the principles of independence, homogeneity and normality. With respect to internal validity, participants were randomly assigned to treatments, which reduced learning effects. Boredom or fatigue was reduced by using alternate sessions. Participants were in the same classroom, working under the same conditions, and sitting apart with no interaction. With respect to construct validity, cause and effect constructs were operationalized in the same way as is reported in [5] and in [9]. With respect to external validity, the use of students instead of practitioners might have compromised this type of validity. However, there exists some evidence suggesting that in some contexts, the results of empirical studies that employ students with enough technical skills are equivalent to the results of empirical studies that use practitioners [48]. For example, using students in their last academic year of an undergraduate program as experiment participants may be comparable to using junior practitioners as participants. In fact, it is common for students in their last academic year to work part-time in IT-related companies. In this sense, the results presented here may be generalizable to junior practitioners.

9. Conclusions

In this work the efficiency of three software testing techniques has been assessed. The replication was composed of four experiments where several instrumented software programs were used. The obtained results suggest that software testing techniques perform in a similar way, but the domain related to the software to be tested might have an effect on the defect detection rates of the testing techniques. We suggest that software verification activities such as software testing be performed only after software engineers have a clear understanding of the software product domain.

The main contributions of this work are the following: 1) the execution of a controlled experiment replication in order to verify previous findings, and 2) the realization of a quantitative synthesis with the aim of consolidating the findings belonging to a family of related experiments.

Acknowledgements

This research study received support from the Prometeo project 20140697BP funded by the Government of the Republic of Ecuador's Department of Higher Education, Science, Technology and Innovation (Senescyt). Special thanks to Jahzeel J. Coss who developed and instrumented the programs used in the experiment replication reported here. César Pardo acknowledges the contribution of the University of Cauca, where he works as an assistant professor.

References

[1] S. McConnell, *Code Complete*, 2nd ed. Redmond, WA, USA: Microsoft Press, 2004.

[2] G.E.P. Box, W.G. Hunter, J.S. Hunter, and W.G. Hunter, *Statistics for Experimenters: An Introduction to Design, Data Analysis, and Model Building.* John Wiley & Sons, Jun. 1978.

[3] R. Kuehl, *Design of Experiments: Statistical Principles of Research Design and Analysis*, 2nd ed. California, USA: Duxbury Thomson Learning, 2000.

[4] N. Juristo and A.M. Moreno, *Basics of Software Engineering Experimentation.* Kluwer Academic Publishers, 2001.

[5] V. Basili and R. Selby, "Comparing the effectiveness of software testing strategies," *IEEE Trans. Softw. Eng.*, Vol. 13, No. 12, 1987, pp. 1278–1296.

[6] E. Kamsties and C.M. Lott, *An Empirical Evaluation of Three Defect-Detection Techniques.* Berlin, Heidelberg: Springer, 1995, pp. 362–383.

[7] E. Kamsties and C. Lott, "An empirical evaluation of three defect detection techniques," Dept. Computer Science, University of Kaiserslautern, Kaiserslautern, Germany, Tech. Rep. ISERN 95-02, 1995.

[8] M. Roper, M. Wood, and J. Miller, "An empirical evaluation of defect detection techniques," *Information and Software Technology*, Vol. 39, No. 11, 1997, pp. 763–775.

[9] N. Juristo and S. Vegas, "Functional testing, structural testing and code reading: What fault type do they each detect?" in *Empirical Methods and Studies in Software Engineering*, ser. Lecture Notes in Computer Science, R. Conradi and A. Wang, Eds. Berlin, Heidelberg: Springer, 2003, Vol. 2765, pp. 208–232.

[10] N. Juristo, S. Vegas, M. Solari, S. Abrahao, and I. Ramos, "Comparing the effectiveness of equivalence partitioning, branch testing and code reading by stepwise abstraction applied by subjects," in *IEEE Fifth International Conference on Software Testing, Verification and Validation (ICST)*, Apr. 2012, pp. 330–339.

[11] S.U. Farooq and S. Quadri, "An externally replicated experiment to evaluate software testing methods," in *Proceedings of the 17th International Conference on Evaluation and Assessment in Software Engineering*, ser. EASE '13. New York, NY, USA: ACM, 2013, pp. 72–77.

[12] O.S. Gómez, R.A. Aguilar, and J.P. Ucán, "Efectividad de técnicas de prueba de software aplicadas por sujetos novicios de pregado," in *Encuentro Nacional de Ciencias de la Computación, (ENC)*, M.D. Rodríguez, A.I. Martínez, and J.P. García, Eds., Ocotlán de Morelos, Oaxaca, México, Nov. 2014.

[13] O.S. Gómez, N. Juristo, and S. Vegas, "Understanding replication of experiments in software engineering: A classification," *Information and Software Technology*, Vol. 56, No. 8, 2014, pp. 1033–1048.

[14] N. Juristo and O.S. Gómez, "Replication of software engineering experiments," in *Empirical Software Engineering and Verification: LASER Summer School 2008–2010*, ser. Lecture Notes in Computer Science, B. Meyer and M. Nordio, Eds. Berlin, Heidelberg: Springer, Nov. 2011, Vol. 7007, pp. 60–88.

[15] O.S. Gómez, "Tipología de replicaciones para la síntesis de experimentos en ingeniería del software," Ph.D. dissertation, Facultad de Informática de la Universidad Politécnica de Madrid, Campus de Montegancedo, 28660, Boadilla del Monte, Madrid, España, May 2012.

[16] D. Sjøberg, J. Hannay, O. Hansen, V. Kampenes, A. Karahasanovic, N.K. Liborg, and A. Rekdal, "A survey of controlled experiments in software engineering," *Software Engineering, IEEE Transactions on*, Vol. 31, No. 9, Sep. 2005, pp. 733–753.

[17] F. da Silva, M. Suassuna, A. França, A. Grubb, T. Gouveia, C. Monteiro, and I. dos Santos, "Replication of empirical studies in software engineering research: A systematic mapping study," *Empirical Software Engineering*, Vol. 19, No. 3, 2014, pp. 501–557.

[18] R.C. Linger, B.I. Witt, and H.D. Mills, *Structured Programming; Theory and Practice the Systems Programming Series.* Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 1979.

[19] J. Carver, "Towards reporting guidelines for experimental replications: A proposal," in *Proceedings of the 1st International Workshop on Replication in Empirical Software Engineering Research (RESER)*, Cape Town, South Africa, May 2010.

[20] W. Howden, "Functional program testing," *IEEE Transactions on Software Engineering*, Vol. 6, 1980, pp. 162–169.

[21] G.J. Myers, *The Art of Software Testing.* New York: John Wiley & Sons, 1979.

[22] B. Marick, *The craft of software testing: subsystem testing including object-based and object-oriented testing.* Upper Saddle River, NJ, USA: Prentice-Hall, Inc., 1995.

[23] B. Beizer, *Software testing techniques*, 2nd ed. New York, NY, USA: Van Nostrand Reinhold Co., 1990.

[24] H.L. Dreyfus and S. Dreyfus, *Mind over Machine. The Power of Human Intuition and Expertise in the Era of the Computer.* New York: Basil Blackwell, 1986.

[25] V. Basili and B. Perricone, "Software errors and complexity: An empirical investigation," *Commun. ACM*, Vol. 27, No. 1, 1984, pp. 42–52.

[26] V. Basili, G. Caldiera, and H. Rombach, "Goal question metric paradigm," in *Encyclopedia of Software Engineering*, J.J. Marciniak, Ed. Wiley-Interscience, 1994, pp. 528–532.

[27] P. Louridas and G. Gousios, "A note on rigour and replicability," *SIGSOFT Softw. Eng. Notes*, Vol. 37, No. 5, Sep. 2012, pp. 1–4.

[28] O.S. Gómez, N. Juristo, and S. Vegas, "Replication, reproduction and re-analysis: Three ways for verifying experimental findings," in *International Workshop on Replication in Empirical Software Engineering Research (RESER)*, Cape Town, South Africa, May 2010.

[29] H. Levene, "Robust tests for equality of variances," in *Contributions to probability and statistics*, I. Olkin, Ed. Palo Alto, CA: Stanford Univ. Press, 1960, pp. 278–292.

[30] A.N. Kolmogorov, "Sulla determinazione empirica di una legge di distribuzione," *Giornale dell'Istituto Italiano degli Attuari*, Vol. 4, 1933, pp. 83–91.

[31] N.V. Smirnov, "Table for estimating the goodness of fit of empirical distributions," *Ann. Math. Stat.*, Vol. 19, 1948, pp. 279–281.

[32] J. Tukey, "Comparing individual means in the analysis of variance," *Biometrics*, Vol. 5, No. 2, 1949, pp. 99–114.

[33] J. Cohen, *Statistical power analysis for the behavioral sciences.* Hillsdale, NJ: L. Erlbaum Associates, 1988.

[34] T. Lumley, "Network meta-analysis for indirect treatment comparisons," *Statistics in Medicine*, Vol. 21, No. 16, 2002, pp. 2313–2324.

[35] G. Lu and A.E. Ades, "Combination of direct and indirect evidence in mixed treatment comparisons," *Statistics in Medicine*, Vol. 23, No. 20, 2004, pp. 3105–3124.

[36] T. Greco, G. Biondi-Zoccai, O. Saleh, L. Pasin, L. Cabrini, A. Zangrillo, and G. Landoni, "The attractiveness of network meta-analysis: A comprehensive systematic and narrative review," *Heart, Lung and Vessels*, Vol. 7, No. 2, 2015, pp. 133–142.

[37] A. Bafeta, L. Trinquart, R. Seror, and P. Ravaud, "Reporting of results from network meta-analyses: Methodological systematic review," *BMJ*, Vol. 348, 2014. [Online]. http://www.bmj.com/content/348/bmj.g1741

[38] A. Nikolakopoulou, A. Chaimani, A.A. Veroniki, H.S. Vasiliadis, C.H. Schmid, and G. Salanti, "Characteristics of networks of interventions: A description of a database of 186 published networks," *PLoS ONE*, Vol. 9, No. 1, Dec. 2014, pp. 1–10.

[39] A. Chaimani and G. Salanti, "Visualizing assumptions and results in network meta-analysis: The network graphs package," *Stata Journal*, Vol. 15, No. 4, 2015, pp. 905–950.

[40] G. Rücker, G. Schwarzer, U. Krahn, and J. König, *netmeta: network Meta-Analysis using Frequentist Methods*, 2016, R package version 0.9-0. [Online]. https://CRAN.R-project.org/package=netmeta

[41] L.V. Hedges and I. Olkin, *Statistical methods for meta-analysis.* Orlando: Academic Press, 1985.

[42] F. Song, Y.K. Loke, T. Walsh, A.M. Glenny, A.J. Eastwood, and D.G. Altman, "Methodological problems in the use of indirect comparisons for evaluating healthcare interventions: Survey of published systematic reviews," *BMJ*, Vol. 338, 2009.

[43] J.P.T. Higgins, D. Jackson, J.K. Barrett, G. Lu, A.E. Ades, and I.R. White, "Consistency and inconsistency in network meta-analysis: Concepts and models for multi-arm studies," *Research Synthesis Methods*, Vol. 3, No. 2, 2012, pp. 98–110.

[44] J.P. Jansen and H. Naci, "Is network meta-analysis as valid as standard pairwise meta-analysis? it all depends on the distribution of effect modifiers," *BMC Medicine*, Vol. 11, May 2013, pp. 159–159.

[45] M. Borenstein, L.V. Hedges, J.P. Higgins, and H.R. Rothstein, *Introduction to Meta-Analysis.* United Kingdom: John Wiley & Sons, Ltd, 2009.

[46] M. Ciolkowski, "What do we know about perspective-based reading? An approach for quantitative aggregation in software engineering," in *Proceedings of the 2009 3rd International Symposium on Empirical Software Engineering and Measurement*, ser. ESEM '09. Washington, DC, USA: IEEE Computer Society, 2009, pp. 133–144.

[47] T. Cook and D. Campbell, *The design and conduct of quasi-experiments and true experiments in field settings.* Chicago: Rand McNally, 1976.

[48] P. Runeson, "Using students as experiment subjects – an analysis on graduate and freshmen student data," in *Proceedings of the 7th International Conference on Empirical Assessment in Software Engineering*, Keele University, UK, 2003, pp. 95–102.

Appendix: Experiment replication raw data

In this appendix we provide the measurements collected in our experiment replication. Table A shows the defects observed and the time that participants spent applying the testing techniques.

Table A. Observed defects and time spent applying the testing techniques

Case	Technique	Program	F1	F2	F3	F4	F5	F6	Minutes
1348	white-box	cmdline					●		127
1349	white-box	cmdline							94
1350	white-box	cmdline							116
1351	white-box	cmdline							280
1343	black-box	cmdline							106
1342	black-box	cmdline						●	120
1344	black-box	cmdline							105
1345	black-box	cmdline						●	127
1355	code reading	cmdline							79
1356	code reading	cmdline						●	84
1358	code reading	cmdline							163
1357	code reading	cmdline				●		●	166
1291	white-box	banking	●					●	109
1292	white-box	banking			●				120
1293	white-box	banking							219
1294	white-box	banking			●			●	109
1295	white-box	banking	●		●		●	●	54
1285	black-box	banking			●				110
1286	black-box	banking	●	●			●	●	106
1287	black-box	banking	●		●				132
1288	black-box	banking			●		●		110
1289	black-box	banking			●				56
1297	code reading	banking							138
1298	code reading	banking		●		●	●	●	154
1299	code reading	banking						●	134
1300	code reading	banking							137
1301	code reading	banking							87
1312	white-box	nametbl	●			●			94
1310	white-box	nametbl	●			●			87
1311	white-box	nametbl							142
1314	white-box	nametbl	●		●				115
1313	white-box	nametbl				●			230
1306	black-box	nametbl	●		●				135
1304	black-box	nametbl							146
1305	black-box	nametbl							134
1307	black-box	nametbl							134
1316	code reading	nametbl			●	●		●	114
1319	code reading	nametbl	●						89
1317	code reading	nametbl							72
1318	code reading	nametbl							250
1329	white-box	ntree	●	●					116
1331	white-box	ntree							141
1330	white-box	ntree				●			115
1332	white-box	ntree							285
1323	black-box	ntree						●	141
1324	black-box	ntree		●				●	107
1327	black-box	ntree							113
1325	black-box	ntree				●			137
1326	black-box	ntree							108
1336	code reading	ntree							85
1335	code reading	ntree			●	●			112
1338	code reading	ntree				●	●	●	97
1337	code reading	ntree		●					100

e-Informatica Software Engineering Journal, Volume 11, Issue 1, 2017, pages: 103–116, DOI 10.5277/e-Inf170105

NRFixer: Sentiment Based Model for Predicting the Fixability of Non-Reproducible Bugs

Anjali Goyal*, Neetu Sardana*
*Jaypee Institute of Information Technology, Noida, India
anjaligoyal19@yahoo.in, neetu.sardana@jiit.ac.in

Abstract

Software maintenance is an essential step in software development life cycle. Nowadays, software companies spend approximately 45% of total cost in maintenance activities. Large software projects maintain bug repositories to collect, organize and resolve bug reports. Sometimes it is difficult to reproduce the reported bug with the information present in a bug report and thus this bug is marked with resolution non-reproducible (NR). When NR bugs are reconsidered, a few of them might get fixed (NR-to-fix) leaving the others with the same resolution (NR). To analyse the behaviour of developers towards NR-to-fix and NR bugs, the sentiment analysis of NR bug report textual contents has been conducted. The sentiment analysis of bug reports shows that NR bugs' sentiments incline towards more negativity than reproducible bugs. Also, there is a noticeable opinion drift found in the sentiments of NR-to-fix bug reports. Observations driven from this analysis were an inspiration to develop a model that can judge the fixability of NR bugs. Thus a framework, NRFixer, which predicts the probability of NR bug fixation, is proposed. NRFixer was evaluated with two dimensions. The first dimension considers meta-fields of bug reports (model-1) and the other dimension additionally incorporates the sentiments (model-2) of developers for prediction. Both models were compared using various machine learning classifiers (Zero-R, Naïve Bayes, J48, random tree and random forest). The bug reports of Firefox and Eclipse projects were used to test NRFixer. In Firefox and Eclipse projects, J48 and Naïve Bayes classifiers achieve the best prediction accuracy, respectively. It was observed that the inclusion of sentiments in the prediction model shows a rise in the prediction accuracy ranging from 2 to 5% for various classifiers.

Keywords: bug report, bug triaging, non-reproducible bugs, sentiment analysis, mining software repositories

1. Introduction

A software bug is an error or fault in a program which causes the software to behave in unintended ways. Software bugs are usually annoying and inconvenient for developers, often leading to serious consequences. Large software projects use bug tracking repositories where the users and developers report all the bugs they encounter. The developers try to reproduce the bugs with the help of information provided by a reporter in a bug report and then make the required corrections in the source code to rectify the issue. However, sometimes it is not possible to reproduce the reported bug with the information specified in a bug report. In such a scenario, the bug is marked with resolution "Non-Reproducible" or "works for me".

NR bugs account for approximately 17% of all bug reports and 3% of these bugs are later marked as fixed [1]. There could be various reasons behind this fixation of NR bugs. It may be due to any new code patch that might be made available by the reporter, user or developer which could help to reproduce the cause of a bug, or there may be various ways of fixing it. Thus the choice of the solution tested by the developer to reproduce or fix the bug could be wrong [2] and either a new solution or a new developer can reproduce and fix the NR bug.

Another reason could be that the developer had initially marked the bug as NR erroneously due to negligence or, possibly, in reluctance to reduce his or her workload. Thus at later times, a new or previously assigned developer solves the bug. If there was a mechanism which could provide information to the developer beforehand whether the bug report currently marked as NR would be fixed in the future or not, it would not only provide insights to a triager, but also help developers to predict if a bug report marked as NR could be fixed in the future or not. This would save time, effort and cost incurred in those NR bugs in the case of which there is low probability of fixing. With the use of such a mechanism, developers and a triager can actually devote their precious time and efforts to those bugs that are regarded as fixable by the proposed mechanism. This would also raise the level of interest among developers towards NR bugs.

The objective of this work is to establish if it is possible that a bug report, currently marked as NR, will get fixed in the future or not. Thus, the investigation of bug reports is carried out at two different levels. At the first level, the sentiments of comment messages in NR bugs are mined to investigate whether there is any difference between the sentiments of NR-to-fix bug reports and NR bug reports that do not get fixed. At the second level, the NRFixer framework that predicts the probability of NR bug fixation is proposed. NRFixer is evaluated with two dimensions. The first dimension considers the meta-fields of bug reports, such as a component, hardware, a platform, etc., to develop a prediction model-1. Another dimension additionally incorporates sentiments along with the existing meta-fields of a bug report to develop prediction model-2. In this work, the investigations were carried out with reference to six research questions (RQs) to attain two research objectives (ROs). RO1 investigates the sentiment analysis of bug reports using (RQ1–RQ4), whereas RO2 examines the performance of NRFixer using (RQ5–RQ6).

Research Objective 1 (RO1): Exploring sentiments in bug reports.

– RQ1. Do sentiments exist in the NR bug reports?
– RQ2. What is the difference between sentiments of reproducible (R) bugs and NR bugs?
– RQ3. Do the sentiments of developers vary in different categories of NR bugs?
– RQ4. Compare the developer's sentiments for the bug report passing through the stages: 'Newbug-to-NR' and 'NR-to-fix'?

Research Objective 2 (RO2): The fixability prediction of NR bugs.

– RQ5. What is the probability of NR fixing with the use of different classifiers?
– RQ6. Does the inclusion of the category of an NR bug and the sentiments of developers affect the accuracy of a prediction model?

For experimental evaluation, bug reports extracted from Eclipse and Firefox projects of Bugzilla repository were used to gauge the presence of sentiments. Bugzilla is the most popular open source bug repository used by different popular projects, such as Firefox, Eclipse, Linux, etc. Both prediction models (model-1 and model-2) were evaluated using various machine learning classifiers. It was observed that model-1 achieved an accuracy of 70.2% for Firefox and 66.4% for the Eclipse project. The inclusion of sentiments (model-2) further achieved an increase of 2–5% in precision values.

The remainder of this paper is structured as follows. Section 2 illustrates the related work. Section 3 discusses certain preliminaries. Section 4 presents the proposed architecture for NRFixer. Section 5 provides experimental details. Section 6 presents the results of experimental evaluation. Section 7 discusses various threats to validity. Finally, section 8 concludes the paper and discusses future directions for research.

2. Background

This section presents the previous works closely related to the areas: a) Sentiment analysis of bug reports and b) Prediction models in bug repositories.

2.1. Sentiment analysis of bug reports

Sentiment analysis is becoming an important area in the field of natural language analysis.

It comprehends the natural language processing, text analysis, computational logistics with the human psychology to gain an individual's attitude to or feeling about a particular situation or product. It is the "task of identifying positive and negative opinions, emotions and evaluations" [3]. Jurado et al. [4] confirmed that developers do leave sentiments in the textual units of issue repositories. Murgia et al. [5] analysed the existence of emotions in software artefacts, such as issue reports. Their finding confirms the presence of various emotions, such as joy, love, surprise, anger, sadness and fear in issue reports. They also reported that emotions have an impact on software development activities, such as bug fixing. A developer possessing negative emotions may not be able to fix the bug and, thus, it should be assigned to some other developer. Tourani et al. [6] evaluated automatic sentiment analysis in open source mailing lists of the Apache project. The manual study of the emails performed by them contains 19.77% positive sentiments and 11.27% contains negative sentiments. Garcia et al. [7] presented a case study on the Gentoo project of the Bugzilla repository to mine the role of emotions in the contributor's activities. Their study found that a contributor become inactive after experiencing strong positive or negative emotions.

Pletea et al. [8] gauged the presence of emotions in the security related discussions on the GitHub repository. They found that more negative emotions are expressed in security related discussions than in other discussions. The results obtained reflect the reluctance of developers towards the sensitive issue of security. Guzman et al. [9] analysed the sentiments of commit comments in the GitHub repository with respect to four parameters: programming language, day of the week and time of writing the comment, geographic distribution of a team and project approval. Destefanis et al. [10] showed that politeness in developers' comments affects the time to fix an issue.

2.2. Prediction models in bug repositories

As for the prediction model, Garcia et al. [11] built a model to predict blocking bugs. This work is similar to the work on using various machine learning classifiers for prediction. They utilized 14 different parameters to discriminate between blocking and non-blocking bugs and then compared the efficiency of a decision tree, Naïve Bayes, kNN, random forest and Zero-R classifier. They achieved an F-measure of 15–42% by tenfold cross validation on various different bug datasets. The prediction analysis described in this paper is similar to their work. However, there is a difference in the manner of predicting the NR bugs that may get fixed in the future.

Shihab et al. [12] addressed the nature of bugs that get reopened. They used 22 different factors categorized under four dimensions: (1) the work habits dimension, (2) the bug report dimension, (3) the bug fix dimension, and (4) the team dimension. Their model achieved a precision of 52.1% to 78.6% and a recall of 70.5% to 94.1% when predicting whether a bug will be reopened or not. They also found a comment text and the last status to be the most influential factors for predicting the possibility of reopening. Hewett et al. [13] predicted the time required to repair software defects. Their model achieved an accuracy of 93.44% on medical software system dataset. Guo et al. [14] proposed a statistical model to predict the possibility of fixing a newly arrived bug. Their model achieved 68% precision and 64% recall on the Windows Vista project. They further validated their model by conducting a survey among 1773 Microsoft employees. Zimmermann et al. [15] also investigated and characterized the reopened bugs in Microsoft Windows to find the possible causes of reopening bugs and their impact.

3. Preliminaries

This section summarizes the basic information about a bug report, sentiment analysis technique and various machine learning classifiers used in this paper.

3.1. Bug report

A bug report is a document containing complete specification related to a bug. A bug report may be created by an end user, developer or beta tester of the software project. A bug report constitutes various predefined meta-fields and free form textual contents. The predefined meta-fields include bug id, product, component, operating system, platform, milestone, severity, version, status, resolution, reporter, reported date and time, assigned to, etc. The textual contents include keywords, summary (or tagline), description and comments. "Summary" refers to the one-line short definition about the bug. "Description" refers to the complete detailed specification submitted by the reporter regarding the submitted bug. It forms the main body of the bug report that generally incorporates the steps to reproduce the issue. "Comments" refers to the open discussion by a group of people to discuss and review the solutions for the reported bug. This group of people generally comprehends some expertise in the related area of the bug.

3.2. Sentiment analysis

Sentiment analysis is a technique to extract, identify or characterize the sentimental content of a text unit. It assigns a quantitative value representing the contextual polarity of the text. To analyse the sentiments in bug reports natural language text processing (NLTK) toolkit was used [16]. NLTK takes a text unit as an input and performs a two-level classification. Level 1 determines whether the text is neutral or polar. A text unit may or may not contain sentiments. If the probability of the lack of sentiment is greater than 0.5, the text is labelled as neutral. Otherwise, if the probability of the presence of sentiments is greater than 0.5, the text is labelled as polar, and the second level classification is performed to determine whether the text expresses positive or negative sentiment. The label with higher probability is finally assigned to the input text.

3.3. Machine learning classifiers

1. **Zero-R**: Zero-R (or no rule classifier) is the simplest classification algorithm. It always predicts the majority class present in the training dataset. Although it has no predictability power, it is useful in the determination of the baseline performance as a benchmark. In this study, during each fold of cross validation, the Zero-R classifier predicts the majority class among NR-to-NR and NR-to-fix classes during that fold. The individual fold efficiencies related to NR-to-fix class are aggregated to compute the overall efficiency of the Zero-R classifier for NR-to-fix class.
2. **Naïve Bayes**: Naïve Bayes [17] is a simple probabilistic classification algorithm based on Bayes' theorem. It classifies a new record $x = < x_1, \ldots, x_p >$ to class k that maximizes the conditional probability:

$$P(C = k/X) = < x_1, \ldots, x_p >$$

Under the assumption that the factors are randomly independent of each other, the Naïve Bayes classifier can be re-written as: Here, $P(C = k)$ is known as the class prior probability and can be approximated with the percentage of training files marked with label k. The likelihood or conditional probability $P(x_i/C = k)$ can be estimated with $N_k * i/N_k$, where the numerator is the number of records marked with label k for which the i_{th}-factor is equal to and the denominator is the number of records regarded with label k. The probability $P(X = x)$ is the predictor of the prior probability and is constant with respect to the different classes.
3. **J48**: J48 is the open source Java implementation of the C4.5 algorithm [18] in the weka data mining toolkit. C4.5 builds decision trees from a set of training data in the same way as ID3, using the concept of information gain and entropy. The training data is a set

$$S = s1, s2, \ldots$$

of already classified samples. Each sample consists of a p-dimensional vector

$$(x_{(1,i)}, x_{(2,i)}, \ldots, x_{(p,i)}),$$

where the x_j represent attribute values or features of the sample, as well as the class in which falls. At each node of the tree, C4.5

algorithm selects the attribute of the data that most efficiently splits the samples into subsets enriched in one class or the other. The criterion for splitting is the largest normalized information gain. The attribute with the highest normalized information gain is selected to build the decision. The C4.5 algorithm is then run recursively on smaller sub lists until data are classified [17].

4. **Random Forest**: Random forest [19, 20] is an ensemble learning algorithm for data classification. It is a meta estimator that makes the prediction based on the majority vote of the multitude of decision trees. This classification algorithm reduces the variance of the individual decision trees and makes the classifier more resilient to noise in the training data set. For constructing the Random Forests of m decision trees, m bootstrap samples are generated from the training data set and each of them is utilized for training a decision tree.
5. **Random Tree**: Random decision tree algorithm builds multiple decision trees randomly. While building a decision tree, the classification algorithm picks a "remaining" feature randomly at each node without any accuracy estimation procedure (such as cross validation). A categorical attribute (such as gender) is considered "remaining" if the same categorical attribute has not been selected formerly in a particular decision path arising from the root to the current node of the tree. A continuous attribute (such as income), on the other hand, can be selected more than once in the same decision path. Every time the continuous attribute is selected, a random threshold is chosen.

4. NRFixer: proposed architecture

Objective: To find out if there is a possibility that a bug report currently marked as NR will get fixed in the future.
Input: To find out if there is a possibility that a bug report currently marked as NR will get fixed in the future.
Output: Predicted class: NR-to-fix or NR-to-NR.

Proposed approach: The proposed approach uses bug reports currently marked as NR to predict whether it will be fixed in the future or not. Two prediction models were developed and are compared. To develop the prediction model-1, as shown in Figure 1, it extracts eight bug report meta-fields such as product, component, hardware, severity, priority, cc count, number of comments and keywords to train machine learning classifiers (Zero-R, Naïve Bayes, J48, random forest and random tree). Prediction model-2 additionally uses the extracted parameters namely developer's sentiments and NR bug category along with the eight meta-fields considered in model-1 to train the machine learning classifiers and predicts the class label (NR-to-fix or NR-to-NR).

5. Experimental details

In this section, the experimental details for the prediction of NR-to-fix bugs are presented. For the Eclipse and Firefox projects, various bug report meta-fields were used for prediction and five different classifiers were compared. The classifiers are Zero-R, Naïve Bayes, J48, random forest and random tree. In this experiment a weka toolkit was used. The NR bugs were investigated and the probability of their fixation was predicted.

5.1. Dataset

The data for this study were extracted from the dataset used by Joorabchi et al. [1]. The bug reports of the Firefox and Eclipse projects were used for experimentation. The sentiments of a total of 419 NR bug reports containing 4250 text units were analysed. In the dataset, a single bug report contains a varying number of comment messages which ranges from 1 to 83.

5.2. Experiment parameters

Various meta-fields of bug reports were considered to determine the fixable NR bugs. All factors are listed below:

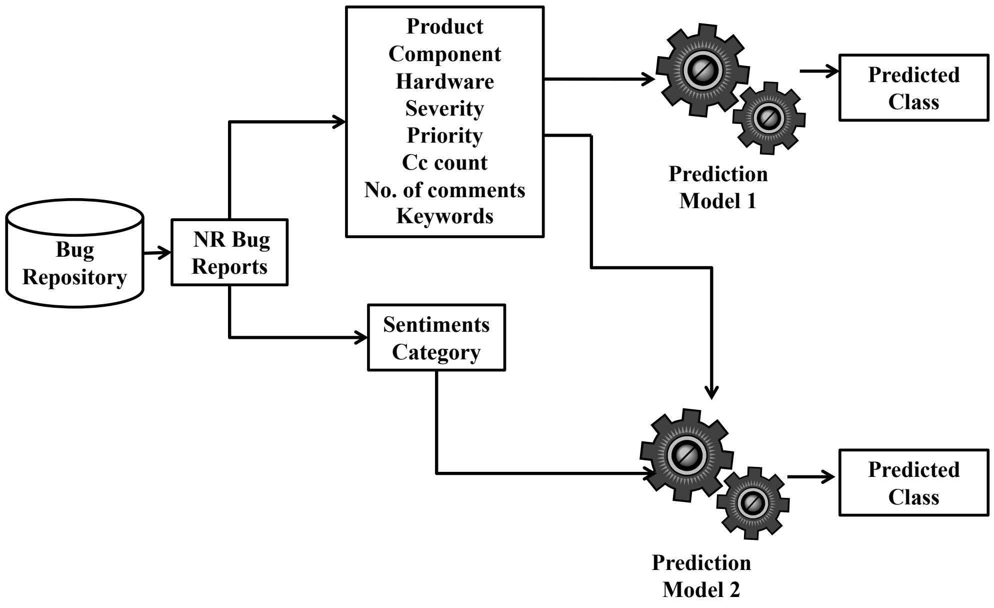


Figure 1. NRFixer: Prediction model for NR bugs

1. **Product**: It refers to the general area the bug belongs to.
2. **Component**: It refers to the second level categorization of a product.
3. **Hardware**: It indicates the computing environment where the bug originated.
4. **Severity**: It describes the impact of the bug. This field offers options, such as severe, normal and minor.
5. **Priority**: The priority field is used to prioritize bug reports. The values of the priority field ranges from P1 to P5 (P1 being the highest priority and P5 being the lowest priority value).
6. **CC Count**: It defines the number of developers in the cc list of the bug report. It is usually the number of developers participating in the bug report.
7. **No. of comments**: It refers to the number of comments made by the developers in order to resolve the bug.
8. **Keywords**: It refers to the tags or categorization of bug reports.

Two more parameters were added in the prediction model:

1. **Sentiment**: It refers to the positive, negative or neutral value expressed by a textual unit.
2. **Category**: It refers to the cause category of the bug. The non-reproducible bugs are classified into six cause categories namely, inter-bug dependencies, environmental differences, insufficient information, conflicting expectations, non-deterministic behaviour and others.

Once the aforementioned factors are extracted, they are used to train various machine learning classifiers in order to predict the fixable bugs from the currently NR marked bugs. The different classifiers used for the prediction of NR in the case of fixed bugs are discussed in the following section.

5.3. Evaluation metrics

To evaluate the performance of the prediction model, standard performance evaluation metrics was used: precision, recall and F-measure.

1. **Precision:** It refers to the fraction of relevant instances retrieved from the total instances that are retrieved.

$$Precision = \frac{tp}{tp + fp}$$

2. **Recall:** It refers to the fraction of relevant instances retrieved from the total relevant instances.

$$Recall = \frac{tp}{tp + fn}$$

3. **F-Measure:** It refers to the harmonic mean of precision and recall.

$$F\text{-}measure = \frac{2 * Precision * Recall}{Precision + Recall}$$

In the equations, tp represents the number of positive samples correctly predicted, fp represents the number of negative samples incorrectly predicted as positive, tn represents the number of positive samples incorrectly predicted and fn represents the number of negative samples correctly predicted. Table 1 shows the confusion matrix.

The tenfold cross validation technique was used in weka to measure the efficiency of NRFixer. When only a limited amount of data is available, cross validation is used to attain an unbiased estimation of model performance. In k-fold cross-validation, data is divided into k subsets of equal size. Thus the model is built k times, each time using sets of data for training the classifier and leaving out one subset as a test set. In order to reduce the impact of the class imbalance problem, the dataset re-sampling technique was used.

6. Experimental results

This section addresses the experimental results of the six identified research questions (RQs). The RQs are classified under two research objectives (RO). RO1 explores the sentiments in bug reports and RO2 investigates the performance of NRFixer.

6.1. Research objective 1 (RO1): exploring sentiments in bug reports

This subsection answers four RQs (RQ1–RQ4) which investigate the sentiments of developers. Joorabchi et al. [1] claimed that a large proportion of bug reports are marked as NR but a small part of these NR bugs (approximately 3%) are fixed later. An empirical evaluation of the sentiments of bug reports was conducted to investigate the developer's behaviour towards R, NR and NR-to-fix bugs. The investigation results of the sentiment analysis of bug reports are presented as below.

RQ1. Do sentiments exist in the NR bug reports? The analysis encompassed 419 NR bug reports to detect whether bug report discussions contain any sentiments or not. These reports constituted a total of 4250 text units (419 taglines, 419 descriptions and 3412 commit comment messages written by software developers). A tagline contains 5 to 10 words and offers a short summary of the bug. A description contains detailed information about the bug and usually constitutes the steps required to reproduce the issue. Commit comment messages contain the developer's discussions regarding the steps that may be useful in bug fixation. The statistics for the sentiment analysis are shown in Figure 2. The percentage of the analysed text units which had either positive or negative sentiments was 65.66%. This confirms the existence of sentiments in software artefacts, such as issue reports as stated by [5].

On the other hand, 34.32% text units exhibited no sentiments. In particular, the tagline field exhibits 67.06% neutral sentiments. This is because a tagline is a 5 to 10-word description of the bug report and thus it is difficult for any sentiment analysis tool to extract the polarity of sentiment in such a small amount of text. Similarly, the description and comments exhibit 39.85% and 29.63% neutral sentiments, respectively. This is due to the fact that these fields sometimes may contain a lot of technical code to resolve the issue and thus may lack any sentiments.

RQ2. What is the difference between sentiments of reproducible bugs and NR bugs? To address this question, two categories of bug reports were considered: the bug reports which are marked as NR at least once in their lifetime and fixed bug reports which are never marked as NR, which are termed as R. For this step, 200 R bug reports were considered (two bug reports from this set contained textual units in languages other than English and thus were removed). Finally 198 R bug reports containing 1556 text units (198 taglines, 198 descriptions and 1160 commit comment messages written by software developers) were studied in addition

Table 1. Confusion Matrix

	Positive (p)	Negative (n)
Positive (p)	*tp*	*fp*
Negative (n)	*fn*	*tn*

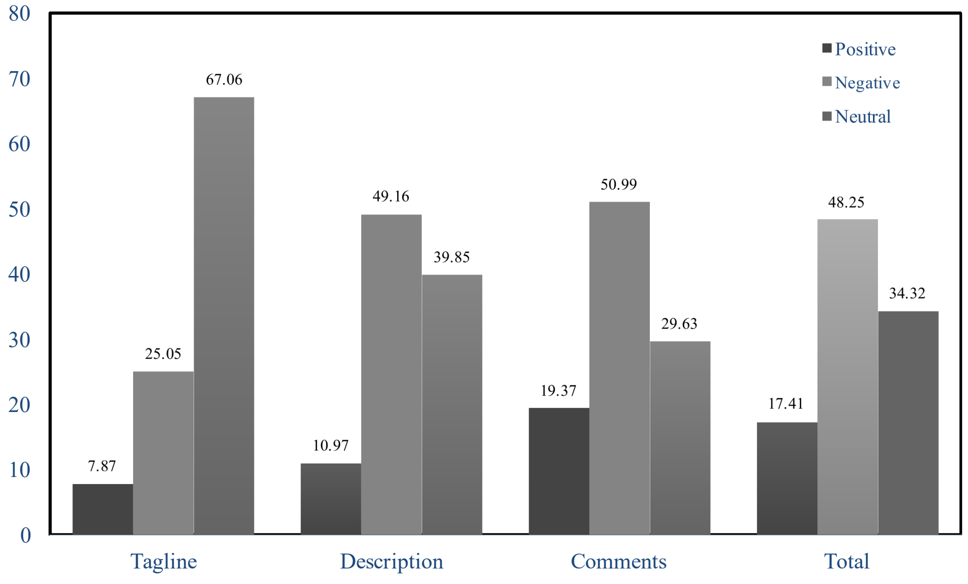


Figure 2. Sentiment distribution in NR bug reports

to 419 NR bug reports (containing 4250 text units). Table 2 presents the statistics of sentiments in reproducible and NR bug reports. The results show that the fraction of negative sentiments is higher in NR bugs (48.25%) than the reproducible bugs (29.24%). Also the fraction of positive sentiments is lower in NR bugs (17.41%) than the reproducible bugs (20.24%). The results confirm that developers have negative sentiments while solving NR bugs. This may be due to the smaller probability of fixing NR bugs.

RQ3. Do the sentiments of developers vary in different categories of NR bugs? To examine this research question, the sentiments of 419 NR bug reports category wise were analysed. The NR bug reports are categorized into six possible cause categories: conflict of knowledge (32 bug reports containing 320 text units), dependent bugs (83 bug reports containing 979 text units), environmental settings (129 bug reports containing 1192 text units), non-deterministic bugs (16 bug reports containing 149 text units), precise information required (113 bug reports containing 1205 text units) and others (46 bug reports containing 405 text units). The statistics for sentiment analysis is shown in Table 3. It was found out that negative sentiments dominated in all six categories of NR bugs. The negative sentiments appear 29.8%–36.24% more than positive sentiments in different cause categories. Among the various categories, the environmental setting contains maximum positive textual units (18.62%) and the category others contains minimum negative textual units (37.17%).

RQ4. Compare the developer's sentiments for the bug report passing through the stages 'Newbug-to-NR' and 'NR-to-fix'? To investigate NR-to-fix bugs, 100 bug reports containing 1648 textual units (100 taglines, 100 descriptions and 1448 textual comments) were considered, they were initially marked from resolution Newbug to NR and were later marked as fixed. NR bugs may go through various stages before being fixed. Stage 1 (Newbug-to-NR) depicts a bug declared as NR and stage 2 (NR-to-Fix) depicts NR being fixed. Figure 3 shows the stages a normal NR bug usually passes through.

Table 2. Percentage distribution of sentiments in various categories of NR bug reports

Bug Reports		Positive (%)	Negative (%)	Neutral (%)
Reproducible	Tagline	5.55	16.66	77.77
	Description	11.11	22.22	66.66
	Comments	24.30	32.59	43.1
	Total	**20.24**	**29.24**	**50.51**
Non-Reproducible	Tagline	7.87	25.05	67.06
	Description	10.97	49.16	39.85
	Comments	19.37	50.99	29.63
	Total	**17.41**	**48.25**	**34.32**

Table 3. Percentage distribution of sentiments in various categories of NR bug reports

Category	Positive (%)	Negative (%)	Neutral (%)
Conflict of Knowledge	16.56	52.18	31.25
Dependant Bugs	15.32	49.23	35.44
Environmental Settings	18.62	51.00	30.36
Non-Deterministic Bugs	16.10	52.34	31.54
Precise Information Required	17.75	47.55	34.68
Others	15.38	37.17	47.43

The investigation was conducted with two different perspectives. For primary investigation, the overall positive and negative percentage of sentiments were searched for in comment messages during both stages. For secondary investigation, the change in sentiments during both stages of each bug report was analysed. To address the developer's sentiments during stage 1 and stage 2, the sentiments of 1448 textual comments were analysed. Table 4 shows the statistics of sentiments at different stages of NR-to-fix bugs.

The primary observation during stage 1 (Newbug-to-NR) comprised 511 textual comments and shows that 13.89% comments have positive sentiments whereas 57.73% comments have negative sentiments. During stage 2 (NR-to-fix), 937 textual comments were analysed. At stage 2, the positive percentage increased to 19.10% whereas negative percentage declined to 47.91%. Thus, it was observed that during stage 2, the positive percentage of sentiments increased by 6% and the negative decreased by 10% as compared to the stage 1. This incline in positivity and decline in negativity reflect the enhanced confidence of developers towards bug reports during the NR-to-fix stage and this optimism leads to fixing NR bugs.

For the secondary investigation, each bug report was analysed to find the change in the positive and negative percentage of sentiments during both stages. It was observed that there is an opinion drift in the sentiments of bug reports in Newbug-to-NR and NR-to-fix stages. The statistics for opinion drift in sentiments during the Newbug-to-NR and NR-to-fix stages is shown in Table 5. This investigation result shows that overall in 71% bug reports either the negativity decreased or positivity increased. It was inferred that during the initial stage of triaging, the developers were reluctant to solve the bug, but this reluctance decreased and as a result NR bugs were fixed.

Investigation summary (RQ1–RQ4). In RO1, four investigations were conducted in the context of the sentiment analysis of bug reports. The investigations confirm that bug reports do express sentiments. The textual units of NR bugs are more inclined towards negative sentiments as compared to reproducible bugs. It has been also found that there is an opinion drift between the Newbug-to-NR and NR-to-fix stages. There

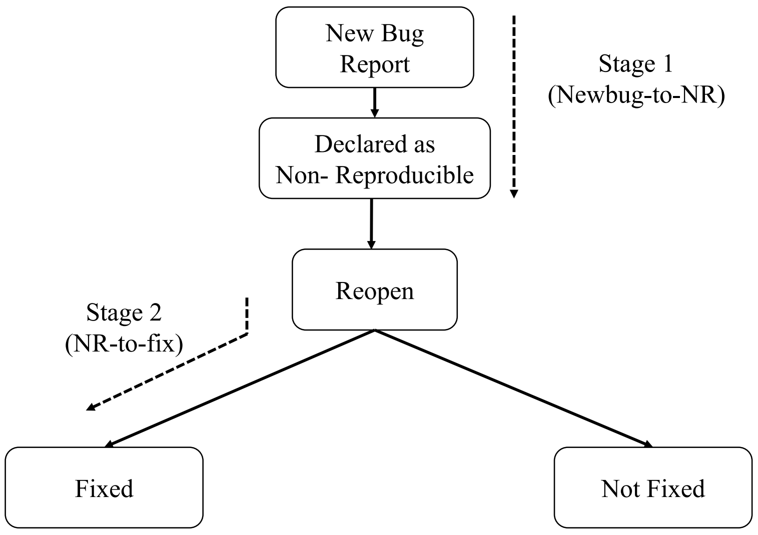


Figure 3. Stages of NR bugs

Table 4. Percentage distribution of sentiments in two different stages of bug reports: Newbug-to-NR and NR-to-fix

Stage	Positive (%)	Negative (%)	Neutral (%)
Newbug-to-NR(Stage 1)	13.89	57.73	28.18
NR-to-fix(Stage 2)	19.10	47.91	32.87

Table 5. Accuracy of NRFixer using various meta-fields of bug report

Opinion drift	% age of bug reports in which Positivity increases	% age of bug reports in which Negativity decreases	Total % age of bugs which observed change
Newbug-to-NR and NR-to-fix (before and after declaring NR)	46%	62%	71%

is a significant drift towards increasing positivity or decreasing negativity in the sentiments of NR bugs during stage NR-to-fix as compared to the Newbug-to-NR stage. This confirms the reluctant behaviour of developers while marking the resolution of a bug report as NR.

The observation of the sentiments of NR bugs highly inclines towards the positive side, it has been inferred that we need an automated way (i.e. prediction model) for judging those NR bugs that have high chances of being fixed. Further, the sentiments can be incorporated to enquire the prediction model's behaviour. This prediction model will not only build the confidence of developers, but will also save time, effort and cost incurred in debugging those bugs which are less likely to be fixed. Thus the NRFixer framework was proposed.

6.2. Research objective 2 (RO2): prediction of the fixability of NR bugs

In this subsection, the performance of NRFixer is investigated (refer to RQ5–RQ6). The results of the research questions addressed in this study are also presented by comparing the performance of five different classifiers: Zero-R, Naïve Bayes, J48, random forest and random tree. In this study, 419 NR bug reports containing 4250 textual units were considered for experimentation. Among these bug reports, 319 bug reports containing 2602 textual units are marked as NR-to-NR (170 bug reports for Mozilla project and 149 bug reports for the eclipse project) whereas 100 bug reports (containing 1648 textual units) are marked as NR-to-fix (50 bug reports

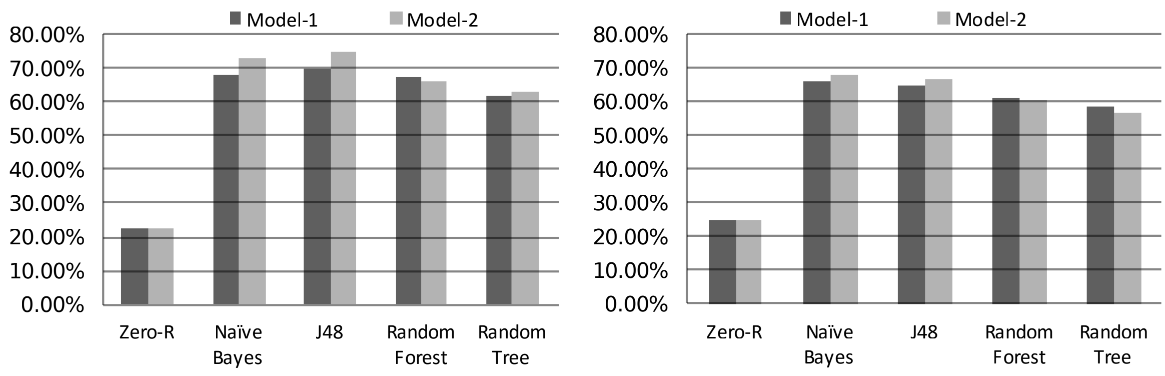


Figure 4. Comparing the accuracy of both models for different classifiers

for the Mozilla project and 50 bug reports for the eclipse project).

RQ5. What is the probability of NR fixing with the use of different classifiers? To address this research question, the performance for prediction model 1 is evaluated. Table 6 presents the precision, recall and F-measure achieved by various machine learning classifiers while using the meta-fields of bug reports, namely product, component, hardware, severity, priority, cc count, number of comments and keywords. For the Firefox project, J48 presents the best precision, recall and F-measure values and the component was found to be the most influencing factor. For the Eclipse project, Naïve Bayes gives the best precision, recall and F-measure values. In this project, severity was found to be the most influencing factor.

RQ6. Does the inclusion of the category of an NR bug and the sentiments of developers affect the efficiency of a prediction model? To address this question, the performance of prediction model-2 in NRFixer is evaluated. Table 7 presents the precision, recall and F-measure achieved by various classifiers using the meta-fields of a bug report, namely product, component, hardware, severity, priority, cc count, number of comments and keywords along with the extracted parameters such as the sentiments of developers and the category of NR bugs. For prediction model-2, the J48 classifier presents the best precision value in the Firefox project and the component was found to be the most influencing factor. Similarly, for the Eclipse project, the Naïve Bayes classifier gives the best precision, recall and F-measure values and severity was found to be the most influencing factor.

The investigation in RO2 confirms that based on bug report meta-fields and sentiment related parameters, it is possible to predict whether the NR bug will get fixed in the future or not. Among different projects, Naïve Bayes and J48 machine learning classifiers achieved the best prediction performance. Taking into consideration the precision metric, J48 is the most suitable classifier for the Firefox project and the Naïve Bayes classifier is the most suitable one for predictions in the Eclipse project. Figure 4 depicts the comparison of accuracy for both prediction models. The inclusion of the sentiments and category of non-reproducible bugs presents better precision of 2–5%.

7. Threats to validity

In this section, we present the various internal and external threats to validity in this work.

External validity. In this work, the bug reports used in experimental evaluation were collected from two popular projects, Firefox and Eclipse of open source bug tracking repository, Bugzilla. Data collected from these projects may vary from other open and closed source projects. Therefore, the outcomes from this study may not generalize well to other commercial software projects. Additional studies are required for other closed source projects or projects that use different software processes. Although we have examined large open source projects which cover a wide range of products, there may be other projects which use different software processes. Thus, the results may not generalize to all of them.

Table 6. Accuracy of NRFixer using various meta-fields of bug report

Project	Classifier	Precision	Recall	F-measure
Firefox	Zero-R	22.7%	22.7%	22.7%
	Naïve Bayes	68%	67.1%	66.6%
	J48	**70.2%**	**70.1%**	**70%**
	Random Forest	67.3%	67.3%	67.3%
	Random Tree	61.8%	61.7%	61.7%
Eclipse	Zero-R	25.2%	25.2%	25.2%
	Naïve Bayes	**66.4%**	**65.2%**	**65.79%**
	J48	65.1%	64.9%	64.99%
	Random Forest	61.3%	61.2%	61.1%
	Random Tree	58.6%	58.6%	58.6%

Table 7. Accuracy of NRFixer using various meta-fields of bug report, sentiments of developers and category of NR bugs

Project	Classifier	Precision	Recall	F-measure
Firefox	Zero-R	22.7%	22.7%	22.7%
	Naïve Bayes	72.9%	**77.5%**	**75.12%**
	J48	**74.7%**	73%	73.84%
	Random Forest	66%	66%	66%
	Random Tree	62.8%	62.5%	62.3%
Eclipse	Zero-R	25.5%	25.2%	25.2%
	Naïve Bayes	**68%**	**65.3%**	**66.62%**
	J48	66.9%	65%	65.6%
	Random Forest	60.5%	60.5%	60.5%
	Random Tree	57%	57%	57%

Internal validity. In this work, it was assumed that the data obtained from a bug repository are optimal. However, there is a possibility of errors or noise in the extracted data, which may affect the results of this study. To mitigate this threat, the bug reports used in this study were obtained from the most widely used projects of the Bugzilla repository. These projects are long lived and are actively maintained, hence it is safe to assume that the extracted data are acceptable (if not optimal). Also, the used dataset suffers from a class imbalance problem and so the re-sampling of dataset was used to overcome this effect. Moreover, the categories of NR bugs may be erroneous. But since the studies related to NR bugs are in their initial phase, this threat was considered to be minor and careful cross-checks of the data and the technique was conducted to eliminate errors in the best possible way.

Five different classifiers were explored: Zero-R, Naïve Bayes, J48, random forest and random tree for the performance evaluation of NRFixer. However, there are many other classifiers (such as genetic algorithms [21], neural network, etc.) and ensemble based techniques [18] (such as stacking, bagging, boosting) which have not been explored in this work. Nevertheless, it is possible that a different set of algorithms would provide better results for NR-to-fix bug prediction as compared to the set of algorithms explored in this work.

Further, in this work, the python NLTK toolkit was utilized [16] for the sentiment analysis of textual contents in bug reports. However, there are many other toolkits available for sentiment analysis, such as SentiStrength, Stanford NLP, etc. Jongeling et al. [22] evaluated various sentiment analysis tools for software engineering studies. They observed that sentiments obtained from various tools neither agree with each other, nor with manual labelling. They suggested a need for a sentiment analysis tool that specifically caters to the software engineering domain. Therefore, the results of this study may improve with the domain specific tailoring of sentiment analysis. But since such a toolkit is not available, we are considering one of the most popular sentiment analysis toolkit, NLTK for experimentation.

8. Conclusion and future work

Non-reproducible bugs are generally frustrating for developers due to the uncertainty of their fixation. To minimize this uncertainty, we have mined the sentiments of textual data present in the non-reproducible bug reports. Mining is done for both categories of bug reports NR-to-fix and NR-to-NR. It was done to find out any clue to assist the developer during the initial stages of bug triaging. The study is being carried out at two different levels. At the first level, the sentiments of bug reports were mined and at the second level framework NRFixer which predicts the probability of NR bug fixation is proposed.

The first level of the study confirms that bug reports do express sentiments. It was found out that the developers possess more negative sentiments for non-reproducible bugs than reproducible bugs. As long as bugs are marked as non-reproducible, the percentage of negative sentiments is 66% bigger than the reproducible bugs. This confirms the reluctant behaviour of developers towards the non-reproducible bugs. It was also found out that there is a major opinion drift found in the sentiments of NR-to-fix bugs. In 71% NR-to-fix bug reports, either there was a decrease in the percentage of negative comments or an increase in the percentage of positive comments when the bug is reopened and is near fixation. This reveals that the developers may have marked the bug as non-reproducible erroneously or it could be due to the lack of some code patch.

The second level of the study deals with the prediction of the possibility of non-reproducible bugs getting fixed using NRFixer. It is evaluated with two dimensions. In the first dimension, we considered various meta-fields of bug report (prediction model-1). In the second dimension, the sentiments of developers were additionally incorporated along with the existing meta-fields of a bug report (prediction model-2). The results of this investigation show that NRFixer could efficiently predict bugs marked as non-reproducible. It was observed that the inclusion of sentiments in prediction model-2 shows an additional rise in the prediction accuracy ranging from 2 to 5% for various classifiers.

For future work, it is planned to explore NRFixer on more machine learning classifiers and software projects, such as closed source applications. Work will also be conducted on improving the performance of NRFixer using ensemble based machine learning techniques. There are also plans to perform a qualitative analysis on domain specific NR-to-fix bug prediction using different textual factors of bug reports. In addition to this work, a fix suggestion tool for non-reproducible bugs that could be fixed will be built.

References

[1] M. Erfani Joorabchi, M. Mirzaaghaei, and A. Mesbah, "Works for me! characterizing non-reproducible bug reports," in *Proceedings of the 11th Working Conference on Mining Software Repositories.* ACM, 2014, pp. 62–71.

[2] E. Murphy-Hill, T. Zimmermann, C. Bird, and N. Nagappan, "The design space of bug fixes and how developers navigate it," *IEEE Transactions on Software Engineering*, Vol. 41, No. 1, 2015, pp. 65–81.

[3] T. Wilson, J. Wiebe, and P. Hoffmann, "Recognizing contextual polarity in phrase-level sentiment analysis," in *Proceedings of the Conference on Human Language Technology and Empirical Methods in Natural Language Processing.* Association for Computational Linguistics, 2005, pp. 347–354.

[4] F. Jurado and P. Rodriguez, "Sentiment analysis in monitoring software development processes: An exploratory case study on GitHub's project issues," *Journal of Systems and Software*, Vol. 104, 2015, pp. 82–89.

[5] A. Murgia, P. Tourani, B. Adams, and M. Ortu, "Do developers feel emotions? an exploratory

analysis of emotions in software artifacts," in *Proceedings of the 11th working conference on mining software repositories*. ACM, 2014, pp. 262–271.

[6] P. Tourani, Y. Jiang, and B. Adams, "Monitoring sentiment in open source mailing lists: Exploratory study on the apache ecosystem," in *Proceedings of 24th Annual International Conference on Computer Science and Software Engineering*. IBM Corp., 2014, pp. 34–44.

[7] D. Garcia, M.S. Zanetti, and F. Schweitzer, "The role of emotions in contributors activity: A case study on the Gentoo community," in *The Third International Conference on Cloud and Green Computing (CGC)*. IEEE, 2013, pp. 410–417.

[8] D. Pletea, B. Vasilescu, and A. Serebrenik, "Security and emotion: Sentiment analysis of security discussions on GitHub," in *Proceedings of the 11th working conference on mining software repositories*. ACM, 2014, pp. 348–351.

[9] E. Guzman, D. Azócar, and Y. Li, "Sentiment analysis of commit comments in GitHub: An empirical study," in *Proceedings of the 11th Working Conference on Mining Software Repositories*. ACM, 2014, pp. 352–355.

[10] G. Destefanis, M. Ortu, S. Counsell, S. Swift, M. Marchesi, and R. Tonelli, "Software development: Do good manners matter?" *PeerJ Computer Science*, Vol. 2, 2016, p. e73.

[11] 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*. ACM, 2014, pp. 72–81.

[12] E. Shihab, A. Ihara, Y. Kamei, W.M. Ibrahim, M. Ohira, B. Adams, A.E. Hassan, and K. Matsumoto, "Studying re-opened bugs in open source software," *Empirical Software Engineering*, Vol. 18, No. 5, 2013, pp. 1005–1042.

[13] R. Hewett and P. Kijsanayothin, "On modeling software defect repair time," *Empirical Software Engineering*, Vol. 14, No. 2, 2009, p. 165.

[14] P.J. Guo, T. Zimmermann, N. Nagappan, and B. Murphy, "Characterizing and predicting which bugs get fixed: An empirical study of Microsoft Windows," in *ACM/IEEE 32nd International Conference on Software Engineering*, Vol. 1. IEEE, 2010, pp. 495–504.

[15] T. Zimmermann, N. Nagappan, P.J. Guo, and B. Murphy, "Characterizing and predicting which bugs get reopened," in *Proceedings of the 34th International Conference on Software Engineering*. IEEE Press, 2012, pp. 1074–1083.

[16] Python NLTK sentiment analysis with text classification demo. [Online]. http://text-processing.com/demo/sentiment/ [Accessed September 2016].

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

[18] J.R. Quinlan, *C4.5: programs for machine learning*. Elsevier, 2014.

[19] L. Breiman, "Random forests," *Machine learning*, Vol. 45, No. 1, 2001, pp. 5–32.

[20] J. Han, J. Pei, and M. Kamber, *Data mining: concepts and techniques*. Elsevier, 2011.

[21] M. Mitchell, *An introduction to genetic algorithms*. MIT press, 1998.

[22] R. Jongeling, S. Datta, and A. Serebrenik, "Choosing your weapons: On sentiment analysis tools for software engineering research," in *IEEE International Conference on Software Maintenance and Evolution (ICSME)*. IEEE, 2015, pp. 531–535.

e-Informatica Software Engineering Journal, Volume 11, Issue 1, 2017, pages: 117–141, DOI 10.5277/e-Inf170106

Machine Learning or Information Retrieval Techniques for Bug Triaging: Which is Better?

Anjali Goyal[*], Neetu Sardana[*]
[*]*Jaypee Institute of Information Technology, Noida, India*
anjaligoyal19@yahoo.in, neetu.sardana@jiit.ac.in

Abstract

Bugs are the inevitable part of a software system. Nowadays, large software development projects even release beta versions of their products to gather bug reports from users. The collected bug reports are then worked upon by various developers in order to resolve the defects and make the final software product more reliable. The high frequency of incoming bugs makes the bug handling a difficult and time consuming task. Bug assignment is an integral part of bug triaging that aims at the process of assigning a suitable developer for the reported bug who corrects the source code in order to resolve the bug. There are various semi and fully automated techniques to ease the task of bug assignment. This paper presents the current state of the art of various techniques used for bug report assignment. Through exhaustive research, the authors have observed that machine learning and information retrieval based bug assignment approaches are most popular in literature. A deeper investigation has shown that the trend of techniques is taking a shift from machine learning based approaches towards information retrieval based approaches. Therefore, the focus of this work is to find the reason behind the observed drift and thus a comparative analysis is conducted on the bug reports of the Mozilla, Eclipse, Gnome and Open Office projects in the Bugzilla repository. The results of the study show that the information retrieval based technique yields better efficiency in recommending the developers for bug reports.

Keywords: bug triaging, bug report assignment, developer recommendation, machine learning, information retrieval

1. Introduction

The explosive growth in size and scale of software systems has led to the creation of various open source bug tracking repositories. Bug tracking repositories gather, organize and keep track of all the reported bugs. Although, a large number of bug reports help to make the final software product error free, it is really challenging for the bug triager to handle such a large volume of reported bugs. When a new bug is reported, a bug triager analyses the feasibility of bug to verify if the reported bug is not a mere duplicate and contains enough information to be reproduced. If the bug is found to be feasible, it is assigned to a developer for resolution. For effective bug resolution, it is extremely important to assign the reported bug to a suitable developer. Bug assignment is an integral part of bug triaging whose goal is the process of assigning a suitable developer to the reported bug. The assigned developer performs various checks and changes in the source code to rectify the reported issue. The selection of a suitable developer for the bug report is a challenging process as it significantly affects time and cost incurred in the project. Thus, it is imperative to make an appropriate developer assignment who is an expert in the area of the reported bug.

In the past, software projects were small in size and the count of bugs was minimal. In those days, it was possible for the bug triager to perform developer assignment manually but with passing time software projects grew in scale and size. Subsequently, software projects became more complex and in the current scenario, it has become really cumbersome for the bug triager to be aware of the expertise of all the developers in a triaging team. To ease the task of the bug triager, various semi and fully automated bug assignment approaches have been proposed in the literature. These approaches gather the information related to developer expertise from various sources and utilize it to make developer recommendations. However, the availability of a huge amount of bug assignment approaches appeals for a comprehensive overview.

At present there is no in-depth and focused survey available specifically in the area of bug triaging. It has been observed that only J. Zhang et al. [1] and T. Zhang et al. [2] reported short discussions on bug triaging in their broad category survey on bug handling. This paper performs a systematic, in-depth and focused literature survey on bug triaging. In this paper, 75 papers from peer reviewed, refereed conferences and journals published during years 2004 to 2016 are summarised in an organized manner. The existing approaches are classified into seven categories: machine learning (ML), information retrieval (IR), auction, social network, tossing graphs, fuzzy set and operational research based techniques. The authors further perform an analysis of these approaches in two perspectives: cumulative frequency distribution and year wise trend analysis. In addition, they compare the identified bug triaging techniques inferred from analytical analysis to find the best bug triaging technique.

The rest of this paper is organized as follows: Section 2 presents the anatomy of a bug report and its life cycle. Section 3 describes the systematic survey process. Section 4 reviews the work on bug report assignment and presents a comparative study on two most popular bug assignment techniques. Section 5 concludes this paper and provides some interesting future research directions.

2. Anatomy of a bug report

A bug report is a detailed record constituting a full description related to a bug discovered in any software. It is generally created by the customers, users, developers or testers of software system. A decent bug report ought to comprise three underlying components:

1. Steps to replicate the bug.
2. What is the reporter expected to see?
3. What did the reporter actually see?

A bug report constitutes a collection of various categorical and free form textual data. The categorical data (or meta-fields) constitute fields, such as bug id, product, component, resolution, status, version, priority, creation date, operating system. The free form textual fields contain keywords, summary, description and comments posted by the developers for discussing a probable solution for fixing the bug report. Figure 1 shows an instance of a bug report in the Mozilla project.

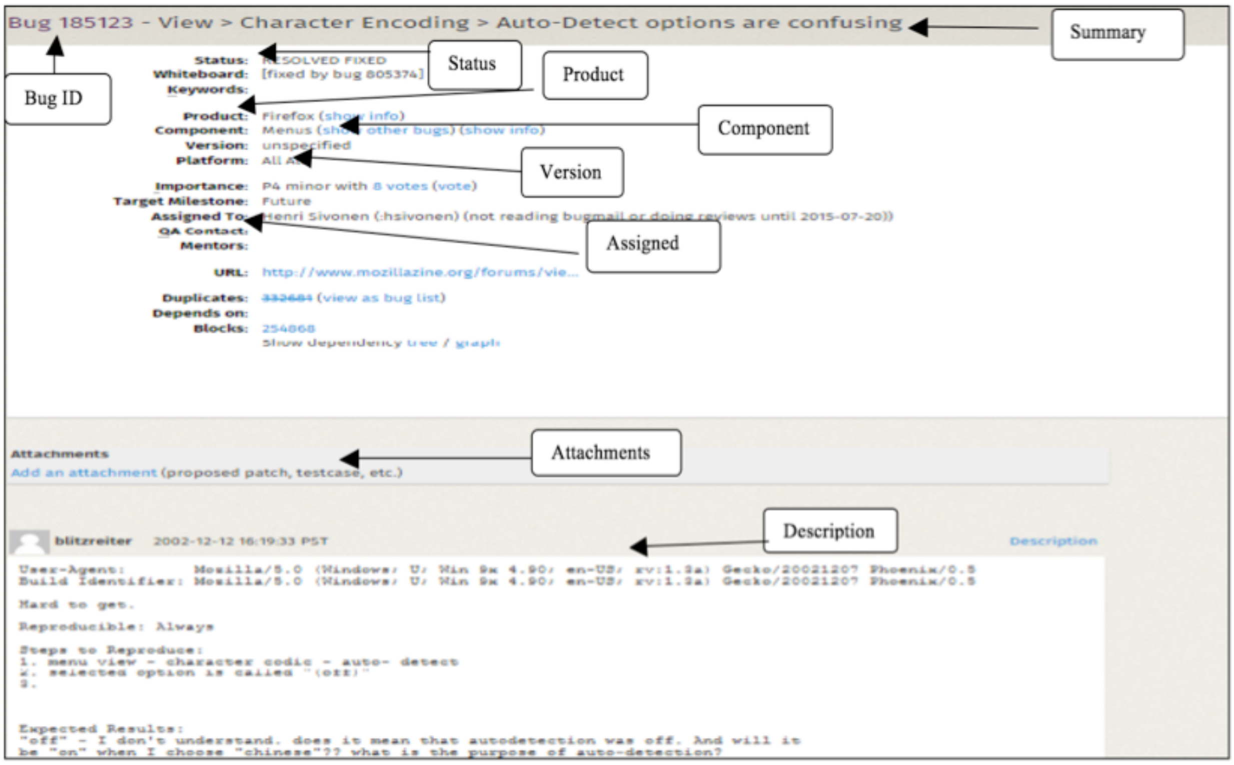
Throughout its lifetime, a bug report goes through a number of stages. Various fields, such as status and resolution, vary many a times. When a bug is reported, the status of bug report is marked as New. The triager then assigns the bug to a developer and its status is marked as Assigned. The developer then fixes the issue and the bug is marked as Resolved. If the tester finds the fix to be correct, the bug is marked as Verified and if not, it is Reopened. After the verification of the bug, it is Closed. At the Resolved status, there are multiple resolutions such as Fixed, Duplicate, Won't Fix, Non-reproducible and Invalid. Figure 2 shows the basic life cycle of a bug report.

3. Systematic review process

This section presents the survey process used in this work. The guidelines of the systematic literature review (SLR) by Kitchenham and Charters [3] were used in this work.

3.1. Survey process

The review process was started with an initial search where the renowned journals and conference proceedings which contained papers concerning bug triaging were selected. Other used materials encompassed even e-sources relevant to



Bug 185123 - View > Character Encoding > Auto-Detect options are confusing

Status:	RESOLVED FIXED
Whiteboard:	[fixed by bug 805374]
Keywords:	
Product:	Firefox (show info)
Component:	Menus (show other bugs) (show info)
Version:	unspecified
Platform:	All A
Importance:	P4 minor with 8 votes (vote)
Target Milestone:	Future
Assigned To:	Henri Sivonen (:hsivonen) (not reading bugmail or doing reviews until 2015-07-20))
QA Contact:	
Mentors:	
URL:	http://www.mozillazine.org/forums/vie...
Duplicates:	~~332684~~ (view as bug list)
Depends on:	
Blocks:	254868
	Show dependency tree / graph

Attachments
Add an attachment (proposed patch, testcase, etc.)

blitzreiter 2002-12-12 16:19:33 PST Description

```
User-Agent:       Mozilla/5.0 (Windows; U; Win 9x 4.90; en-US; rv:1.3a) Gecko/20021207 Phoenix/0.5
Build Identifier: Mozilla/5.0 (Windows; U; Win 9x 4.90; en-US; rv:1.3a) Gecko/20021207 Phoenix/0.5

Hard to get.

Reproducible: Always

Steps to Reproduce:
1. menu view - character codic - auto- detect
2. selected option is called "(off)"
3.


Expected Results:
"off" - I don't understand. does it mean that autodetection was off. And will it
be "on" when I choose "chinese"?? what is the purpose of auto-detection?
```

Figure 1. An instance of bug report

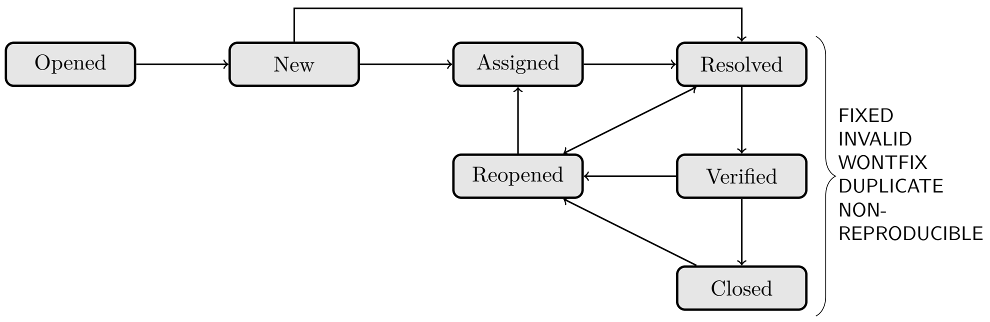


Figure 2. Life cycle of a bug report

software engineering: IEEExplore, ACM Digital library, Google scholar, Citeseer library, Inspec, ScienceDirect and EI Compendex. Selecting such venues ensured that the selected articles meet worthy standards.

To further ensure that no important papers in bug assignment are missed, certain keywords closely related to bug report assignment were identified in the articles obtained from the above venues. A google search was performed to find the identified keywords: bug triaging, bug fixing, bug resolution, bug report assignment and bug AND developer recommendation. These keywords were intentionally broad enough to cover as many articles as possible, although many were less relevant to the present scope of the study. After performing the preliminary keywords and venue search, the studies that propose new bug assignment algorithms were identified. A large number of papers in the keyword search also resulted in papers other than bug report assignment, such as bug duplication, bug localization, severity/priority prediction. All such papers were excluded from this review. After reviewing the titles, abstracts and

Table 1. Distribution of reviewed papers among various sources

Type	Acronym	Description	No. of papers
Journal	JSEP	Journal of Software: Evolution and Process	3
	JSS	Journal on Systems and Software	2
	JSW	Journal of Software	2
	TSE	IEEE Transaction on Software Engineering	2
		Others	**7**
		Total	16
Conference	APSEC	Asia Pacific Software Engineering Conference	2
	ESESC/FSE	European Software Engineering Conference/ ACM SIGSOFT Symposium on the Foundations of Software Engineering	3
	ICSEA	International Conference on Software Engineering Advances	2
	ICSE	International Conference on Software Engineering	5
	ICPC	International Conference on Program Comprehension	2
	ICSM	International Conference on Software Maintenance	3
	ICT-KE	International Conference on ICT and Knowledge Engineering	2
	MSR	Mining Software Repository	7
	PROMISE	International Conference on Predictive Models in Software Engineering	2
	SAC	ACM Symposium on Applied Computing	3
	SEKE	International Conference on Software Engineering and Knowledge Engineering	2
	ESEM	International Symposium on Empirical Software Engineering and Measurement	3
	COMPSAC	International Conference on Computers, Software and Applications	2
		Others	**21**
		Total	59
TOTAL PAPERS (16 + 59)			75

skimming through full articles wherever required, finally 75 papers were reviewed in this study. Each paper was thoroughly verified to assure its the correctness and relevance. Table 1 enlists the distribution of papers across various sources concerning bug report assignment. The venues at which only one surveyed paper was published are grouped together in the "Others" category.

3.2. Inclusion and exclusion criteria

This paper surveys the articles meeting the following inclusion and exclusion criteria:

Inclusion criteria:

1. Papers must relate to developer assignment in bug repositories.
2. Papers must describe the methodology and experimental evaluation of proposed algorithms.
3. Papers must be published in peer reviewed journals and conferences.

Exclusion Criteria:

1. Papers that are duplicates of similar work.
2. Papers that do not describe the methodology and experimental evaluation.
3. Papers that are not published in peer reviewed venues.

3.3. Related surveys

In the past, J. Zhang et al. [1] and T. Zhang et al. [2] performed surveys closely related to this work. These surveys cover all the stages of bug handling, i.e. bug report analysis, bug triaging and bug fixing as shown in Figure 3. Short discussions related to all these stages were carried out in their respective studies. However, a comprehensive overview on each individual stage of bug handling is still missing. This paper focuses on the second stage of bug handling, i.e. bug triaging (or bug report assignment). Bug triaging is an integral stage of bug handling which focuses on the selection of a suitable developer for bug fixing. Hence, this work presents the first large-scale, in-depth, and focused study of bug report assignment. J. Zhang et al. [1] reviewed

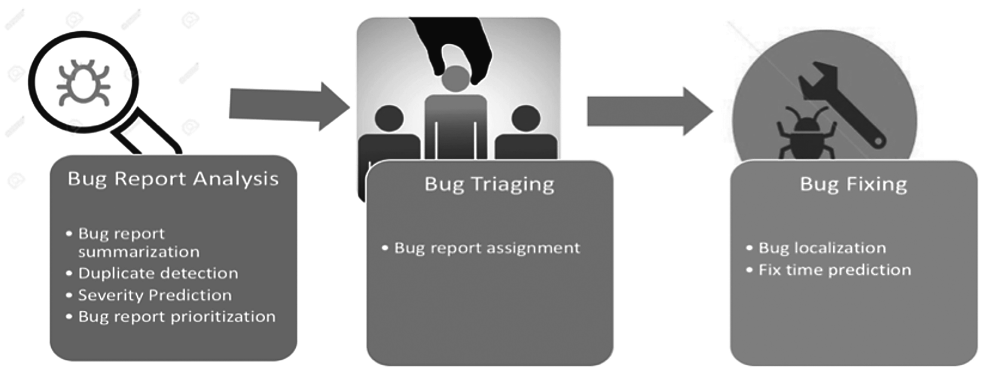


Figure 3. Classification scheme for bug handling process

14 papers whereas T. Zhang et al. [2] reviewed 21 papers related to bug report assignment in their respective studies. The range of surveyed papers covered in this work is larger than in these earlier works. This investigation encompassed the reviews of 75 papers on bug report assignment. It covers papers published before July 2016.. Hence, this study is more comprehensive and up-to-date as compared to the other surveys.

3.4. Research contribution

The following new research contributions differentiate this work from the prior studies:

1. This paper presents the first in-depth, systematic and focused survey on bug triaging considering 75 papers from peer reviewed, refereed conferences and journals published during years 2004 to 2016.
2. Inference drawn from the systematic literature review illustrates ML and IR to be the most popular bug triaging techniques. Thus, a comparison of these popular techniques was done to identify the best bug triaging technique.
3. The paper presents the experimental results of the empirical analysis of two four scale open source projects, Mozilla, Eclipse, Gnome and Open Office of the Bugzilla repository.

4. Bug report assignment

Numerous researchers proposed different bug assignment approaches to semi or fully automate the developer recommendation process. Bug assignment approaches can be classified by the methodology used in the recommendation process. It can be divided into two broad categories: activity profiling of developers [4–8] and location based techniques [9, 10]. The general idea behind the activity profile based techniques is to develop an expertise profile of each developer by using topic modelling. A list of topics is made on the basis of historically fixed bug reports and a membership score is computed for each developer with respect to each topic. This score represents the involvement of a developer in a particular topic in the past. For any new bug report, the topics are extracted and the developer with maximum score corresponding to the obtained topics is recommended. The activity profiling of developers suffers from two major problems: a) Obsolete profiles after some time, b) The developers switch teams or new developers are added, which changes the developer profiles to a major extent thus reducing the recommendation accuracy after some time. However, the high efficiency achieved by activity profile based approaches when the profiles are updated cannot be overlooked.

The location based bug triaging techniques, on the other hand, locate the source code files that need to be updated in order to resolve the issue. The developers who had earlier worked upon these files are considered to be suitable for further updating of these files. These approaches usually make use of the version control repository of the project and thus the data source is more reliable. However, the two-level predictions, firstly the source code files that need to be changed in

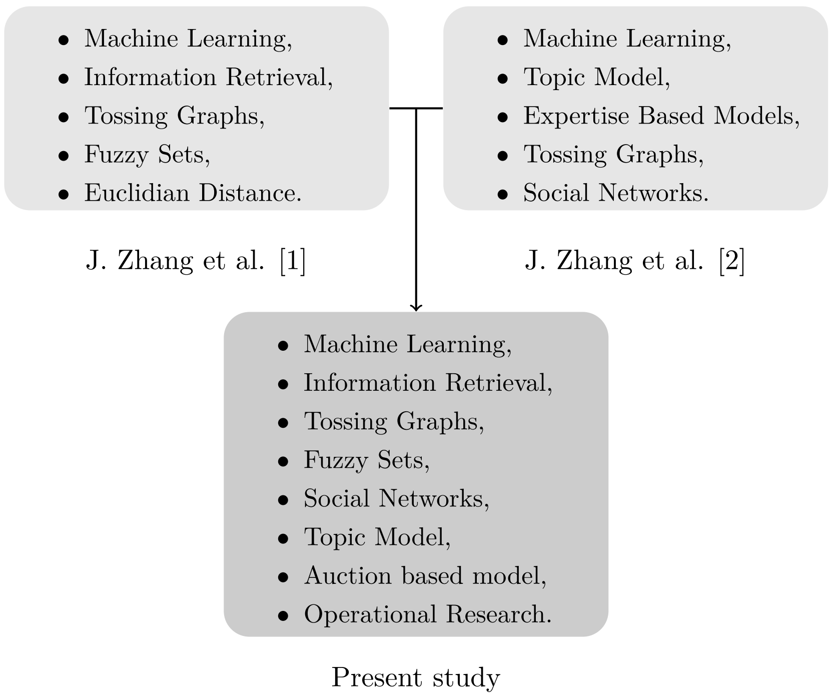


Figure 4. Classification categories in different studies

order to fix the bug and secondly the developer choice, limit the accuracy of the location based approaches as compared to the activity profile based approaches. Therefore, the activity profile based approaches are more popular for bug report assignment in industry.

4.1. Classification based on popular techniques

J. Zhang et al. [1] conducted their study in 2015 and identified bug triaging into five categories: machine learning, information retrieval, tossing graphs, fuzzy set and the Euclidean distance. T. Zhang et al. [2] conducted their study in 2016 and identified bug triaging into the categories: machine learning, topic model, tossing graphs, social networks and expertise model based techniques. In this study, seven categories for bug report assignment were identified after careful inspection. The categories considered first are the ones which were present in both studies, i.e. machine learning and tossing graphs. Next, the papers related to topic modelling and the expertise based model in the category information retrieval were added. The categories which are present in either of the previous studies, i.e. fuzzy sets and social networks, were also considered. In addition, two new categories were identified: auction based techniques and operational research (OR) based approaches. Further, the OR based category include the work related to the areas: Euclidean distance, genetic algorithm and greedy optimization. Hence, after a systematic evaluation of literature, finally seven categories for bug report assignment were inferred: machine learning, information retrieval, auction, social network, tossing graphs, fuzzy set and operational research based techniques. Figure 4 shows the classification categories considered in different studies. Among the seven identified categories, machine learning and information retrieval based techniques are automated and the others are semi-automated. The literature was classified and reviewed under these seven heads as follows:

Machine learning based approaches (see Tab. 2). These approaches train a supervised or unsupervised machine learning classifier with bug reports fixed in the past and then use it for the selection of prominent developers for new bug reports. Cubranic et al. [11] presented one of the few initial bug report assignment approaches based on supervised machine learning classification. They considered the report developer assignment as a text classification problem and trained the machine learning classifier using the tokens obtained from a textual description of fixed bug reports. They correctly classified 30% of Eclipse bug report assignments using supervised Bayesian learning. Xuan et al. [12] highlighted a drawback resulting from the deficiency of labelled bug reports in bug repositories. They first labelled all the unlabelled bug reports using a combination of Naïve Bayes and the Expectation Maximization algorithms and then used the labelled data for training machine learning classifiers.

Anvik et al. [13] recommended to use eight information sources for developer assignment in contrast to the usage of tokens obtained only from a textual description of bug reports. They proposed to use the textual description, component, operating system, hardware, version, developer who owns the code, current workload of developers and developers actively participating in the project to select a prominent developer for a new bug report. They classified bug reports using the support vector machine algorithm and obtained 57% precision for the Eclipse project and 64% precision for the Mozilla project [14]. Although, the inclusion of eight information sources augmented the proficiency of bug assignment selection, sometimes it is still not probable that all the designated eight parameters from each bug tracking system will be obtained.

For instance, large open source bug repositories do not distribute the data concerning the workload of their developers. Thus, it is not always practical to incorporate all the eight fields. Anvik et al. [15, 16] extended their work in order to equate various machine learning classifiers, such as Naïve Bayes, Support Vector Machine (SVM), C4.5, Expectation Maximization, Conjunctive Rules and the Nearest Neighbour (NN) algorithm. Their experimental results exhibited that SVM is the most efficient tool for bug assignment. Similarly, Lucca et al. [17] compared k-NN, SVM and the probabilistic model for bug report assignment.

Bhattacharya et al. [18] surveyed the influence of different dimensions on bug report assignment. They studied how different dimensions such as the choice of a classifier, feature selection, the inclusion of tossing graphs and incremental learning affects bug triaging. Their investigation showed that the Naïve Bayes classifier and the product-component pair as the parameters and the inclusion of tossing graphs along with incremental learning are the best suited dimensions for bug triaging. Their approach achieved significant reduction in tossing lengths. Hu et al. [19] presented a developer-component-bug based bug triaging framework, BugFixer. Xuan et al. [20] focused on the problem of using large datasets for bug assignment, thereby increasing the computation time and complexity of different algorithms. They utilized the combination of feature and instance selection algorithms to choose a dataset for the training of a classifier. Their results demonstrated that scaling down the dataset significantly diminishes the computation complexity and also increases the classification accuracy. Xia et al. [21] proposed DevRec, a dual analysis model which consists of bug report (BR based) and developer (D based) analysis. DevRec is tested on five large projects: GNU, Compiler Collection, Open Office, Mozilla, NetBeans and Eclipse. The precision@5 and precision@10 of DevRec vary from 21.00% to 31.96% and 13.31% to 18.59%, respectively [22].

Machine Learning based approaches consider bug report assignment as a single-label learning problem. In the previous studies, Naïve Bayes is the most popular classifier in machine learning based approaches and it is extensively experimented on in the bug reports of the Bugzilla repository.

Information retrieval based approaches (see Tab. 3). These approaches consider bug reports as documents and transform them to feature vectors which are then processed for optimal developer assignment. These approaches work on the principle that developers with similar expertise towards a certain kind of bugs are proficient enough to solve the new bug report of a similar kind. These techniques consider developer's past expertise towards historically fixed bug reports so as to select a prominent developer.

Moin et al. [23] presented an n-gram based string matching algorithm for bug triaging in the Eclipse JDT project. They transformed the historically fixed bug reports to n-gram tokens. The proposed an approach which matches the n-grams of a new bug report to the n-grams of historically fixed bug reports and allows to find the related fixed bug report. The developer who had fixed the historically similar bug report is designated for the new bug report as well. Matter et al. [4] utilized vocabulary obtained from the source code contributions of developers to build a term-author matrix. Each entry in the matrix represents the frequency of term with respect to a developer. This frequency is considered the expertise of a particular developer with respect to a particular term. For a new bug report, the vocabulary obtained from the textual description of new bug report is matched with the vocabulary of the term-author-matrix and a developer with the highest expertise is designated for the new bug report. Similarly, other researchers used the smoothed unigram model [24], latent semantic indexing [7,25,26], similarity computation [27,28], vector space modelling [28–30] and topic or term modelling approaches [5, 6, 31–37] for developer recommendation. Ahsan et al. [25] implemented dimensionality reduction using feature selection and latent semantic indexing in the expertise matrix.

Somasundaram et al. [38] merged information retrieval with a machine learning based technique for effective developer recommendation. They reviewed three algorithms, SVM-TF-IDF (Support Vector Machine–Term Frequency–Inverse Document Frequency), SVM-LDA (Latent Dirichlet Allocation) and LDA-KL (Kullback Leibler Divergence) and determined LDA-KL to be most effective for developer selection. Shokripur et al. [39] mined information from the version control repository of the project to propose a location based technique for bug triaging. Unlike other approaches, they did not utilize the information obtained from bug tracking systems. Their approach allowed data to be used in new projects also as the underlying data used for recommendation which does not get obsolete after some time.

Shokripur et al. [9] used only the index of unigram noun terms for bug triaging. They concluded that using only unigram noun terms shortens the token index and does not affect the recommendation accuracy. They associated the noun terms with the source code files of the project and then fetched developers who had earlier worked on the linked files for recommendation. Time based expertise decay is also efficient for developer selection in bug report assignment [40–43]. The knowledge of a developer degrades with time. Hence, the calculation of developers' expertise should also comprise time usage as a factor for frequency normalization. This normalization lowers the weight for terms that were previously used and keeps the training data updated. This capability of information retrieval based techniques makes them popular for optimal bug report assignment.

The information retrieval based approaches consider the developer's expertise for bug report assignment. They utilize a large number of the meta-fields of bug reports along with tokens obtained from textual contents. Term frequency modelling is the most popular IR based bug assignment approach.

Auction based approaches (see: Tab. 4). Hosseini et al. [44] proposed an auction based technique for developer recommendation in bug repositories. Upon receipt of a new bug report, the bug triager auctions off the bug report to developers. The software developers who want to work on the auctioned bug report bid to gain it. The bug report is assigned to one of the interested developers on the basis of their bids and current workload status. These techniques are advantageous as the chances of success in such approaches are high as the bidders themselves desire to take the responsibility for fixing the bug. Such approaches usually benefit by moving the style from 'doing the job right' to 'doing the right job'. However, they usually suffers from time delays as the time required for suitable developer assignment is long.

The auction based approach leads to a slower developer assignment process. However, this increases developer's confidence towards a bug.

Social network based approaches (see Tab. 5). A social network refers to the network of social interactions and personal relationships. Social network approaches utilize the relationships between developers and bug reports for the selection of a suitable developer. They compute the developer expertise based on various influencing factors of the network. Various past studies used social network based approaches for bug report assignment [45–49].

The social network based approaches are a recent development in bug triaging. They consider various parameters for decision making and additionally incorporate complex computations in bug triaging task.

Tossing graph based approaches (see Tab. 6). In a normal scenario, a bug triager assigns a bug report to a software developer who makes source code changes in order to fix the bug. If the assigned developer is not able to resolve the bug, then a new developer is assigned for the bug report. Such a process of switching over the bug to new developers is known as bug tossing. Bug tossing is a major problem in bug triaging as approximately 93% of bug reports are tossed at least once in their lifetime [50]. Various researchers propose the use of tossing graph based approaches for bug report assignment in the literature [50–52]. These approaches consider the historical tossing chains illustrating the switching of developers in the past. For a new bug report, the developer is selected on the basis of previous expertise and then tossing chains are checked to identify the most suitable developer.

The tossing graph based approaches help to significantly reduce tossing path lengths. They use various bug meta-fields and topics obtained from textual parameters for similarity calculation and then use historical tossing chains to find the most suitable developer.

Fuzzy set based approaches (see Tab. 7). Fuzzy sets are sets whose elements have degrees of membership. Fuzzy set based bug triaging approaches compute the expertise (or membership score) of developers with respect to various topics obtained from bug parameters. Tamrawi et al. [53, 54] proposed the fuzzy set based approaches in the past. These approaches formulate the term frequency values of IR based approaches into fuzzy set memberships. When a new bug report arrives, matching tokens are obtained and the corresponding membership scores are aggregated.

The fuzzy set based approaches use the fuzzy set theory in which the most descriptive terms characterizing each developer are collected and then used to measure the suitability of a developer for a new bug report.

Operational research based approaches (see Tab. 8). Bug report assignment is an NP hard problem. Hence, different practitioners and researchers have used mathematical techniques, such as greedy optimization, genetic algorithm, the Euclidean distance, to resolve the problem of bug report assignment. Niknafs et al. [55] presented a study on using the genetic algorithm and the multi criteria decision making technique in the personnel assignment problem. Rahman et al. [56] proposed the usage of the greedy optimization technique for developer assignment in bug tracking systems. Xia et al. [21] proposed a machine learning based approach for bug assignment where the similarity between developer and bug report is calculated using the Euclidean distance. Panagiotou et al. [57] proposed the STARDOM approach for bug report assignment and concluded that the analytic hierarchy process (AHP) should be used for the profile construction of developers and for the ranking of developers.

The operational research based approaches utilize mathematical models for bug report assignment. However, scalability issues in such models for large scale open source projects are still questionable.

4.2. Key observations

In this work, the authors have reviewed 75 research papers published during the years 2004-2016. As a result seven categories of bug assignment approaches have been identified. The categories, as mentioned earlier, are: machine learning, information retrieval, auction based, social network, tossing graphs, fuzzy set and operational research based techniques. Based on this in-depth survey, this study is analysed from

Table 2. Comparison of machine learning based approaches

Paper	Year	Technique used	Experimental dataset	Parameters (#Parameters)	Method summary and merits	Remarks/demerits
Cubranic et al. [11]	2004	Machine learning (Naïve Bayes)	Eclipse	Summary and description (2)	Tokens generated from the textual parameters for classification are considered.	30.5% accuracy for eclipse bug reports
Xuan et al. [12]	2010	Machine learning (Naïve Bayes, expectation maximization	Eclipse	Summary and description (2)	The problem of the deficiency of labelled bug reports is considered. Classification efficiency improved by 6% as compared to using only the Naïve Bayes classifier.	Additional overhead to calculate the label of a bug report first.
Bhattacharya et al. [18]	2012	Machine learning (Naïve Bayes, Bayesian Network, J48, SVM)	Mozilla and Eclipse	Product, component (2)	It examined the impact of various dimensions such as classifier selection, feature selection, inclusion of tossing graphs and incremental learning on bug triaging. The Naïve Bayes algorithm is concluded for classifier, product-component pair for features and the use of tossing graphs with incremental learning as the best suited model for bug assignment. Significantly reduced tossing length.	High computational time and cost
Zou et al. [70]	2011	Machine learning (feature selection, instance selection, Naïve Bayes)	Eclipse	Summary and description (2)	Feature selection is used for removing noisy data. It removed 50% of bugs after training set reduction. The performance of original Naïve Bayes algorithm is improved by up to 5% for the Eclipse project.	It used only the Eclipse bug for experimentation. It needs to be applied on more datasetsand also needs more features.
Lin et al. [71]	2009	Machine learning (J48)	Proprietary Chinese dataset	Summary, description, step, bug type, bug class, phase Id, submitter, module Id, bug priority (9)	It proposed a bug assignment model for the Chinese bug dataset. It demonstrated that the non- text based approach outperformed the text based approach.	The proprietary Chinese dataset is used for evaluation. It needs more features.

Banitaan et al. [72]	2013	Machine learning (Naïve Bayes)	Mozilla, Eclipse, NetBeans, Free Desktop	Summary, reporter, component (3)	It demonstrated that the component is the most influential parameter in bug assignment.	It needs more evaluation parameters.
Anvik et al. [73]	2006	Machine learning (Naïve Bayes, SVM, J48)	Mozilla, Eclipse, gcc	Summary and description (2)	The obtained precision was 57% for the Mozilla and 64% for the Eclipse project.	Test set size is too small. 170 for Mozilla and 22 for Eclipse.
Xia et al. [21]	2013	Machine learning (k-Nearest Neighbour)	Mozilla, Eclipse, Netbeans, Open Office, gcc	Product, component, summary, description , developer Id (5)	It proposed Devrec, a composite model which performs two types of analysis: developer based (to find a set of prominent developers) and bug report based analysis (to find similar bug reports).	It needs more features for bug similarity calculation.
Sharma et al. [74]	2015	Machine Learning (association rule mining)	Thunder Bird, Addon SDK, Mozilla	Severity, Priority, Summary (3)	It proposed a novel association rule mining based on bug triaging algorithm.	Reported accuracy up to 81% for top-3 recommendations. More rigorous testing on large dataset required.

Table 3. Comparison of information retrieval based approaches

Paper	Year	Technique used	Experimental dataset	Parameters (#Parameters)	Method summary and merits	Remarks/demerits
Woo et al. [75]	2011	Information retrieval (topic modelling)	Apache, Eclipse, Linux Kernel, Mozilla	Version, platform, milestone, textual description (4)	A content boosted collaborative filtering model which reduces cost without significantly sacrificing accuracy is proposed.	Important bug features, such component in parameter selection, are missing.

Matter et al. [4]	2009	Information retrieval (vocabulary expertise based model)	Eclipse	A developer owns the associated code and description, active participation of developers (3)	The term author matrix to model the expertise of developers was coined here. No need to train the data. The used time decay factor for developers is 3 months.	Low precision value (33.6% for top-1 developer list size in the Eclipse Project)
Shokripour et al. [9]	2013	Information retrieval (location based technique)	Mozilla and Eclipse	Noun terms extracted from source code files, comment messages and identifiers (1)	It proposed a two-phased location based technique for bug report assignment. The first phase predicts the source code files to be updated and second phase lists the probable developers. Reported accuracy up to 89.41% for the Eclipse and 59.76% for Mozilla project for top-5 recommendation list.	Evaluation datasets too small. Test set size:100 No. of unique developers in dataset: 9 and 57 only
Shokripour et al. [40]	2015	Information retrieval (term frequency)	Eclipse, NetBeans, ArgoUML	Noun terms (1)	It proposed a time-based approach term weighting approach. Experimental evaluation shows accuracy improvement up to 11.8%.	No. of unique developer in the dataset too small
Naguib et al. [6]	2013	Information retrieval (term frequency)	Atlas Reconstruction, Eclipse BIRT, UNICASE	Component and description (2)	Proposed an activity profile based technique for bug report assignment.	Considered small datasets for experimentation
Xie et al. [31]	2012	Information retrieval (topic modelling)	Mozilla and Eclipse JDT	Frequent topics in bug report (1)	In the proposed approach Dretom models the developer's expertise based on topic models built from historical fixed bug reports.	Requires more features.
Alenezi et al. [32]	2013	Information retrieval (topic modelling)	Eclipse, NetBeans, Maemo	Bug ID, Assignee, Opened, Changed, Summary, Component (6)	Investigated the use of four term selection methods, namely Log Odds Ratio, Chi-Square, Term Frequency Relevance Frequency and mutual information.	Chi-Square was found to be the best technique.
Alijarah et al. [7]	2011	Information retrieval (latent semantic analysis)	Eclipse	Textual description (1)	It modelled a bug term matrix to compute similar bug reports. The developers are then assigned according to the past expertise.	More rigorous testing is needed.

Anjali et al. [41]	2016	Information retrieval (term frequency)	Mozilla and Eclipse	Component, severity, priority, operating system (4)	It proposed a time decay based technique for bug triaging which uses bug meta-fields to create a term author matrix. The frequency values in the matrix are then degraded according to the last usage time.	More features are needed.

Table 4. Auction based approach

Paper	Year	Technique used	Experi-mental dataset	Parameters (#Parameters)	Method summary and merits	Remarks/demerits
Hosseini et al. [44]	2012	Auction based technique	Mozilla and Eclipse	Bug id, creation date, last updated date, classification id, product, component, version, platform, operating system, bug status, resolution, duplicate id, bug file location, keywords, priority, bug severity, target milestone, dependent bugs, blocked, votes, reporter name, assigned to name and number of comments (22)	The developers place their own bids as per their own interest/expertise. It reduces the chances of bug tossing. Overall the method saves bug rectification time.	Slow developer assignment process. Low Accuracy values up to 33.54% and 25.14% obtained for the Mozilla and Eclipse projects, respectively.

Table 5. Comparison of social network based approaches

Paper	Year	Technique used	Experi-mental dataset	Parameters (#Parameters)	Method summary and merits	Remarks/demerits
Wu et al. [45]	2011	Social network	Mozilla	Summary and description (2)	It finds historical similar bug reports with the help of the k-Nearest Neighbour search and then uses in-degree, out-degree, page rank, betweenness and closeness metrics to rank developers from a social network. The obtained recall value is up to 0.60 for the Mozilla project.	Additional cost to adjust algorithmic parameters.

Zhang et al. [46]	2012	Social network	JBoss	Summary and description (2)	It builds the concept profile with the topic terms of bug reports and then uses the social network of developers to find the developer with the highest probability of fixing.	Additional cost to maintain social network of developers. Computationally intensive.
Xuan et al. [47]	2012	Social network	Mozilla and Eclipse	Bug ID, reporter, fixer, summary, description, creation time, and comments (7)	It models the bug report assignment as the developer prioritization problem by extending a socio-technical approach. In the proposed approach, the out-degree of developers is used to construct a social network which is then used in prioritization.	Low accuracy values (up to 50%).
Yang et al. [48]	2014	Social network	JBoss	Bug ID, description, number of comments, summaries, developer ID, component (6)	It proposed a multi developer network based approach for bug report assignment. Potential contributors are extracted from similar component and keyword matching.	Additional bug parameters should be used for similarity matching.
Zhang et al. [49]	2013	Social network	Mozilla and Eclipse	It utilized objects such as developers, bugs, comments, and components, as well as links denoting different relations among these objects. (2)	It introduced a heterogeneous developer contribution network to model the multiple types of contribution from developers to components in software bug repositories.	Computationally complex process.

Table 6. Comparison of tossing graph based approaches

Paper	Year	Technique used	Experi-mental dataset	Parameters (#Parameters)	Method summary and merits	Remarks/demerits
Bhattacharya et al. [50]	2010	Tossing graph	Mozilla and Eclipse	Bug Id, developer Id, product, component, keywords obtained from summary and description (6)	It utilized multi-feature tossing graphs along with machine learning classifiers to reduce the tossing path lengths and improve the efficiency of a bug recommender. Reduced tossing length up to 86%	More bug parameters should be added for the construction of tossing graphs.

Jeong et al. [51]	2009	Tossing graph	Mozilla and Eclipse	Tossing information of developers (1)	It introduced the use of Markov chains based on tossing graphs for efficient developer recommendation. Reduced tossing events by up to 72%.	Additional cost to obtain the path from the first recommendation to the final fixer.
Chen et al. [52]	2011	Tossing graph	Mozilla and Eclipse	Tossing information of developers, reporter, classification, component, product and summary (6)	It introduced an approach to improve bug assignment using a bug tossing graph and bug similarity (vector space model).	Better bug similarity techniques could be employed.

Table 7. Comparison of fuzzy set based approaches

Paper	Year	Technique used	Experimental dataset	Parameters (#Parameters)	Method summary and merits	Remarks/demerits
Tamrawi et al. [53]	2011	Fuzzy sets	Eclipse	Bug ID, developer ID, summary, description (4)	It introduced the fuzzy set based model for developer recommendation by using membership functions.	Low accuracy: 37.81%
Tamrawi et al. [54]	2011	Fuzzy sets	Mozilla, Eclipse, Apache, NetBeans, FreeDesktop, Gcc and Jazz	Bug ID, developer ID, summary, description, creation/fixing time (5)	It developed the fuzzy set and cache based approach, Bugzie for the calculation of developer expertise. Lower response time and better accuracy as compared to the SVM based classification.	Additional overhead in topic modelling and calculating fuzzy numbers.

Table 8. Comparison of operational research based approaches

Paper	Year	Technique used	Experimental dataset	Parameters (#Parameters)	Method summary and merits	Remarks/demerits
Nikanfs et al. [55]	2010	Operational research based technique (genetic algorithm)	Eclipse JDT and two industrial projects	Current workload of developers (1)	Proposed the genetic algorithm based technique for developer recommendation.	Requires rigorous experimental evaluation.
Rahman et al. [56]	2011	Operational research based technique (greedy optimization)	Industrial project	Fluency, contribution, effectiveness and receny (4)	It proposed a profile creation and maintenance module. The importance factor of developers is calculated based on various features which are further prioritized on the basis of various properties.	Difficult to implement due to the calculation of various extracted parameters.
Karim et al. [22]	2016	Operational research based technique (genetic algorithm)	Eclipse	Component, no. of file changes, fix time. (3)	It proposed a single objective (minimization of bug fix time) and as a bi-objective (minimization of bug fix time and cost). The developer assignment algorithm uses the genetic algorithm.	Two level prediction (fix time and prominent developer) minimizes the accuracy.
Rahman et al. [76]	2009	Operational research based technique (greedy optimization)	Eclipse	Lines of code (1)	It proposed the greedy optimization algorithm for bug report assignment.	Line of code is a weak metric for any kind of prediction as the coding styles of developers vary to a great extent.

two perspectives: the frequency wise distribution of techniques and the year wise distribution of each technique.

- **Frequency wise distribution of each bug assignment technique:** In this perspective, the popularity of all bug triaging techniques identified in this study was analysed. The frequency of each technique was accumulated and the cumulative frequency distribution was developed. The resultant histogram is presented in Figure 5b. A similar analysis was performed on the papers from the existing surveys [1, 2]. The frequency distribution of techniques in their studies is shown in Figure 5a. The conducted analyses show, Figure 5a and 5b, that the machine learning and information retrieval based techniques are most popular for bug assignment.

Another analysis was done in order to check the year wise trend of the bug assignment techniques in the last two decades.

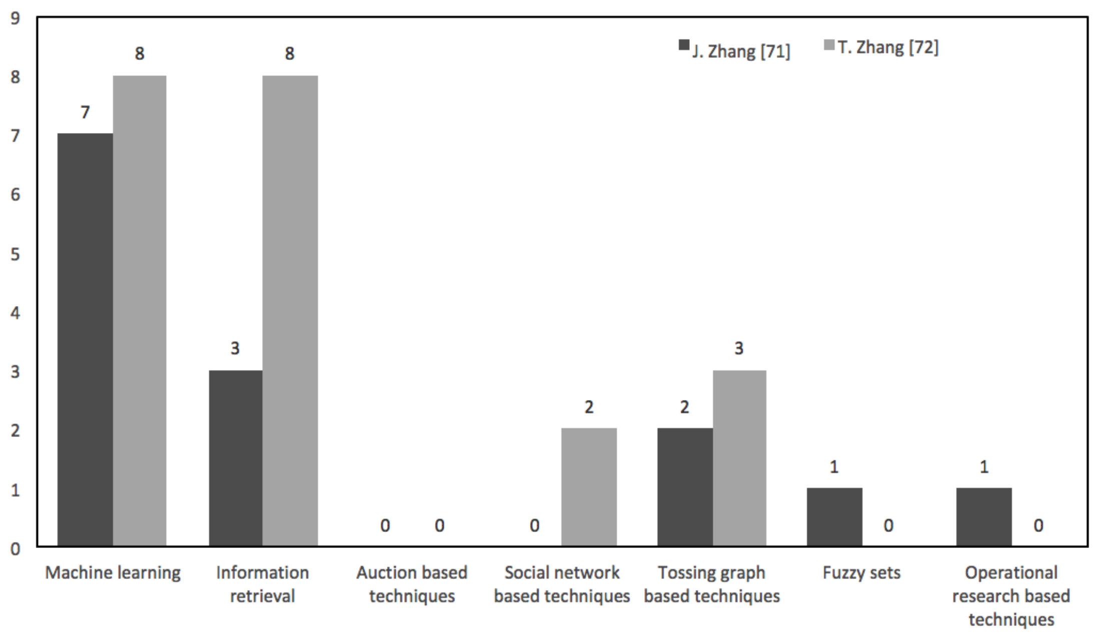
- **Year wise distribution of each bug assignment technique:** a) From the above analysis, ML and IR were identified to be the most popular techniques among all categories of bug triaging. To further analyse the on-going trend among the popular techniques, the year wise frequency distribution of ML and IR based techniques was plotted. Figure 6a shows the year wise frequency distribution of ML and IR based techniques in the existing study [2]. Since, J. Zhang et al. [1] surveyed very few papers on bug triaging, the trend analysis will not give any significant insights. Hence, it is believed that this survey is not useful for this analysis. Figure 6b represents the year wise trend analysis of ML and IR based techniques in the current study. It was observed that there is a considerable trend shift from ML to IR. Researchers prefer to use the IR based technique for bug triaging.

To further examine the reason behind this trend shift, an empirical study on two most popular techniques, ML and IR for bug triaging, was performed.

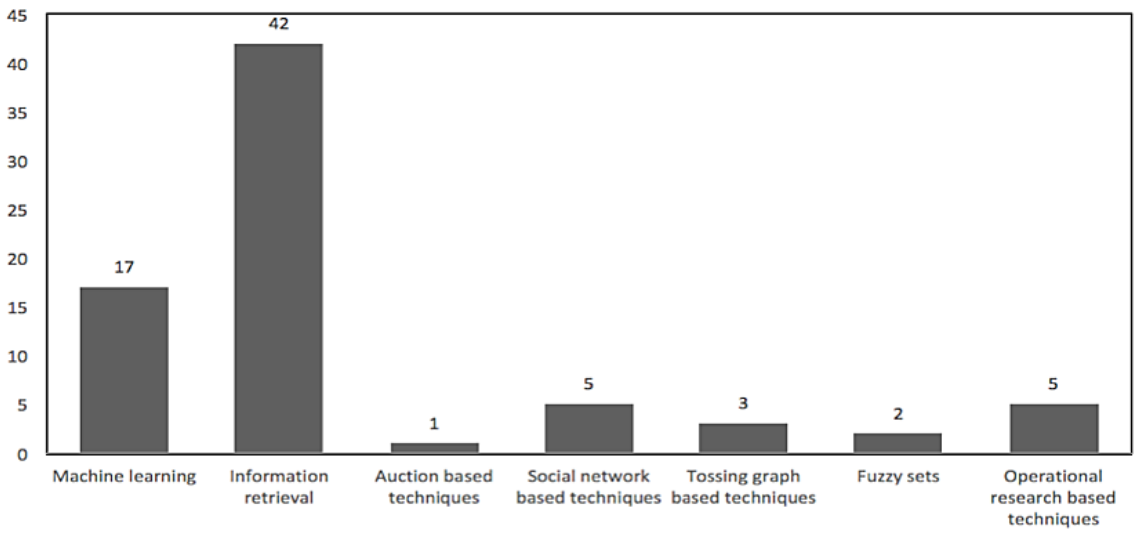
4.3. Comparative study of machine learning and information retrieval techniques

To evaluate the efficiency of the machine learning and information retrieval based techniques, the techniques based on the bug reports of four large scale popular projects of Bugzilla repository, Mozilla, Eclipse, Gnome and Open Office, were applied. Bugzilla is the most popular open source bug repository used by many varied size software projects. The projects selected for the comparative study contain large number of bug reports and are widely used in Bugzilla. They have been developed for years and thus now they are aged. This increases the confidence of researchers in the use of these projects for experimental evaluations in their work.

The datasets for the comparative study were collected from the issue tracking system (ITS) of the Mozilla, Eclipse, Gnome and Open Office projects. Bug reports submitted over the span of 6 years (from January 01, 2011 to December 31, 2016) were collected in this study. Only bug reports with their resolution marked as fixed and the status marked as resolved, verified or closed were extracted. This extraction scheme will ensure the presence of developers who had actually fixed the bug. In this study, four most important bug meta-fields were used: component, severity, priority and operating system. These parameters are selected as they contain the most important information related to a bug and are extensively used in literature [13, 77]. Moreover, these fields generally do not contain any missing values for both fixed and new bug reports. This allows enough training and testing tokens for optimized bug report assignment. Initially, a total of 68,904 bug reports was obtained for all the four projects (20,483 bug reports for the Mozilla project, 39,758 for the Eclipse project, 6,326 for the Gnome project and 2,337 for the Open Office project). The collected bug reports were fixed by 3,301 unique developers (1,218 developers for the Mozilla project, 1,342 for the Eclipse project, 611 for the Gnome project and 130 for the Open Office project).



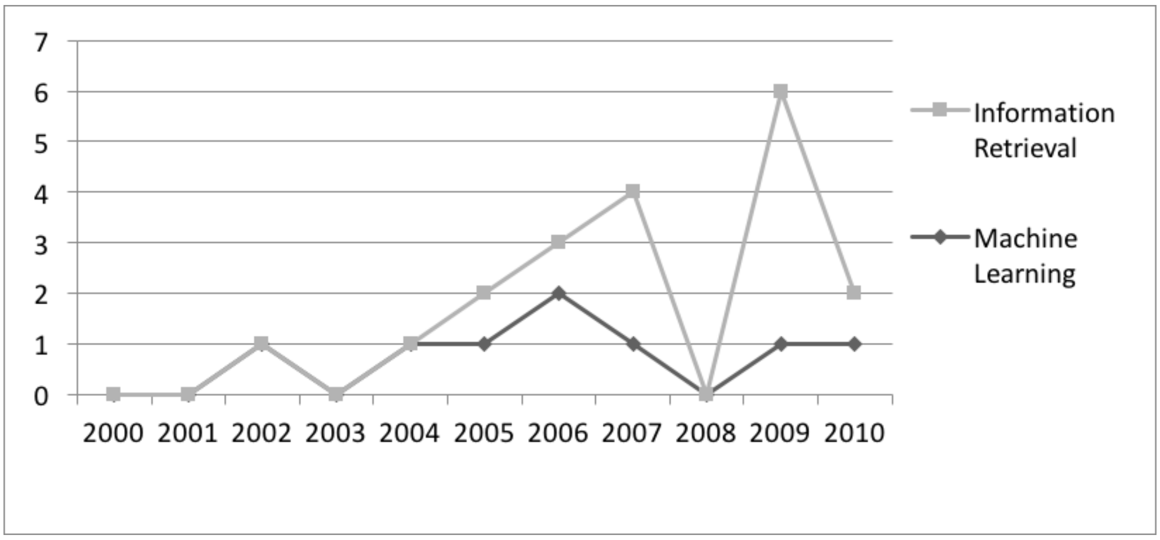
(a) Past studies



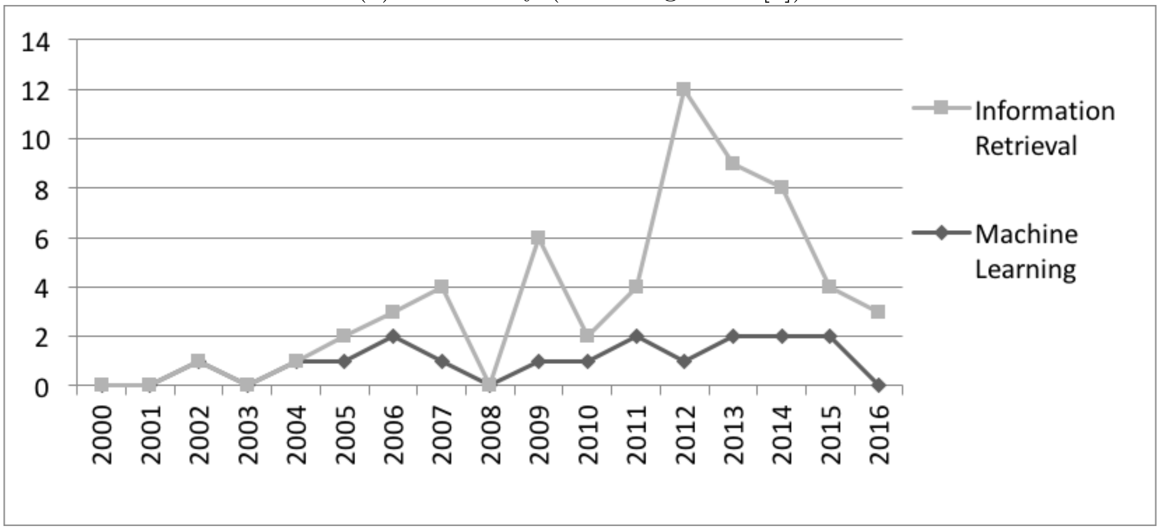
(b) Present study

Figure 5. Frequency distribution

For pre-processing, the bug reports in which the assigned-to field was unspecified were removed. In the Bugzilla repository, there are developers who had fixed few bugs. The inclusion of such developers would deteriorate the model performance so the parameter was further tuned, ($N \geq 10$), i.e., the number of bug reports fixed by a developer in the past. Hence, finally a total of 59,448 bug reports were obtained for all four projects (15,017 bug reports for the Mozilla project, 37,425 for the Eclipse project, 4,947 for the Gnome project and 2,059 for the Open Office project). The pre-processed bug reports were fixed by 940 unique developers (267 developers for the Mozilla project, 505 for the Eclipse project, 140 for the Gnome project and 28 for the Open Office project). Table 9 shows various details of the datasets used for



(a) Past study (T. Zhang et al. [2])



(b) Present study

Figure 6. Year wise distribution

comparison: start date, end date, number of collected bugs, number of distinct assignees (or developers), number of bug reports with $N \geq 10$, number of assignees who have fixed more than 10 bug reports in the past, unique number of tokens in various meta-fields of bug reports, such as component, severity, priority and operating system. A total 500 fixed bug reports randomly selected from each project were used for testing.

For the machine learning based classification, the use of four machine learning algorithms was investigated: Naïve Bayes, J48, Random tree and Bayes Net. These algorithms were selected as they covered different categories of supervised machine learning algorithms. The Weka toolkit was used for experimentation. Table 10 shows the results of the 10-fold cross validation of the machine learning based approach. Different classifiers achieved the best classification accuracy among different projects. For instance, in the Mozilla project the J48 classifier obtained the best classification accuracy of 44%, whereas the Naïve Bayes, Random Tree and Bayes net classifiers achieved 35%, 40% and 39% accuracy, respectively. Similarly, for the Eclipse project J48 and Random Tree obtained the best classification results of 44% accuracy. For the Gnome

Table 9. Dataset Details

	Mozilla	Eclipse	Gnome	Open Office
Start Date	01/01/2011	01/01/2011	01/01/2011	01/01/2011
End date	31/12/2016	31/12/2016	31/12/2016	31/12/2016
#bug reports collected	20,483	39,758	6,326	2,337
#assignees	1,218	1,342	611	130
#bug reports (N>=10)	15,017	37,425	4,947	2,059
#assignees(N>=10)	267	505	140	28
#component	367	498	400	100
#severity	7	7	7	6
#priority	5	5	5	5
#operating system	27	30	11	28

project, the J48 classifier obtained an accuracy of 53% and for the Open Office project the Random Tree classifier achieved the best accuracy of 45.2%. Overall, it was found out that a single classifier could not be declared as the best one for all the projects and different classifiers perform variably for different projects. However, it was observed that tree based classifiers, J48 and Random Tree are best suitable for bug report assignment.

For the information retrieval based technique, the term frequency (TF) based approach was used as it is most widely used in the literature [5, 6, 40]. First a term-author-matrix was created, $M[i, j]$, from the tokens obtained from the different meta-fields of bug reports (component, severity, priority and operating system). In the term-author-matrix, M denotes all the unique developers, i are authors and all the values in the various tokens in the meta-fields of a bug report are considered as terms, j. Each entry in the matrix represents the frequency, f_{ij} of developer, i with respect to a term, j. Frequency f_{ij} represents the expertise of a developer, i with respect to a term, j based on the work done by the developer in the past. Figure 7 shows an instance of a term-author-matrix. In the figure, gui, general, regression represents various distinct terms (or tokens) obtained from the various meta-fields of bug report and pollman, jaze and rick are the developers in the bug repository. The numeric values in the matrix represent the expertise values of developers while w.r.t. the terms in the past fixed bug reports.

To identify a suitable developer for a new bug report, its terms are extracted from the meta-fields and are considered as a search query. Columns from term-author-matrix matching the terms in the search query are extracted. To calculate the final expertise score for each developer, the frequency values of each developer are aggregated. The developer with a higher score is considered to be suitable as they have more expertise in the areas of the new bug report. Table 10 shows the results of the top-k ($k = 5$ and 10) recommendation list sizes in the informational retrieval based approach. In the Mozilla project, the achieved maximum accuracy is 52% for the top-10 list size. Similarly, the maximum achieved accuracy is 49.6%, 72% and 87% for the Eclipse, Gnome and Open Office projects, respectively.

Comparing the results of the machine learning and information retrieval based techniques, it was found out that the information retrieval based techniques yield better accuracy as compared to the machine learning based technique. Thus, information retrieval is a better technique for activity profile based bug report assignment approaches. In the Mozilla project, the J48 machine learning algorithm gives 44% accuracy which increases by 6% for the top-10 recommendation list in information retrieval. Similarly, in the Eclipse, Gnome and Open Office projects the accuracy of the information retrieval based technique is significantly higher than in the machine learning based approach. This supports the view that the information retrieval based technique achieves better accuracy and thus there is a trend shift in bug assignment approaches from the machine learning based techniques to the information retrieval based technique.

		Terms		
		gui	general	regression
Developers/ Authors	Pollman	8	6	5
	Jaze	4	1	8
	Rick	3	4	2

$\Leftarrow$ Expert values of developers w.r.t. terms

Figure 7. An instance of term-author-matrix

Table 10. Classification accuracy of Machine Learning and Information Retrieval algorithms

		Mozilla	Eclipse	Gnome	Open Office
Machine Learning	Naïve Bayes	35%	33%	43%	44.2%
	J48	44%	44%	53%	42.3%
	Random Tree	40%	44%	52%	45.2%
	Bayes Net	39%	35%	47%	44.2%
Information Retrieval	Top-5	47%	45.2%	60%	62%
	Top-10	52%	49.6%	72%	87%

5. Conclusion and future work

Bug report assignment is a time consuming and tedious task for a bug triager. This paper presents a review and classification of 75 research papers in the area of automated bug assignment. Seven categories of bug assignment approaches have been identified in this study. The identified categories are machine learning, information retrieval, auction, social network, tossing graph, fuzzy set and operational research based techniques. We systematically organized 75 surveyed papers in one of the seven identified techniques of bug triaging. Further, we analysed the surveyed papers in two perspectives: the frequency wise distribution of techniques and the year wise distribution of each technique. Interesting facts are captured in this analytical study. First, the machine learning and information retrieval based techniques are most popular for automatic bug report assignment. Second, the current trend of bug assignment approaches is shifting from machine learning to the information retrieval based techniques.

To examine the reason behind this shift, an empirical study was performed on the machine learning and information retrieval based bug triaging technique. The study was done on real time, large scale, open source projects, Mozilla, Eclipse, Gnome and Open Office. The results of the analysis showed an increase of up to 12.8% in the efficiency for the top-5 list size in the information retrieval based technique. Thus, the information retrieval based techniques are the best choice for bug triaging. The possible reasons for this shift are better efficiency, ability to consider the current expertise of developers and the ability to cooperate with other techniques.

Although a high volume of literature is available in the area of automated bug assignment, there is still a deficiency of a technique which presents an acceptable efficiency to be used in the real time environment. There are three major difficulties in bug handling. a) Sheer volume of information available in bug repositories, b) collaborative work by developers for bug rectification, and c) continuous evolvement of project or software systems. These difficulties lead to a series of open issues in bug assignment approaches, such as profiling new developers, maintaining updated profiles, workload balancing, assignment of reopened bugs and most importantly the reliability of the data in bug tracking repositories. In the future, we plan to implement an information retrieval based technique which considers the time-based expertise computation and to test it on a large dataset considering a huge number of developers as is the case in real time environment.

References

[1] J. Zhang, X. Wang, D. Hao, B. Xie, L. Zhang, and H. Mei, “A survey on bug-report analysis,” *Science China Information Sciences*, Vol. 58, No. 2, 2015, pp. 1–24.

[2] T. Zhang, H. Jiang, X. Luo, and A.T. Chan, “A literature review of research in bug resolution: Tasks, challenges and future directions,” *The Computer Journal*, Vol. 59, No. 5, 2016, pp. 741–773.

[3] B. Kitchenham and S. Charters, “Guidelines for performing systematic literature reviews in software engineering,” Software Engineering Group, School of Computer Science and Mathematics, Keele University and Department of Computer Science, University of Durham, Tech. Rep. EBSE 2007-001, 2007. [Online]. https://pdfs.semanticscholar.org/e62d/bbbbe70cabcde3335765009e94ed2b9883d5.pdf

[4] D. Matter, A. Kuhn, and O. Nierstrasz, “Assigning bug reports using a vocabulary-based expertise model of developers,” in *6th IEEE International Working Conference on Mining Software Repositories.* IEEE, 2009, pp. 131–140.

[5] A.S.K. Singh, “Bug triaging: Profile oriented developer recommendation,” *International Journal of Innovative Research in Advanced Engineering*, Vol. 2, 2014, pp. 36–42.

[6] H. Naguib, N. Narayan, B. Brügge, and D. Helal, “Bug report assignee recommendation using activity profiles,” in *10th IEEE Working Conference on Mining Software Repositories (MSR).* IEEE, 2013, pp. 22–30.

[7] I. Aljarah, S. Banitaan, S. Abufardeh, W. Jin, and S. Salem, “Selecting discriminating terms for bug assignment: a formal analysis,” in *Proceedings of the 7th International Conference on Predictive Models in Software Engineering.* ACM, 2011.

[8] A. Sureka, H. Kumar Singh, M. Bagewadi, A. Mitra, and R. Karanth, “A decision support platform for guiding a bug triager for resolver recommendation using textual and non-textual features,” in *3rd International Workshop on Quantitative Approaches to Software Quality*, 2015, p. 25.

[9] R. Shokripour, J. Anvik, Z.M. Kasirun, and S. Zamani, “Why so complicated? simple term filtering and weighting for location-based bug report assignment recommendation,” in *Proceedings of the 10th Working Conference on Mining Software Repositories.* IEEE Press, 2013, pp. 2–11.

[10] F. Servant and J.A. Jones, “WhoseFault: automatic developer-to-fault assignment through fault localization,” in *Proceedings of the 34th International Conference on Software Engineering.* IEEE Press, 2012, pp. 36–46.

[11] G. Murphy and D. Cubranic, “Automatic bug triage using text categorization,” in *Proceedings of the Sixteenth International Conference on Software Engineering & Knowledge Engineering*, 2004.

[12] J. Xuan, H. Jiang, Z. Ren, J. Yan, and Z. Luo, “Automatic bug triage using semi-supervised text classification,” in *22nd International Conference on Software Engineering and Knowledge Engineering (SEKE)*, 2010, pp. 209–214.

[13] J. Anvik, “Automating bug report assignment,” in *Proceedings of the 28th International Conference on Software Engineering.* ACM, 2006, pp. 937–940.

[14] J. Anvik, L. Hiew, and G.C. Murphy, “Coping with an open bug repository,” in *Proceedings of the 2005 OOPSLA Workshop on Eclipse Technology Exchange.* ACM, 2005, pp. 35–39.

[15] J. Anvik and G.C. Murphy, “Reducing the effort of bug report triage: Recommenders for development-oriented decisions,” *ACM Transactions on Software Engineering and Methodology (TOSEM)*, Vol. 20, No. 3, 2011, pp. 10:1–10:35.

[16] J.K. Anvik, “Assisting bug report triage through recommendation,” Ph.D. dissertation, University of British Columbia, 2007.

[17] G.A. Di Lucca, M. Di Penta, and S. Gradara, “An approach to classify software maintenance requests,” in *International Conference on Software Maintenance.* IEEE, 2002, pp. 93–102.

[18] P. Bhattacharya, I. Neamtiu, and C.R. Shelton, “Automated, highly-accurate, bug assignment using machine learning and tossing graphs,” *Journal of Systems and Software*, Vol. 85, No. 10, 2012, pp. 2275–2292.

[19] H. Hu, H. Zhang, J. Xuan, and W. Sun, “Effective bug triage based on historical bug-fix information,” in *IEEE 25th International Symposium on Software Reliability Engineering (ISSRE).* IEEE, 2014, pp. 122–132.

[20] J. Xuan, H. Jiang, Y. Hu, Z. Ren, W. Zou, Z. Luo, and X. Wu, “Towards effective bug triage with software data reduction techniques,” *IEEE Transactions on Knowledge and Data Engineering*, Vol. 27, No. 1, 2015, pp. 264–280.

[21] X. Xia, D. Lo, X. Wang, and B. Zhou, “Accurate developer recommendation for bug resolution,” in *20th Working Conference on Reverse Engineering (WCRE).* IEEE, 2013, pp. 72–81.

[22] M.R. Karim, G. Ruhe, M. Rahman, V. Garousi, T. Zimmermann *et al.*, "An empirical investigation of single-objective and multiobjective evolutionary algorithms for developer's assignment to bugs," *Journal of Software: Evolution and Process*, Vol. 28, No. 12, 2016, pp. 1025–1060.

[23] A. Moin and G. Neumann, "Assisting bug triage in large open source projects using approximate string matching," in *Seventh International Conference on Software Engineering Advances (ICSEA)*, Lisbon, Portugal, 2012.

[24] T. Zhang and B. Lee, "A hybrid bug triage algorithm for developer recommendation," in *Proceedings of the 28th Annual ACM Symposium on Applied Computing*. ACM, 2013, pp. 1088–1094.

[25] S.N. Ahsan, J. Ferzund, and F. Wotawa, "Automatic software bug triage system (BTS) based on latent semantic indexing and support vector machine," in *Fourth International Conference on Software Engineering Advances*. IEEE, 2009, pp. 216–221.

[26] G. Canfora and L. Cerulo, "How software repositories can help in resolving a new change request," in *IEEE International Workshop on Software Technology and Engineering Practice (STEP)*, 2005, pp. 99–103.

[27] H. Kagdi, M. Gethers, D. Poshyvanyk, and M. Hammad, "Assigning change requests to software developers," *Journal of Software: Evolution and Process*, Vol. 24, No. 1, 2012, pp. 3–33.

[28] N.K. Nagwani and S. Verma, "Predicting expert developers for newly reported bugs using frequent terms similarities of bug attributes," in *9th International Conference on ICT and Knowledge Engineering (ICT & Knowledge Engineering)*. IEEE, 2012, pp. 113–117.

[29] O. Baysal, M.W. Godfrey, and R. Cohen, "A bug you like: A framework for automated assignment of bugs," in *IEEE 17th International Conference on Program Comprehension*. IEEE, 2009, pp. 297–298.

[30] K. Kevic, S.C. Müller, T. Fritz, and H.C. Gall, "Collaborative bug triaging using textual similarities and change set analysis," in *6th International Workshop on Cooperative and Human Aspects of Software Engineering (CHASE)*. IEEE, 2013, pp. 17–24.

[31] X. Xie, W. Zhang, Y. Yang, and Q. Wang, "Dretom: Developer recommendation based on topic models for bug resolution," in *Proceedings of the 8th International Conference on Predictive Models in Software Engineering*. ACM, 2012, pp. 19–28.

[32] M. Alenezi, K. Magel, and S. Banitaan, "Efficient bug triaging using text mining," *JSW*, Vol. 8, No. 9, 2013, pp. 2185–2190.

[33] G. Canfora and L. Cerulo, "Supporting change request assignment in open source development," in *Proceedings of the 2006 ACM Symposium on Applied Computing*. ACM, 2006, pp. 1767–1772.

[34] T. Zhang, G. Yang, B. Lee, and E.K. Lua, "A novel developer ranking algorithm for automatic bug triage using topic model and developer relations," in *21st Asia-Pacific Software Engineering Conference (APSEC)*, Vol. 1. IEEE, 2014, pp. 223–230.

[35] E. Linstead, P. Rigor, S. Bajracharya, C. Lopes, and P. Baldi, "Mining eclipse developer contributions via author-topic models," in *Fourth International Workshop on Mining Software Repositories*. IEEE, 2007, pp. 30–30.

[36] G. Yang, T. Zhang, and B. Lee, "Towards semi-automatic bug triage and severity prediction based on topic model and multi-feature of bug reports," in *IEEE 38th Annual Computer Software and Applications Conference (COMPSAC)*. IEEE, 2014, pp. 97–106.

[37] X. Xia, D. Lo, Y. Ding, J.M. Al-Kofahi, T.N. Nguyen, and X. Wang, "Improving automated bug triaging with specialized topic model," *IEEE Transactions on Software Engineering*, Vol. 43, No. 3, 2017, pp. 272–297.

[38] K. Somasundaram and G.C. Murphy, "Automatic categorization of bug reports using latent dirichlet allocation," in *Proceedings of the 5th India Software Engineering Conference*. ACM, 2012, pp. 125–130.

[39] R. Shokripour, Z.M. Kasirun, S. Zamani, and J. Anvik, "Automatic bug assignment using information extraction methods," in *International Conference on Advanced Computer Science Applications and Technologies (ACSAT)*. IEEE, 2012, pp. 144–149.

[40] R. Shokripour, J. Anvik, Z.M. Kasirun, and S. Zamani, "A time-based approach to automatic bug report assignment," *Journal of Systems and Software*, Vol. 102, 2015, pp. 109–122.

[41] D. Mohan, N. Sardana *et al.*, "Visheshagya: Time based expertise model for bug report assignment," in *Ninth International Conference on Contemporary Computing (IC3)*. IEEE, 2016, pp. 1–6.

[42] S. Zamani, S.P. Lee, R. Shokripour, and J. Anvik, "A noun-based approach to feature location using time-aware term-weighting,"*Information*

and Software Technology, Vol. 56, No. 8, 2014, pp. 991–1011.

[43] T.T. Nguyen, A.T. Nguyen, and T.N. Nguyen, "Topic-based, time-aware bug assignment," *ACM SIGSOFT Software Engineering Notes*, Vol. 39, No. 1, 2014, pp. 1–4.

[44] H. Hosseini, R. Nguyen, and M.W. Godfrey, "A market-based bug allocation mechanism using predictive bug lifetimes," in *16th European Conference on Software Maintenance and Reengineering (CSMR)*. IEEE, 2012, pp. 149–158.

[45] W. Wu, W. Zhang, Y. Yang, and Q. Wang, "Drex: Developer recommendation with *k*-Nearest-Neighbor search and expertise ranking," in *18th Asia Pacific Software Engineering Conference (APSEC)*. IEEE, 2011, pp. 389–396.

[46] T. Zhang and B. Lee, "An automated bug triage approach: A concept profile and social network based developer recommendation," in *International Conference on Intelligent Computing*. Springer, 2012, pp. 505–512.

[47] J. Xuan, H. Jiang, Z. Ren, and W. Zou, "Developer prioritization in bug repositories," in *34th International Conference on Software Engineering (ICSE)*. IEEE, 2012, pp. 25–35.

[48] G. Yang, T. Zhang, and B. Lee, "Utilizing a multi-developer network-based developer recommendation algorithm to fix bugs effectively," in *Proceedings of the 29th Annual ACM Symposium on Applied Computing*. ACM, 2014, pp. 1134–1139.

[49] W. Zhang, S. Wang, Y. Yang, and Q. Wang, "Heterogeneous network analysis of developer contribution in bug repositories," in *International Conference on Cloud and Service Computing (CSC)*. IEEE, 2013, pp. 98–105.

[50] P. Bhattacharya and I. Neamtiu, "Fine-grained incremental learning and multi-feature tossing graphs to improve bug triaging," in *IEEE International Conference on Software Maintenance (ICSM)*. IEEE, 2010, pp. 1–10.

[51] G. Jeong, S. Kim, and T. Zimmermann, "Improving bug triage with bug tossing graphs," in *Proceedings of the the 7th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering*. ACM, 2009, pp. 111–120.

[52] L. Chen, X. Wang, and C. Liu, "An approach to improving bug assignment with bug tossing graphs and bug similarities," *JSW*, Vol. 6, No. 3, 2011, pp. 421–427.

[53] A. Tamrawi, T.T. Nguyen, J. Al-Kofahi, and T.N. Nguyen, "Fuzzy set-based automatic bug triaging: NIER track," in *33rd International Conference on Software Engineering (ICSE)*. IEEE, 2011, pp. 884–887.

[54] A. Tamrawi, T.T. Nguyen, J.M. Al-Kofahi, and T.N. Nguyen, "Fuzzy set and cache-based approach for bug triaging," in *Proceedings of the 19th ACM SIGSOFT Symposium and the 13th European Conference on Foundations of Software Engineering*. ACM, 2011, pp. 365–375.

[55] A. Niknafs, J. Denzinger, and G. Ruhe, "A systematic literature review of the personnel assignment problem," in *Proceedings of the International Multiconference of Engineers and Computer Scientists*, 2013.

[56] M.M. Rahman, S. Sohan, F. Maurer, and G. Ruhe, "Evaluation of optimized staffing for feature development and bug fixing," in *Proceedings of the 2010 ACM-IEEE International Symposium on Empirical Software Engineering and Measurement*. ACM, 2010, p. 42.

[57] D. Panagiotou, F. Paraskevopoulos, and L. Stojanovic, Specifications of developer profile, (2010). [Online]. http://www.alert-project.eu/sites/portal2-alert.atosorigin.es/files/content-files/download/Specification%20of%20Developer%20Profile.pdf

[58] N.K. Nagwani and S. Verma, "Rank-me: A Java tool for ranking team members in software bug repositories," *Journal of Software Engineering and Applications*, Vol. 5, 2012, pp. 255–261.

[59] D. Kim, Y. Tao, S. Kim, and A. Zeller, "Where should we fix this bug? a two-phase recommendation model," *IEEE Transactions on Software Engineering*, Vol. 39, No. 11, 2013, pp. 1597–1610.

[60] M. Linares-Vásquez, K. Hossen, H. Dang, H. Kagdi, M. Gethers, and D. Poshyvanyk, "Triaging incoming change requests: Bug or commit history, or code authorship?" in *28th IEEE International Conference on Software Maintenance (ICSM)*. IEEE, 2012, pp. 451–460.

[61] B. Ashok, J. Joy, H. Liang, S.K. Rajamani, G. Srinivasa, and V. Vangala, "DebugAdvisor: a recommender system for debugging," in *Proceedings of the the 7th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering*. ACM, 2009, pp. 373–382.

[62] H. Kagdi and D. Poshyvanyk, "Who can help me with this change request?" in *17th International Conference on Program Comprehension*. IEEE, 2009, pp. 273–277.

[63] J. Anvik and G.C. Murphy, "Determining implementation expertise from bug reports," in *Proceedings of the Fourth International Workshop on Mining Software Repositories.* IEEE Computer Society, 2007, p. 2.

[64] S. Minto and G.C. Murphy, "Recommending emergent teams," in *Fourth International Workshop on Mining Software Repositories.* IEEE, 2007, pp. 5–12.

[65] G. Gousios, E. Kalliamvakou, and D. Spinellis, "Measuring developer contribution from software repository data," in *Proceedings of the 2008 International Working Conference on Mining Software Repositories.* ACM, 2008, pp. 129–132.

[66] T. Zhang, G. Yang, B. Lee, and A.T. Chan, "Guiding bug triage through developer analysis in bug reports," *International Journal of Software Engineering and Knowledge Engineering*, Vol. 26, No. 03, 2016, pp. 405–431.

[67] T. Zhang, G. Yang, B. Lee, and I. Shin, "Role analysis-based automatic bug triage algorithm," IPSJ SIG Technical Report, Tech. Rep., 2012.

[68] X. Xia, D. Lo, X. Wang, and B. Zhou, "Dual analysis for recommending developers to resolve bugs," *Journal of Software: Evolution and Process*, Vol. 27, No. 3, 2015, pp. 195–220.

[69] J. Helming, H. Arndt, Z. Hodaie, M. Koegel, and N. Narayan, "Automatic assignment of work items," in *International Conference on Evaluation of Novel Approaches to Software Engineering.* Springer, 2010, pp. 236–250.

[70] W. Zou, Y. Hu, J. Xuan, and H. Jiang, "Towards training set reduction for bug triage," in *IEEE 35th Annual Computer Software and Applications Conference (COMPSAC).* IEEE, 2011, pp. 576–581.

[71] Z. Lin, F. Shu, Y. Yang, C. Hu, and Q. Wang, "An empirical study on bugassignment automation using Chinese bug data," in *3rd International Symposium on Empirical Software Engineering and Measurement.* IEEE, 2009, pp. 451–455.

[72] S. Banitaan and M. Alenezi, "Tram: An approach for assigning bug reports using their metadata," in *Third International Conference on Communications and Information Technology.* IEEE, 2013, pp. 215–219.

[73] J. Anvik, L. Hiew, and G.C. Murphy, "Who should fix this bug?" in *Proceedings of the 28th International Conference on Software Engineering.* ACM, 2006, pp. 361–370.

[74] M. Sharma, M. Kumari, and V. Singh, "Bug assignee prediction using association rule mining," in *International Conference on Computational Science and Its Applications.* Springer, 2015, pp. 444–457.

[75] J. Park, M. Lee, J. Kim, S. Hwang, and S. Kim, "Costriage: A cost-aware triage algorithm for bug reporting systems," in *Proceedings of the National Conference on Artificial Intelligence*, 2011, p. 139.

[76] M.M. Rahman, G. Ruhe, and T. Zimmermann, "Optimized assignment of developers for fixing bugs an initial evaluation for eclipse projects," in *Proceedings of the 2009 3rd International Symposium on Empirical Software Engineering and Measurement.* IEEE Computer Society, 2009, pp. 439–442.

[77] R.K. Saha, S. Khurshid, and D.E. Perry, "An empirical study of long lived bugs," in *Software Evolution Week–IEEE Conference on Software Maintenance, Reengineering and Reverse Engineering (CSMR-WCRE).* IEEE, 2014, pp. 144–153.