Information and Software Technology 61 (2015) 93–106

Contents lists available at ScienceDirect

Information and Software Technology

journal homepage: www.elsevier.com/locate/infsof

ELBlocker: Predicting blocking bugs with ensemble imbalance learning

Xin Xia[a], David Lo[b], Emad Shihab[c], Xinyu Wang[a,*], Xiaohu Yang[a]

[a] College of Computer Science and Technology, Zhejiang University, Hangzhou, China
[b] School of Information Systems, Singapore Management University, Singapore
[c] Department of Computer Science and Software Engineering, Concordia University, Montreal, QC, Canada

A R T I C L E I N F O

Article history:
Received 18 July 2014
Received in revised form 28 December 2014
Accepted 28 December 2014
Available online 14 January 2015

Keywords:
Blocking bug
Ensemble learning
Imbalance learning

A B S T R A C T

Context: Blocking bugs are bugs that prevent other bugs from being fixed. Previous studies show that blocking bugs take approximately two to three times longer to be fixed compared to non-blocking bugs.
Objective: Thus, automatically predicting blocking bugs early on so that developers are aware of them, can help reduce the impact of or avoid blocking bugs. However, a major challenge when predicting blocking bugs is that only a small proportion of bugs are blocking bugs, i.e., there is an unequal distribution between blocking and non-blocking bugs. For example, in Eclipse and OpenOffice, only 2.8% and 3.0% bugs are blocking bugs, respectively. We refer to this as the *class imbalance phenomenon*.
Method: In this paper, we propose *ELBlocker* to identify blocking bugs given a training data. *ELBlocker* first randomly divides the training data into multiple disjoint sets, and for each disjoint set, it builds a classifier. Next, it combines these multiple classifiers, and automatically determines an appropriate imbalance decision boundary to differentiate blocking bugs from non-blocking bugs. With the imbalance decision boundary, a bug report will be classified to be a blocking bug when its likelihood score is larger than the decision boundary, even if its likelihood score is low.
Results: To examine the benefits of *ELBlocker*, we perform experiments on 6 large open source projects – namely Freedesktop, Chromium, Mozilla, Netbeans, OpenOffice, and Eclipse containing a total of 402,962 bugs. We find that *ELBlocker* achieves F1 and EffectivenessRatio@20% scores of up to 0.482 and 0.831, respectively. On average across the 6 projects, *ELBlocker* improves the F1 and EffectivenessRatio@20% scores over the state-of-the-art method proposed by Garcia and Shihab by 14.69% and 8.99%, respectively. Statistical tests show that the improvements are significant and the effect sizes are large.
Conclusion: ELBlocker can help deal with the class imbalance phenomenon and improve the prediction of blocking bugs. ELBlocker achieves a substantial and statistically significant improvement over the state-of-the-art methods, i.e., Garcia and Shihab's method, SMOTE, OSS, and Bagging.

1. Introduction

Software bugs are prevalent in all stages of the software development and maintenance lifecycle. To manage the reporting of software bugs, most software projects use bug tracking systems, such as Bugzilla. Prior studies showed that the cost of bug fixing in a software system consumed 50–80% of the development and maintenance cost [1]. In 2002, a report from the National Institute of Standards and Technology (NIST) found that software bugs cost $59 billions of the US economy annually [2].

Due to the importance of software bugs, a large number of automated techniques have been proposed to manage and reduce the impact of software bugs. These techniques include bug triaging and developer recommendation [3–6], bug severity/priority assignment [7–9], duplicate bug report detection [10,11], bug fixing time prediction [12–14], and reopened bug prediction [15,16]. In general, the above techniques extract data from bug reports in bug tracking systems to build their prediction models.

In a typical bug fixing process, a tester or a user detects a bug, and submits a bug report[1] to describe the bug in bug tracking systems. Then, the bug is assigned to a corresponding developer to fix. Once the bug is fixed, another developer would verify the fixes, and finally close the bug report. However, in certain cases, the whole fixing process is stalled due to the existence of *a blocking bug* [17]. Blocking bugs refer to bugs that prevent other bugs from being fixed.

* Corresponding author.
E-mail addresses: xxkidd@zju.edu.cn (X. Xia), davidlo@smu.edu.sg (D. Lo), eshihab@cse.concordia.ca (E. Shihab), wangxinyu@zju.edu.cn (X. Wang), yangxh@zju.edu.cn (X. Yang).

[1] In this paper, we use the terms "bug" or "bug report" interchangeably, which refer to an issue report stored in a bug tracking system that is marked as a bug.

http://dx.doi.org/10.1016/j.infsof.2014.12.006
0950-5849/© 2015 Elsevier B.V. All rights reserved.

This means that developers cannot fix their bugs, not because they do not have the ability or resources required to do so, but because the modules they need to fix depend on other modules which still have unresolved (blocking) bugs.

Garcia and Shihab study blocking bugs and find that blocking bugs need 15–40 more days to be fixed compared with non-blocking bugs, i.e., the time to fix blocking bugs is approximately two to three times longer than these of non-blocking bugs [17]. Thus, an automated tool which predicts blocking bugs can help reduce the impact of blocking bugs. Garcia and Shihab further leverage machine learning techniques to predict blocking bugs. They pre-process the training bug reports by using re-sampling strategy [18], and build various classifiers based using the pre-processed bug reports by leveraging various machine learning techniques (e.g., decision trees (C4.5) [19], Naive Bayes [20], kNN [20], and Random Forests [21]). They find random forest achieves the best performance compared to the other techniques. However, the overall performance of all the classifiers were not optimal.

A major challenge in blocking bug prediction is the fact that only a small proportion of bug reports are actually blocking bugs. There is an unequal distribution between blocking and non-blocking bug reports. Only 8.9%, 2.3%, 12.5%, 3.2%, 3.0%, and 2.8% of the bug reports in the whole bug report repository of Freedesktop, Chromium, Mozilla, Netbeans, OpenOffice, and Eclipse projects respectively are blocking bugs. We refer to this as the *class imbalance phenomenon* [22]. Due to the class imbalance phenomenon, predicting blocking bugs with high accuracy is a difficult task.

In this paper we propose *ELBlocker* to predict blocking bugs. *ELBlocker* combines multiple prediction models built on a subset of training bug reports. More specifically, we first divide the training data into multiple disjoint sets, and in each disjoint set, we build a separate classifier (i.e., a prediction model). Next, we combine these multiple classifiers, and automatically determine an appropriate imbalanced decision boundary (or threshold) to differentiate blocking bugs from non-blocking bugs. Traditional machine learning techniques will classify a bug report to be a blocking bug if its likelihood score to be a blocking bug is higher than its likelihood to be a non-blocking bug. With the imbalanced decision boundary, a bug report will be classified to be a blocking bug when its likelihood score is larger than the decision boundary, no matter if its likelihood score to be blocking is low or lower than its likelihood score to be a non-blocking bug. This imbalanced decision boundary is needed since imbalanced data causes a classifier to favor the majority class. Also, since imbalanced data tends to cause poor performance, to boost the performance further, we combine multiple classifiers instead of using a single one following the ensemble learning paradigm [23] that has often been shown effective [22].

To evaluate *ELBlocker*, we use two metrics: F1-score [17,7,15,9] and cost effectiveness [24–27]. F1-score is a summary measure that combines both precision and recall. F1-score is a good evaluation metric when there is enough resources to manually check all the predicted blocking bugs. A higher F1-score means that a method can detect more blocking bugs (true positives) and reduce the time wasted checking non-blocking bugs. Cost effectiveness evaluates prediction performance given a limited resource, e.g., percentage of bug reports to check. In this paper, we use EffectivenessRatio@20% (ER@20%) as the default cost effectiveness metric. The ER@*K*% score of a technique is the ratio of the number of blocking bugs detected by the technique to the number detected by the *perfect technique* that ranks all blocking bugs first followed by non-blocking ones, considering the first *K*% of the bugs appearing in the ranking list of our proposed technique and the perfect technique.

To evaluate the effectiveness of *ELBlocker*, we perform experiments on 6 large open source projects: Freedesktop, Chromium, Mozilla, Netbeans, OpenOffice, and Eclipse containing a total of 402,962 bugs. On average across the 6 projects, *ELBlocker* achieves F1 and ER@20% scores of 0.345 and 0.668, respectively. These results correspond to improvements in the F1 and ER@20% scores over the method proposed in the prior work of Garcia and Shihab by 14.69% and 8.99%, respectively. Statistical tests show that the improvements are significant and the effect sizes are large. We also compare *ELBlocker* with other imbalanced learning algorithms (e.g., SMOTE [28] and one-sided selection (OSS) [29]) and an ensemble learning algorithm (i.e., Bagging [30]), and the results show that our *ELBlocker* achieves the best performance.

The main contributions of this paper are:

1. We consider the class imbalance phenomenon and propose a novel method, named *ELBlocker*, to predict blocking bugs, which utilizes the advantages of ensemble learning to combine multiple prediction models and learn an appropriate decision boundary.
2. We compare our method with Garcia and Shihab's method, SMOTE, OSS, and Bagging on 6 large software projects. The experiment results show that our method achieves substantial and statistically significant improvements over these methods.

The remainder of the paper is organized as follows. We describe some preliminary materials on blocking bug prediction and a motivating example in Section 2. We describe the high-level architecture of *ELBlocker* in Section 3. We elaborate on *ELBlocker* and detail our approach in Section 4. We present our experiment results in Section 5. We present the threats to validity in Section 6. We discuss related work in Section 7. We conclude and mention future work in Section 8.

2. Preliminaries & motivation

In this section, we first introduce some preliminaries about blocking bugs. Next, we provide the technical motivation as to why we need an ensemble of prediction models and why we need to consider the decision boundary.

2.1. Blocking bugs

Blocking bugs refer to bugs that prevent other bugs from being fixed. Garcia and Shihab find that blocking bugs take approximately two to three times longer to be fixed compared to non-blocking bugs [17]. Fig. 1 presents an example of a report of a blocking bug of Mozilla.[2] This bug report specifies that "when content is appended or inserted, the existing implementation of constructing pseudo frames does not work correctly".

Observations and implications. From the blocking bug in Fig. 1, we can observe the following:

1. Blocking bugs need a long time to be fixed. For example, the bug in Fig. 1 took a long time to be fixed. It was created on 2002-06-03, but on 2009-03-26 it was fixed; it took nearly 7 years to fix this bug.
2. Blocking bugs also prevent a number of other bugs from being fixed, and the bugs which depend on the blocking bugs also need a long time to be fixed. The bug in Fig. 1 blocked a number of others bugs in Mozilla, such as bugs 30378, 208305, and 294065, which also delayed the fixing of these bugs. For example, bug 208305 was created on 2003-06-04, but only until 2009-03-26 this bug was finally fixed.

[2] https://bugzilla.mozilla.org/show_bug.cgi?id=148810.