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.

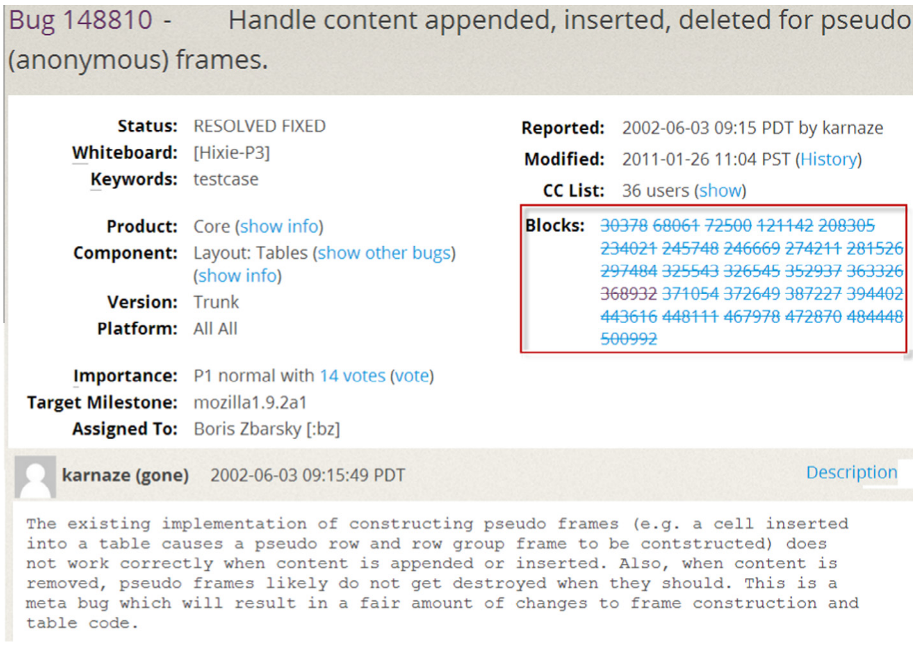
2.1. Blocking bugs

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.



Bug 148810 - Handle content appended, inserted, deleted for pseudo (anonymous) frames.

Status:	RESOLVED FIXED
Whiteboard:	[Hixie-P3]
Keywords:	testcase
Product:	Core (show info)
Component:	Layout: Tables (show other bugs) (show info)
Version:	Trunk
Platform:	All All
Importance:	P1 normal with 14 votes (vote)
Target Milestone:	mozilla1.9.2a1
Assigned To:	Boris Zbarsky [:bz]
Reported:	2002-06-03 09:15 PDT by karnaze
Modified:	2011-01-26 11:04 PST (History)
CC List:	36 users (show)
Blocks:	~~30378~~ ~~68061~~ ~~72500~~ ~~121142~~ ~~208305~~ ~~234021~~ ~~245748~~ ~~246669~~ ~~274211~~ ~~281526~~ ~~297484~~ ~~325543~~ ~~326545~~ ~~352937~~ ~~363326~~ ~~368932~~ ~~371054~~ ~~372649~~ ~~387227~~ ~~394402~~ ~~443616~~ ~~448111~~ ~~467978~~ ~~472870~~ ~~484448~~ ~~500992~~

karnaze (gone) 2002-06-03 09:15:49 PDT Description

```
The existing implementation of constructing pseudo frames (e.g. a cell inserted
into a table causes a pseudo row and row group frame to be contstructed) does
not work correctly when content is appended or inserted. Also, when content is
removed, pseudo frames likely do not get destroyed when they should. This is a
meta bug which will result in a fair amount of changes to frame construction and
table code.
```

Fig. 1. An example of a blocking bug in Mozilla with BugID = 148810.

2.2. Technical motivation

The effectiveness of our *ELBlocker* technique relies on the answers of the following 2 research investigations:

Investigation 1: Does a prediction model built on an ensemble of classifiers that are built on subsets of the training bug reports achieve better performance compared to a model that is built using all of the bug reports?

Investigation 2: Do different decision boundaries (i.e., thresholds) result in significantly different prediction performances?

To answer Investigation 1, we divide the training bug reports from Freedesktop into K (i.e., $K \in (2, \ldots, 10)$) equal-sized disjoint sets. Then, we build a classifier for each of the K subsets. Thus, in total we have K classifiers. For an unlabeled bug report, we input it into each of the K classifiers, and we pick the prediction that has the highest confidence (or likelihood) score. For comparison purposes, we build a classifier (referred to as the baseline classifier) based on all bug reports in the training set, and evaluate it using the same testing set as we do for the K classifiers. In addition, we also build a random prediction classifier (referred to as the random classifier), which randomly predicts a bug to be blocking or not. We use the random forest algorithm [21] to build these classifiers and measure the quality of these two approaches (ensemble vs. single classifier) using precision, recall, and F1-scores using 10-fold cross validation. More specifically, we divide all the bug reports in Freedesktop into 10 equal-sized folds, and we choose 9 folds of the data for training, and evaluate the performance of an approach in the remaining fold; the above process iterates 10 times and the aggregate score across the 10 iterations is reported. Table 1 presents the precision, recall, and F1-score of various approaches (ensemble vs. single classifier). For the ensemble approach, we vary K from 2 to 10. We observe that an ensemble of different classifiers can improve the performance of blocking bug prediction. For example, when we choose $K = 5$ or 7, the F1-score would be 0.377 while this score for the baseline is only 0.307. We also notice that randomly predicting whether a bug report is a blocking bug or not achieves low precision and F1-score values. Thus, a prediction model built on an ensemble of classifiers, which are built on subsets of training bug reports can achieve better performance compared to the model built using all of the bug reports.

To answer Investigation 2, we build a classifier using all training bug reports and predict the label for a new bug report by comparing its likelihood score with a decision boundary (or threshold). If the likelihood score is larger than the threshold, we predict it as a blocking bug; else it is predicted as a non-blocking bug. We vary the threshold from 0.1 to 0.9. We also use random forest to train

Table 1
The precision, recall, and F1-score for various classifiers built on an ensemble of different number of classifiers built on subsets of training bug reports in Freedesktop.

Classifiers	Precision	Recall	F1-score
Baseline	0.628	0.203	0.307
Random	0.023	0.500	0.045
$K = 2$	0.489	0.269	0.347
$K = 3$	0.420	0.274	0.331
$K = 4$	0.409	0.302	0.347
$K = 5$	0.417	0.344	0.377
$K = 6$	0.389	0.351	0.369
$K = 7$	0.367	0.387	0.377
$K = 8$	0.342	0.361	0.351
$K = 9$	0.327	0.436	0.374
$K = 10$	0.318	0.420	0.362

Table 2
The precision, recall, and F1-score for classifiers with different threshold values in Freedesktop.

Classifiers	Precision	Recall	F1-score
Baseline	0.628	0.203	0.307
Threshold = 0.1	0.248	0.724	0.37
Threshold = 0.2	0.352	0.524	0.421
Threshold = 0.3	0.446	0.377	0.409
Threshold = 0.4	0.563	0.276	0.370
Threshold = 0.5	0.628	0.203	0.307
Threshold = 0.6	0.718	0.120	0.206
Threshold = 0.7	0.806	0.059	0.110
Threshold = 0.8	0.857	0.028	0.055
Threshold = 0.9	0.667	0.005	0.009

the classifier, and perform 10-fold cross-validation. Table 2 presents precision, recall, and F1-score for different threshold values. We observe that using different threshold values, the performance is different. For example, if we choose a threshold of 0.2, the F1-score is 0.421 while the F1-score for the baseline is only 0.307; however, if we choose a threshold of 0.9, the F1-score is only 0.009. Thus, a prediction model with different decision boundaries (or thresholds) will result in different performance, and in practice it is necessary for us to determine a good decision boundary.

With the above preliminary experiments, we find that using an ensemble of classifiers built on subsets of a training data, and learning a good decision boundary (or threshold) can improve the performance of blocking bug prediction. Thus, in this paper, we propose *ELBlocker* which combines multiple classifiers, and learns a good threshold, to achieve a good performance.

3. ELBlocker architecture

Fig. 2 presents the architecture of the ELBlocker framework. ELBlocker contains two phases: a model building phase and a prediction phase. In the model building phase, ELBlocker builds a composite model from historical bug reports that have known labels (blocking or non-blocking). In the prediction phase, we apply this model to predict whether an unknown bug report is a blocking bug or not.

Our framework takes as input historical training bug reports with known labels (blocking or non-blocking). Next, it extracts the values of various features from these bug reports (Step 1). In this paper, we use the features that are listed in Table 3. To enable easy comparison with the state-of-the-art, we use the same features that were previously proposed by Garcia and Shihab for blocking bug prediction [17]. For a blocking bug report in the training set, the values of its features are obtained from the contents of its fields right before it is assigned as a blocking bug by the developer. For example, for the feature "priority increase" which denotes whether the priority of this bug has increased, we set the value of this feature as true when the priority really increases before the bug is identified as a blocking bug, else we set the value of this feature as false. Similarly, for features "comment text" and "comment size", we only consider comments that a bug report receives before it is identified as a blocking bug. For a non-blocking bug report in the training set, the values of its features are obtained from the last contents of its fields when we extract them from the bug tracking systems.

Then, ELBlocker randomly divides the collection of training bug reports into multiple equal-sized disjoint subsets ($Sub_1, Sub_2, \ldots, Sub_n$) (Step 2), and for each subset Sub_i, ELBlocker builds a classifier C_i (Step 3). In total, we end up with n classifiers built using n subsets of the training set. Next, these n classifiers are combined to form ELComposer, and we search for the optimal decision boundary (i.e., threshold value) that provides the best F1-scores in the training set (Step 4). ELComposer is a machine learning classifier which assigns labels (in our case: blocking or non-blocking) to a bug report based on its feature values.

After the ELComposer model is constructed, it is then used to predict whether a bug report with an unknown label is a blocking bug or not. For each new bug report, we first extract the values of the same set of features as those considered in the model building step (Step 5). We then input the values of these features into the learned model (Step 6). It will output a prediction result, which is one of the following labels: *blocking* or *non-blocking* (Step 7).

4. ELBlocker approach

ELBlocker is a composite approach that combines multiple classifiers built on the disjoint subsets of the collection of training bug reports. In this section, we first present the definition of subset scores in Section 4.1. Next, we detail the procedure of ELBlocker in Section 4.2.

4.1. Subset scores

We denote the n classifiers that are built on the n equal-sized disjoint sets as $C_1, C_2, \ldots, C_n$. Given an unknown bug report, C_i will output its likelihood scores that this bug is a blocking bug.

Definition 1 (*Subset scores*). Consider a subset of training bug report collection Sub_i, we build a classifier C_i trained on Sub_i. For a new bug report br, we use C_i to get the likelihood that br is a blocking bug. We refer to this likelihood score as the subset score for br, and denote it as $Sub_i(br)$.

There are many classification algorithms that can be used to build a classifier; most of them assign weights to the features and use the presence and absence of each of these features in a new bug report, along with the weights of the features to compute the likelihood of the new bug report to be assigned a particular label (i.e., blocking or non-blocking). By default, we use random forest [21] as the classification algorithms. Random forest is one of the ensemble learning approaches that constructs a number of

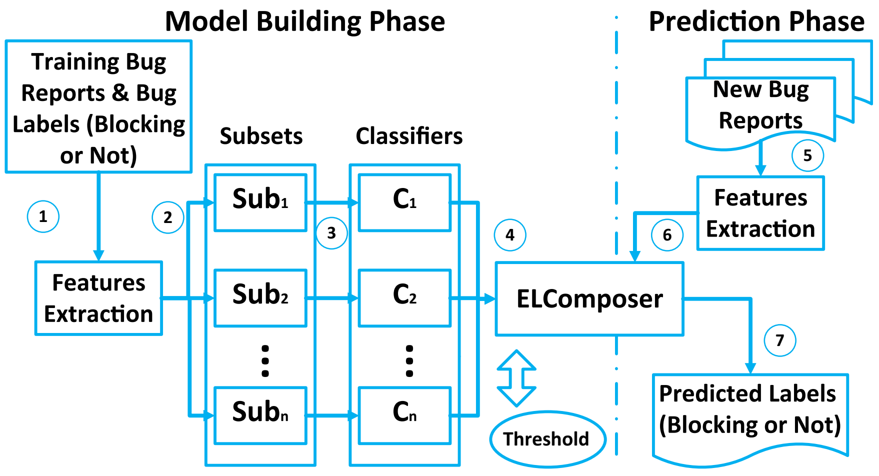


Fig. 2. Overall architecture of ELBlocker.

Table 3
Features for blocking bug prediction used by Garcia and Shihab [17].

Name	Description
Product	Product affected by the bug
Component	Component affected by the bug
Platform	Platform affected by the bug
Severity	The severity of the bug as assigned by the bug reporter. Severity is used as a measure of how much of an impact the bug has
Priority	Indicates how fast the bug should be addressed. In many cases, priority is related to severity
No. CC list	The number of developers that appear in the CC list of the bug report
Description size	Number of words in the description of the bug report
Description text	Textual content appearing in the description field of the bug report. Similar to prior work [17], we convert the description text into a Bayesian score that represents how related the description text is to a blocking bug
Comment size	Number of words in the comments of the bug report
Comment text	Textual content appearing in the comments of the bug report. Similar to prior work [17], we convert the comment text into a Bayesian score that represents how related the comment text is to a blocking bug
Priority increase	Whether the priority of this bug has increased since the time it was reported
Reporter name	Name of the developer who reports this bug
Reporter exp.	Number of previous bug reports filed by the reporter
Reporter block. exp.	Number of previous blocking bug reports filed by the reporter

decision trees [20] by using historical data in the model building phase. In the prediction phase, random forest inputs instances (in our case, bug reports) into the sets of decision trees, and predicts the label of the instances based on the majority voting of the outputs of the set of decision trees.

4.2. ELComposer classifier

Given n classifiers $C_1, C_2, \ldots, C_n$, for a new bug report br, we have n subset scores, i.e., $Sub_1(br), Sub_2(br), \ldots, Sub_n(br)$. An ELComposer classifier computes an average sum of all likelihood scores assigned by the n classifiers and predicts whether a new bug report br is a blocking bug or not based on a threshold score. Definition 2 provides a more mathematical definition of the ELComposer classifier.

Definition 2 (*ELComposer classifier*). Consider n subset classifiers $\{C_1, C_2, \ldots, C_n\}$. A ELComposer classifier composes these n classifiers and assigns a label to a bug report br, denoted as $Label(br)$, as follows:

$$Label(br) = \begin{cases} \text{Blocking}, & \text{if } Comp(br) \geqslant threshold \\ \text{Non-Blocking}, & \text{Otherwise} \end{cases} \quad (1)$$

where,

$$Comp(br) = \frac{\sum_{i=1}^{n} Sub_i(br)}{n} \quad (2)$$

In the above equation, $Sub_i(br)$ is the likelihood score outputted by the ith subset classifier for bug report br. Bug report br is classified as a blocking bug if its composite confidence score $Comp(br)$ is larger or equal than $threshold$ ($threshold$ is the decision boundary); otherwise it is classified as non-blocking.

To deal with the imbalanced data, *ELComposer* introduces a threshold. The value of the threshold varies between 0 and 1. To automatically produce a good *threshold* value for *ELComposer*, we propose a greedy algorithm.

Algorithm 1 presents the training phase of *ELBlocker*, which will fine tune the threshold. We input a bug report collection BR, number of disjoint sets n, and sample size *Sample*. We first divide the training bug reports into n equal-sized disjoint sets, and built subset classifiers $C_1, C_2, \ldots, C_n$ on the n disjoint sets (Lines 8 and 9). Then, we sample a small bug report collection $Samp_{BR}$ according to the sample size *Sample* (Line 10). By default, we randomly select 10% of the bug reports in BR. Next, for each bug report in $Samp_{BR}$, we compute its subset scores $Sub_i(br)$ for each subset classifier according to Definition 1, and we compute the composite confidence score $Comp(br)$ according to Eq. (2) (Lines 11–14). Finally, to tune the best threshold value, we gradually increase *threshold* from 0 to 1 (every time we increase *threshold* by 0.01), and for each sampled bug report br, we predict its label according to Eq. (1); we output *threshold* which maximizes the F1-score for bug reports in $Samp_{BR}$ (Lines 15–19).

Algorithm 1. *EstimateThreshold*: Estimation of Threshold.

```
1: EstimateThreshold($BR, n, Sample$)
2: Input:
3: $BR$: Training Bug Report Collection and Their Labels
4: $n$: Number of Disjoint Sets
5: $Sample$: Sample Size (10% in default)
6: Output: $threshold$
7: Method:
8: Divide the training bug reports into $n$ equal-sized disjoint
     sets;
9: Built subset classifiers $C_1, C_2, \ldots, C_n$ on the $n$ disjoint sets;
10: Sample a bug report collection $Samp_{BR}$ of size $Sample$ from
     $BR$;
11: for all Bug Report $br \in Samp_{BR}$, and Label $l \in L$ do
12:    Compute subset scores $Sub_i(br)$ for each subset
       classifiers according to Definition 1;
13:    Compute $Comp(br)$ according to Eq. (2);
14: end for
15: for all $threshold$ from 0 to 1, every time increase $threshold$
     by 0.01 do
16:    Predict the labels for bug reports in $Samp_{BR}$ according to
       Eq. (1);
17:    Compute the F1-score on $Samp_{BR}$;
18: end for
19: Return $threshold$ which maximizes the F1-score for bug
     reports in $Samp_{BR}$
```

5. Experiments and results

In this section, we evaluate *ELBlocker*. The experimental environment is a Windows 7, 64-bit, Intel(R) Xeon(R) 2.53 GHz server with 32 GB RAM. We present our experiment setup, evaluation metrics, and four research questions in Sections 5.1–5.3, respectively. We then present our experiment results that answer the four research questions.

5.1. Experiment setup

To facilitate a fair comparison with the state-of-the-art, we use the same datasets as Garcia and Shihab [17], which contain bug reports from 6 open source software projects: Freedesktop, Chromium, Mozilla, NetBeans, OpenOffice, and Eclipse. In total we analyze 402,962 bug reports, and among these bug reports, only 18,422 are blocking bugs, which accounts for 4.6% of the total number of bug reports. Table 4 presents the statistics of Garcia and Shihab's data. The columns correspond to the name of projects (**Project**), number of bug reports (**# Bug Reports**), number of blocking bugs (**# Blocking Bugs**), and the percentage of blocking bugs (**% Blocking Bugs**).

To determine whether a bug is a blocking, we use the following strategies:

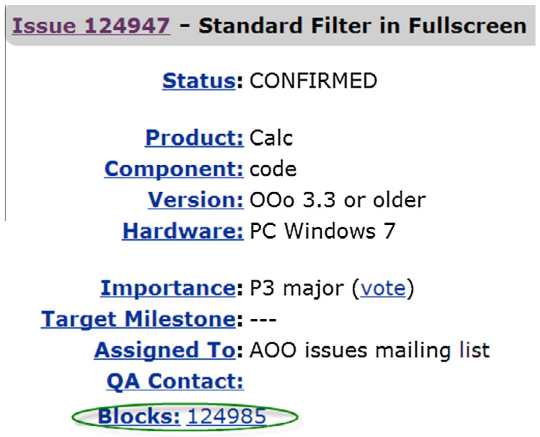
- Mozilla, Eclipse, Freedesktop and NetBeans use Bugzilla as their issue tracking system. In Bugzilla, there is a field named "Blocks" in the bug reports (as shown in Fig. 1). Hence, we use the "Blocks" field in the bug report to identify whether a bug is blocking or not.
- OpenOffice uses a modified version of Bugzilla called IssueTracker. However, similar to Bugzilla, IssueTracker has a field named "Blocks". We also use this field to identify whether or not a bug is blocking. For example, Fig. 3 presents a blocking bug report of OpenOffice. We notice this bug blocks another bug, i.e., bug 124985.
- Chromium has its own issue tracking system in Google code, and it also has a field named "Blocking". We also use this field to identify whether a bug is a blocking bug. For example, Fig. 4 presents a blocking bug report in Chromium. We notice this bug blocks another bug, i.e., bug 365701.

To validate *ELBlocker* and to reduce the training set selection bias, we perform 10-fold cross-validation 100 times and take the average performance. For each 10-fold cross validation we randomly split the data into ten subsets. The random splitting is performed differently for each of the 10-fold cross validations. Cross validation is a standard evaluation setting, which is widely used in software engineering studies, c.f., [15,31–33,7,34,17].

For ELBlocker, we set the number of disjoint sets as 10 by default, i.e., we build 10 subset classifiers and combine them. In a previous study, Garcia and Shihab propose the use of re-sampling to address the imbalance class phenomenon, and they find re-sampling with random forest achieves the best performance. In this paper, we use their approach as one of the baselines. There are also other imbalance learning algorithms; in this paper, we also choose two state-of-the-art algorithms, SMOTE [28] and one-sided selection (OSS) [29]. SMOTE is an over-sampling algorithm, which produces a number of new synthetic minority data by extrapolating values from the K nearest neighbors of each of the original minority class instances (in our case: blocking bugs) [28]. We set the number of neighbors for SMOTE as 5 which is also used in [28], and we increase the number of minority class by 10 times

Table 4
Statistics of collected data.

Project	# Bug reports	# Blocking bugs	% Blocking bugs
Freedesktop	4785	424	8.9
Chromium	39,619	924	2.3
Mozilla	67,597	8476	12.5
NetBeans	76,731	2424	3.2
OpenOffice	83,536	2520	3.0
Eclipse	127,040	3654	2.8
All projects	402,962	18,422	4.6


Issue 124947 - Standard Filter in Fullscreen

Status: CONFIRMED

Product: Calc
Component: code
Version: OOo 3.3 or older
Hardware: PC Windows 7

Importance: P3 major (vote)
Target Milestone: ---
Assigned To: AOO issues mailing list
QA Contact:
Blocks: 124985

Fig. 3. An example of a blocking bug in OpenOffice with BugID = 124947.

(i.e., we create new synthetic blocking bugs 9 times the number of the original blocking bugs). One-sided selection (OSS) is an under-sampling algorithm, which removes noisy and redundant instances of the majority class (in our case: non-blocking bugs), using the one-nearest-neighbor method [29]. Since our ELBlocker uses the idea of ensemble learning, we also select Bagging [30], which is the most similar ensemble learning algorithm to our approach, as a baseline. Bagging samples a subset of a training set by using bootstrap sampling method, and then builds a classifier on the subset. This process iterates n times, and in total, Bagging builds n classifiers. To determine the label of an instance, Bagging uses a majority voting mechanism. In total, we chose 4 baselines, Garcia and Shihab's method, SMOTE, OSS, and Bagging. Notice all of the above approaches could use a different underlying classifier, to make a fair comparison, we use random forest as the underlying classifier for all approaches. For Garcia and Shihab's method, we use the original implementation obtained from the authors. For SMOTE, OSS, Bagging, and random forest, we use their implementations in Weka [35], and we implement ELBlocker on top of Weka [35].

5.2. Evaluation metrics

We use two evaluation metrics: F1-score and cost effectiveness [17,7,15,9,24–27]. These two measures are useful in different situations. F1-score is useful when there is sufficient resource to check all of the predicted blocking bugs. Cost effectiveness is useful when there are limited resources to check a limited amount of bug reports due to a hectic schedule of development, e.g., top 20% number of bugs with highest likelihood scores.

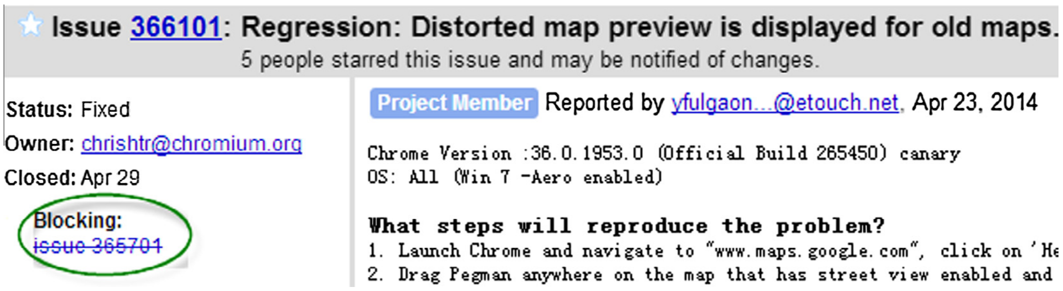
5.2.1. F1-score

There are four possible outcomes for a bug report in the test data: A bug can be classified as a blocking bug when it truly is a blocking bug (true positive, TP); it can be classified as a blocking bug when it is actually a non-blocking bug (false positive, FP); it can be classified as a non-blocking bug when it is actually a blocking bug (false negative, FN); or it can be classified as a non-blocking bug and it truly is a non-blocking bug (true negative, TN). Based on these possible outcomes, precision, recall and F1-score are defined as:

Precision: the proportion of bug reports that are correctly labeled as blocking bugs among those labeled as blocking bugs.

$$P = TP/(TP + FP) \quad (3)$$

Recall: the proportion of blocking bugs that are correctly labeled.


Issue 366101: Regression: Distorted map preview is displayed for old maps.
5 people starred this issue and may be notified of changes.

Status: Fixed
Owner: chrishtr@chromium.org
Closed: Apr 29
Blocking:
~~issue 365701~~

Project Member Reported by yfulgaon...@etouch.net, Apr 23, 2014

```
Chrome Version :36.0.1953.0 (Official Build 265450) canary
OS: All (Win 7 -Aero enabled)

What steps will reproduce the problem?
1. Launch Chrome and navigate to "www.maps.google.com", click on 'He
2. Drag Pegman anywhere on the map that has street view enabled and
```

Fig. 4. An example of a blocking bug in Chromium with BugID = 366101.

$$R = TP/(TP + FN) \tag{4}$$

F1-score: a summary measure that combines both precision and recall – it evaluates if an increase in precision (recall) outweighs a reduction in recall (precision).

$$F = (2 \times P \times R)/(P + R) \tag{5}$$

There is a trade-off between precision and recall. One can increase precision by sacrificing recall (and vice versa). In our framework, we can sacrifice precision (recall) to increase recall (precision), by manually lowering (increasing) the value of the *threshold* parameter in Eq. (1). The trade-off causes difficulties to compare the performance of several prediction models by using only precision or recall alone [20]. For this reason, we compare the prediction results using F1-score, which is a harmonic mean of precision and recall. This follows the setting used in many software analytics studies [33,17,7,15,9].

5.2.2. Cost effectiveness

Cost effectiveness is a widely used evaluation metric for software engineering studies [24–27], which evaluates prediction performance given a cost limit. In our setting, the cost is the number of bug reports to check. By default, we set the number of bugs to check as 20% of the total number of bugs. In this paper, we use EffectivenessRatio@20% (ER@20%) as the default cost effectiveness metric. The ER@20% 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, considering the top 20% bugs.

To compute ER@20% we sort bug reports in the test data that are predicted as blocking bugs based on the confidence level that a prediction technique outputs for each of them. Aside from outputting labels (in our case: blocking or not), ELBlocker, Garcia and Shihab's approach [17], and many other classifiers can also output confidence levels. A bug report with a higher confidence level is deemed to be more likely to be a blocking bug by a prediction technique. We then count the number of blocking bugs that appear in the top 20% of the sorted bug reports. We also count the number of bugs that can be identified by a hypothetical perfect technique that ranks all blocking bugs first, when only the top 20% of the bug reports are checked. Based on these two numbers, ER@20% is computed as:

$$\frac{\text{Number of blocking bugs in the first 20\% of the ranking produced by ELBlocker}}{\text{Number of blocking bugs in the first 20\% of the ranking produced by the perfect technique}} \tag{6}$$

5.3. Research questions

We are interested in answering the following research questions:

Table 5
Cliff's delta and the effectiveness level [37].

Cliff's delta ($|\delta|$)	Effectiveness level
$|\delta| < 0.147$	Negligible
$0.147 \leqslant |\delta| < 0.33$	Small
$0.33 \leqslant |\delta| < 0.474$	Medium
$|\delta| \geqslant 0.474$	Large

RQ1. *How effective is ELBlocker? How much improvement can it achieve over other state-of-the-art methods?*

We need to compare ELBlocker with the state-of-the-art methods. Answer to this research question shows how much ELBlocker advances the state-of-the-art. In a recent study, Garcia and Shihab propose the use of re-sampling with random forest to improve the performance of blocking bug prediction [17]. There are also other imbalance learning and ensemble learning algorithms in the machine learning literature, such as SMOTE, OSS, and Bagging. Thus, to answer this research question, we compare ELBlocker with Garcia and Shihab's method, SMOTE, OSS, and Bagging. We compute F1 and ER@20% scores to evaluate the performance of the 5 approaches on the 6 projects. Also, since we use 100 times 10-fold cross-validation to evaluate each of the methods, we apply the Wilcoxon signed-rank test [36] on the 100 paired data to test whether the improvement of ELBlocker over the baselines are statistically significant.

We also use Cliff's delta (δ) [37], which is a non-parametric effect size measure that quantifies the amount of difference between two groups. In our context, we use Cliff's delta to compare ELBlocker to the baseline approaches. The delta values range from -1 to 1, where $\delta = -1$ or 1 indicates the absence of overlap between two approaches (i.e., all values of one group are higher than the values of the other group, and vice versa), while $\delta = 0$ indicates the two approaches are completely overlapping. Table 5 describes the meaning of different Cliff's delta values and their corresponding effectiveness level [37].

RQ2. *How effective are ELBlocker and the baseline methods when different percentages and numbers of bug reports predicted as blocking bugs are checked?*

By default, we evaluate the performance of the techniques when only the top 20% of the bugs are checked which follows previous studies [27,26,25]. In this RQ, we also investigate the performance of ELBlocker and the baseline methods when different percentages of bug reports are checked. Additionally, we also investigate the effectiveness of ELBlocker and the baseline methods when a fixed budget, i.e., an absolute number of bug reports to check, is specified. Answering this research question can verify whether ELBlocker still improves the baseline methods in different settings. To answer this research question, we plot the ER@*K*% graphs that show the percentages of blocking bugs that can be detected by checking different percentages of bug reports. We also show a table that shows the number of bugs that can be detected by inspecting different numbers of bug reports.

RQ3. *How effective are ELBlocker with different number of subset classifiers?*

By default, we build 10 subset classifiers. In this RQ, we also investigate the performance of ELBlocker with a different number of subset classifiers. Answering this research question can verify the suitable parameter setting range for ELBlocker. To answer this research question, we vary the number of subset classifiers *n* from 2 to 20.

RQ4. *How much time does it take for ELBlocker to run?*

The efficiency of ELBlocker will affect its practical use. Thus, in this research question, we investigate the time efficiency of ELBlocker. We report the model building and prediction time. Model building time refers to the time it takes to convert the training data into an ELBlocker classifier (aka. ELComposer). Prediction time refers to the time it takes for an ELBlocker classifier to predict the label of a bug report. We compare the model building and prediction time of ELBlocker with those of the baseline methods.

5.4. RQ1: performance of ELBlocker

Tables 6–12 presents the experiment results of ELBlocker compared with Garcia and Shihab's method, SMOTE, one-sided selection (OSS), and Bagging, respectively. The statistically significant improvements are marked in bold. The experiment results for Garcia and Shihab's method are a little different than what were reported in their paper [17]. This is the case since the 10-fold cross validation used in our experiments randomly partitions the dataset into 10 sets. Due to the randomness in the process, the resultant sets are different than those produced by the random partitioning performed in Garcia and Shihab's experiments. Also, different from Garcia and Shihab's experiment setup, we run 10-fold cross-validation 100 times, and record the average experiment results.

The F1 and ER@20% scores of ELBlocker vary from 0.136 to 0.482 and 0.473 to 0.831 respectively. On average, across the 6 projects, ELBlocker can achieve F1 and ER@20% scores of 0.345 and 0.668, respectively. The improvement of ELBlocker over Garcia and Shihab's method, SMOTE, OSS, and Bagging are statistically significant. All the *p*-values are less than $2.2e^{-16}$ showing that the improvement is significant. Furthermore, Tables 13 and 14 presents the Cliff's delta values (δ) for ELBlocker compared with Garcia and Shihab's method, SMOTE, OSS, and Bagging in terms of F1-score and ER@20 values. We observe that in all cases the Cliff's delta δ values are greater than 0.474, which shows that the effect size for ELBlocker compared with other approaches is large. This

Table 6
Experiment results of ELBlocker compared with Garcia and Shihab's method (Gar.), SMOTE, one-sided selection (OSS), and Bagging in Freedesktop. Improv.Gar. = Improvement of ELBlocker over Garcia and Shihab's method. Improv.SMO. = Improvement of ELBlocker Over SMOTE. Improv.OSS = Improvement of ELBlocker Over OSS. Improv.Bag. = Improvement of ELBlocker Over Bagging. Statistically significant improvements are highlighted in bold.

Project	Method	Precision	Recall	F1-score	ER@20%
Freedesktop	ELBlocker	0.417	0.430	0.422	0.658
	Gar.	0.268	0.629	0.376	0.608
	Improv.Gar.	**12.23%**			**8.23%**
	SMOTE	0.309	0.477	0.375	0.616
	Improv.SMO.	**12.53%**			**6.85%**
	OSS	0.276	0.488	0.353	0.583
	Improv.OSS	**19.55%**			**12.95%**
	Bagging	0.714	0.012	0.023	0.521
	Improv.Bag.	**1734.78%**			**26.25%**

Table 7
Experiment results of ELBlocker compared with Garcia and Shihab's method (Gar.), SMOTE, one-sided selection (OSS), and Bagging in Chromium.

Project	Method	Precision	Recall	F1-score	ER@20%
Chromium	ELBlocker	0.108	0.184	0.136	0.473
	Gar.	0.437	0.145	0.112	0.444
	Improv.Gar.	**21.43%**			**6.49%**
	SMOTE	0.111	0.082	0.111	0.361
	Improv.SMO.	**22.52%**			**30.74%**
	OSS	0.061	0.189	0.092	0.343
	Improv.OSS	**47.83%**			**37.77%**
	Bagging	0.000	0.000	0.000	0.016
	Improv.Bag.	∞			**2798.58%**

Table 8
Experiment results of ELBlocker compared with Garcia and Shihab's method (Gar.), SMOTE, one-sided selection (OSS), and Bagging in Mozilla.

Project	Method	Precision	Recall	F1-score	ER@20%
Mozilla	ELBlocker	0.437	0.538	0.482	0.628
	Gar.	0.354	0.571	0.437	0.561
	Improv.Gar.	**10.30%**			**11.97%**
	SMOTE	0.417	0.393	0.405	0.551
	Improv.SMO.	**19.01%**			**14.15%**
	OSS	0.335	0.559	0.419	0.534
	Improv.OSS	**15.04%**			**17.60%**
	Bagging	0.536	0.148	0.232	0.543
	Improv.Bag.	**107.76%**			**15.66%**

Table 9
Experiment results of ELBlocker compared with Garcia and Shihab's method (Gar.), SMOTE, one-sided selection (OSS), and Bagging in NetBeans.

Project	Method	Precision	Recall	F1-score	ER@20%
NetBeans	ELBlocker	0.347	0.361	0.354	0.746
	Gar.	0.250	0.392	0.305	0.691
	Improv.Gar.	**16.07%**			**7.92%**
	SMOTE	0.386	0.229	0.287	0.656
	Improv.SMO.	**23.34%**			**13.66%**
	OSS	0.210	0.339	0.259	0.584
	Improv.OSS	**36.68%**			**27.70%**
	Bagging	0.739	0.120	0.207	0.714
	Improv.Bag.	**71.01%**			**4.47%**

Table 10
Experiment results of ELBlocker compared with Garcia and Shihab's method (Gar.), SMOTE, one-sided selection (OSS), and Bagging in OpenOffice.

Project	Method	Precision	Recall	F1-score	ER@20%
OpenOffice	ELBlocker	0.453	0.447	0.450	0.831
	Gar.	0.322	0.490	0.389	0.794
	Improv.Gar.	**15.68%**			**4.60%**
	SMOTE	0.425	0.375	0.398	0.770
	Improv.SMO.	**13.07%**			**7.88%**
	OSS	0.240	0.467	0.317	0.679
	Improv.OSS	**41.96%**			**22.39%**
	Bagging	0.782	0.185	0.299	0.761
	Improv.Bag.	**50.50%**			**9.18%**

Table 11
Experiment results of ELBlocker compared with Garcia and Shihab's method (Gar.), SMOTE, one-sided selection (OSS), and Bagging in Eclipse.

Project	Method	Precision	Recall	F1-score	ER@20%
Eclipse	ELBlocker	0.199	0.262	0.226	0.672
	Gar.	0.159	0.274	0.201	0.586
	Improv.Gar.	**12.44%**			**14.71%**
	SMOTE	0.205	0.120	0.151	0.554
	Improv.SMO.	**49.67%**			**21.25%**
	OSS	0.140	0.257	0.181	0.573
	Improv.OSS	**24.86%**			**17.42%**
	Bagging	0.000	0.000	0.000	0.000
	Improv.Bag.	∞			∞

Table 12
Overall experiment results of ELBlocker compared with Garcia and Shihab's method (Gar.), SMOTE, one-sided selection (OSS), and Bagging across the 6 projects.

Method	Precision	Recall	F1-score	ER@20%
ELBlocker	0.327	0.370	0.345	0.668
Gar.	0.241	0.417	0.303	0.614
(Improv.Gar.)	**14.69%**			**8.99%**
SMOTE	0.309	0.279	0.288	0.585
(Improv.SMO.)	**23.36%**			**15.76%**
OSS	0.211	0.383	0.270	0.549
(Improv.OSS)	**30.98%**			**22.64%**
Bagging	0.462	0.077	0.127	0.426
(Improv.Bag.)	**171.65%**			**56.82%**

Table 13
Cliff's delta δ in terms of F1-score between ELBlocker and prior approaches.

Project	Gar.	SMOTE	OSS	Bagging
Freedesktop	0.995	0.996	1.000	1.000
Chromium	0.999	1.000	1.000	1.000
Mozilla	0.990	1.000	1.000	1.000
Netbeans	1.000	1.000	1.000	1.000
OpenOffice	1.000	1.000	1.000	1.000
Eclipse	1.000	1.000	1.000	1.000

Table 14
Cliff's delta δ in terms of ER@20 between ELBlocker and prior approaches.

Project	Gar.	SMOTE	OSS	Bagging
Freedesktop	0.979	0.990	0.990	0.990
Chromium	0.900	0.990	0.990	1.000
Mozilla	0.990	0.990	0.990	1.000
Netbeans	0.990	0.990	0.990	1.000
OpenOffice	0.990	0.990	0.990	1.000
Eclipse	1.000	1.000	1.000	1.000

indicates that the improvement of ELBlocker over the baseline methods are substantial.

To summarize, on average ELBlocker improves the F1-scores over Garcia and Shihab's method, SMOTE, OSS, and Bagging by 14.69%, 23.36%, 30.98%, and 171.65%, respectively. Also, on average ELBlocker improves the ER@20% over Garcia and Shihab's method, SMOTE, OSS, and Bagging by 8.99%, 15.76%, 22.64%, and 56.82%, respectively. Using the Wilcoxon signed-rank test, we find that the improvements provided by ELBlocker are statistically significantly and have a large effect size.

5.5. RQ2: effectiveness at different K

We investigate the effectiveness of ELBlocker, Garcia and Shihab's method, SMOTE, OSS, and Bagging when different percentages (K%) of bug reports are checked (i.e., K varies from 5 to 100). Figs. 5–16 presents the ER@K% scores of ELBlocker and the baseline methods for various K in Freedesktop, Chromium, Mozilla, Netbeans, OpenOffice, and Eclipse, respectively. We notice ELBlocker is better than the baseline methods for a wide range K values (i.e., a wide range of the number of bug reports to check). For example, in NetBeans, when we set K% as 25%, the ER@25% scores for ELBlocker, Garcia and Shihab's method, SMOTE, OSS, and Bagging are 0.790, 0.746, 0.741, 0.653, and 0.714 respectively; when we set K% to 60%, the ER@60% scores for ELBlocking, Garcia and Shihab's method, SMOTE, OSS, and Bagging are 0.943, 0.916, 0.848, 0.870, and 0.714, respectively.

Table 15 presents the results of ELBlocker and the baseline methods when only 100, 200, and 500 bug reports are inspected. The results show that ELBlocker achieves a substantial improvement over the baseline methods. For example, with a budget of 100 bug reports, on average across the 6 projects, the ER@100 for ELBlocker is 0.520, the ER@100 scores for Garcia and Shihab's method, SMOTE, OSS, and Bagging are 0.466, 0.449, 0.431, and 0.387, respectively. Statistical tests show that the improvements of ELBlocker over the baseline methods are statistically significant.

5.6. RQ3: effectiveness of ELBlocker with different number of subset classifiers

Figs. 17 and 18 presents the F1 and ER@20% scores of ELBlocker with the number of subset classifiers is varying from 2 to 20. The results show that the performance of ELBlocker is generally stable across various numbers of subset classifiers. For example, in OpenOffice, its F1 and ER@20% scores vary from 0.432 to 0.457, and 0.761 to 0.840 when the number of subset classifiers is varied from 2 to 20.

5.7. RQ4: time efficiency

Table 16 presents the model building and prediction time for each of the 6 projects. We notice that the model building and prediction time of ELBlocker are reasonable, e.g., on average, we need about 6.04 s to train a model, and 0.50 s to predict the labels of the instances in the testing set using the model. Notice that the model does not need to be updated all the time. A trained model can be used to label many changes. Compared to other models, ELBlocker has the fastest model building time; this is the case since ELBlocker will build subset classifiers using the disjoint sets of the training data (which are smaller in size), while the remaining methods build a classifier using all the training data. Also, some of them (e.g., Garcia and Shihab's method, SMOTE, and OSS) need to preprocess all of the training data. The prediction time of ELBlocker is longer than the other methods but we believe it is still acceptable (it can label thousands of bug reports in seconds).

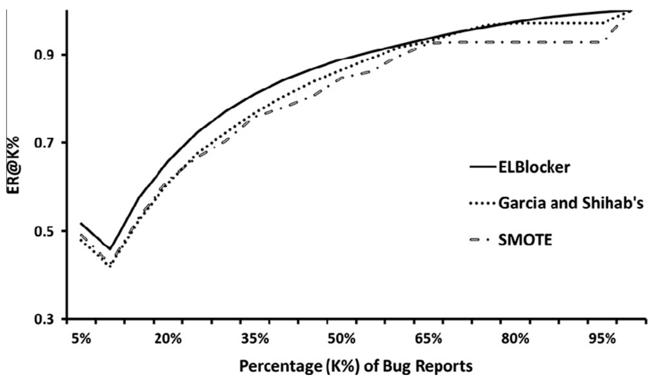


Fig. 5. EffectivenessRatio@K% (ER@K%) of ELBlocking, Garcia and Shihab's method, and SMOTE for various K in Freedesktop.

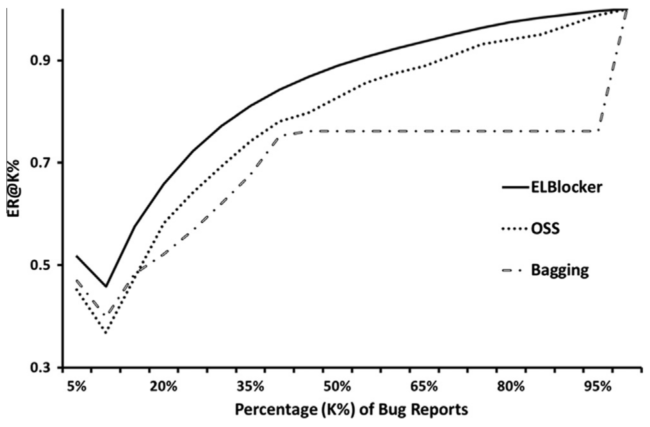


Fig. 6. EffectivenessRatio@*K*% (ER@*K*%) of ELBlocking, OSS, and Bagging for various *K* in Freedesktop.

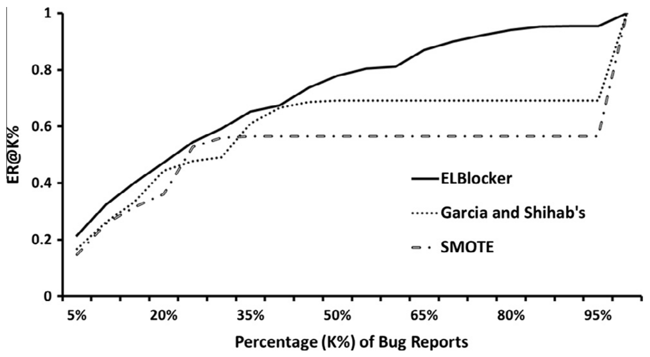


Fig. 7. EffectivenessRatio@*K*% (ER@*K*%) of ELBlocking, Garcia and Shihab's method, and SMOTE for various *K* in Chromium.

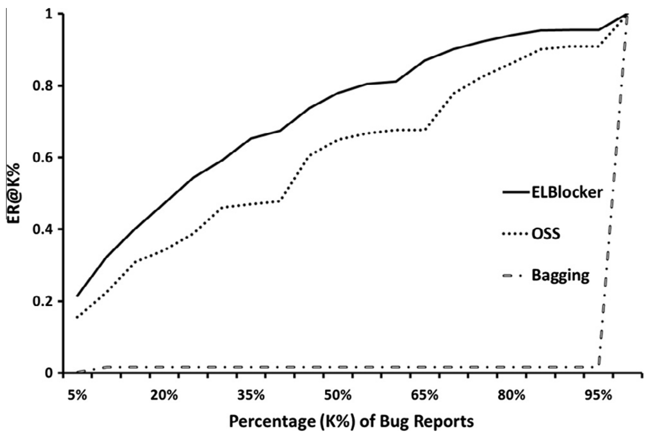


Fig. 8. EffectivenessRatio@*K*% (ER@*K*%) of ELBlocking, OSS, and Bagging for various *K* in Chromium.

6. Discussion

6.1. Usefulness of ELBlocker

In practice, if developers do not have a tool to help identify blocking bugs, they would determine whether a bug is blocking in an almost random way. To show the usefulness of our proposed approach, we also compare ELBlocker with random prediction [17]. In random prediction, we randomly predict a bug to be a blocking or a non-blocking bug according to the ratio of blocking bugs to total bugs in the bug report collections. The precision for random prediction is the percentage of blocking bugs in the data set. Since random prediction is a random classifier with two possible outcomes (e.g., blocking or non-blocking), its recall is 0.5.

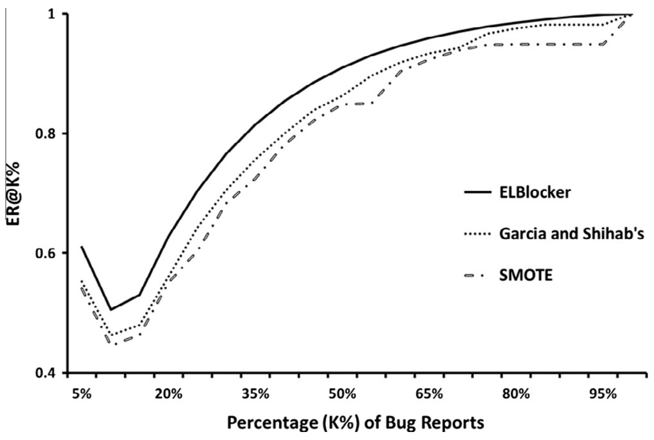


Fig. 9. EffectivenessRatio@*K*% (ER@*K*%) of ELBlocking, Garcia and Shihab's method, and SMOTE for various *K* in Mozilla.

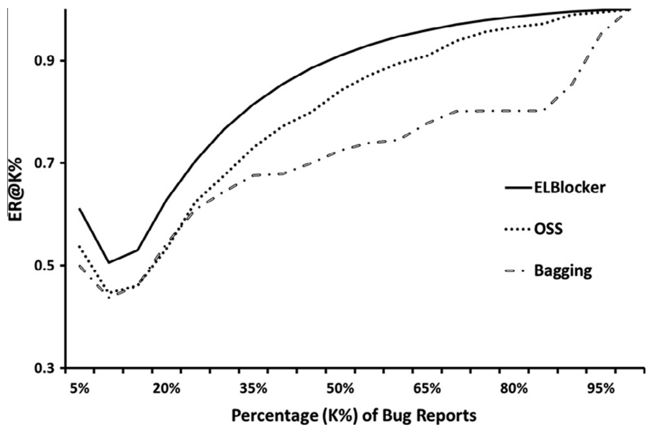


Fig. 10. EffectivenessRatio@*K*% (ER@*K*%) of ELBlocking, OSS, and Bagging for various *K* in Mozilla.

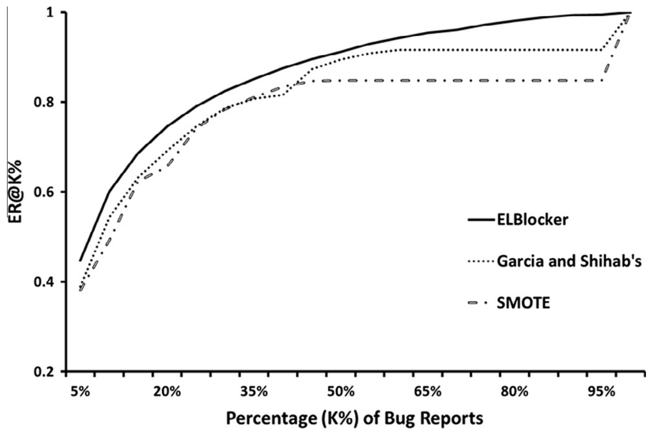


Fig. 11. EffectivenessRatio@*K*% (ER@*K*%) of ELBlocking, Garcia and Shihab's method, and SMOTE for various *K* in Netbeans.

Table 17 presents the experimental results for ELBlocker compared with random prediction. The precision values achieved by ELBlocker are better than the precision values of random predictor for all the projects. ELBlocker precision values range from 0.108 to

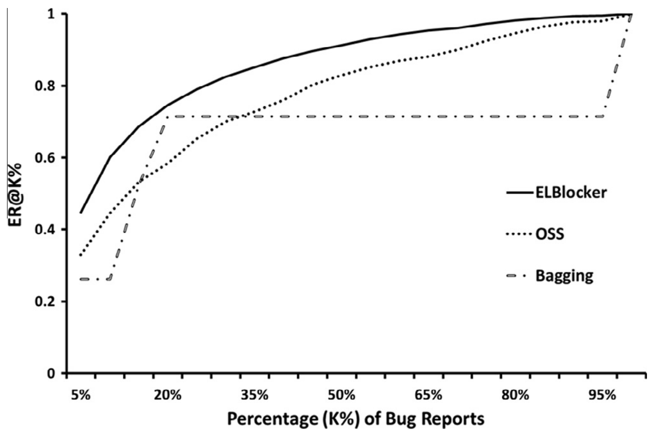


Fig. 12. EffectivenessRatio@*K*% (ER@*K*%) of ELBlocking, OSS, and Bagging for various *K* in Netbeans.

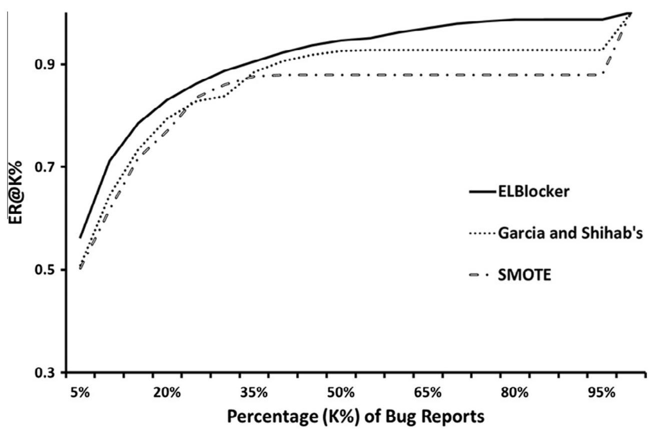


Fig. 13. EffectivenessRatio@*K*% (ER@*K*%) of ELBlocking, Garcia and Shihab's method, and SMOTE for various *K* in OpenOffice.

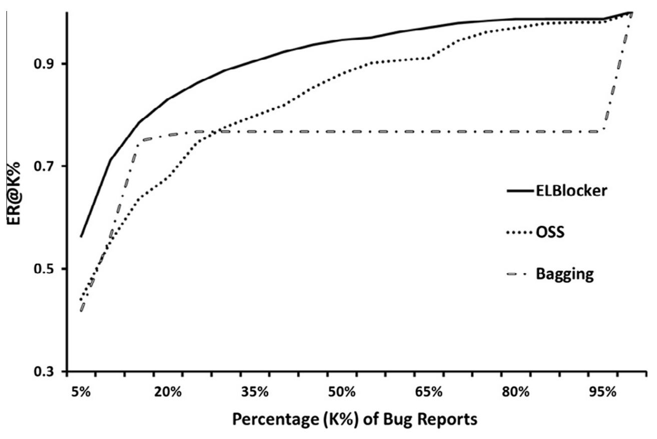


Fig. 14. EffectivenessRatio@*K*% (ER@*K*%) of ELBlocking, OSS, and Bagging for various *K* in OpenOffice.

0.453. Comparing these results with those of random prediction (0.023–0.125), we observe that ELBlocker provides a ∼4 to ∼18 fold improvement over random prediction in terms of precision.

We notice the recall values for ELBlocker are lower than the values of random prediction. In practice, there is a trade-off between precision and recall. One can increase precision by sacrificing recall (and vice versa). In ELBlocker, we can sacrifice precision (recall) to increase recall (precision), by manually lowering (increasing) the

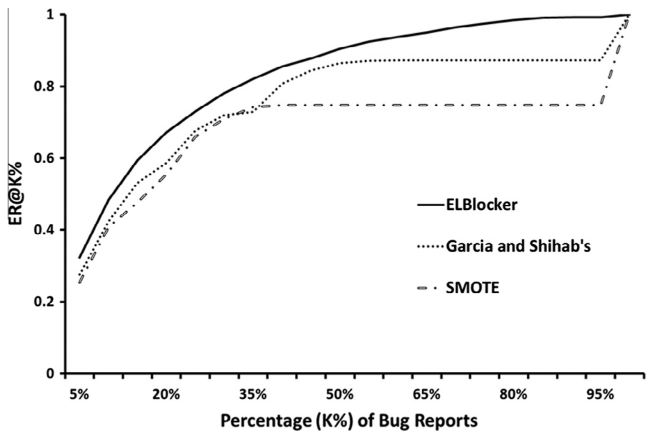


Fig. 15. EffectivenessRatio@*K*% (ER@*K*%) of ELBlocking, Garcia and Shihab's method, and SMOTE for various *K* in Eclipse.

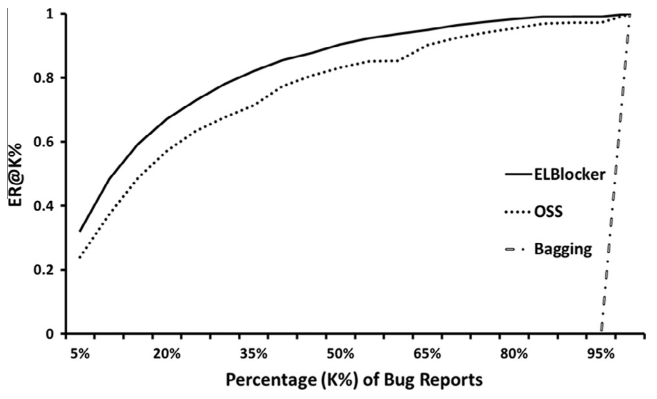


Fig. 16. EffectivenessRatio@*K*% (ER@*K*%) of ELBlocking, OSS, and Bagging for various *K* in Eclipse.

value of the *threshold* parameter in Eq. (1). The trade-off causes difficulties to compare the performance of several prediction models by using only precision or recall alone [20]. Thus, F1-score which is a trade-off between precision and recall, is used as the main metric to evaluate the performance of ELBlocker and random prediction. The F1-score values of ELBlocker are better than the F1-score values of random prediction. Our F1-score values range from 0.136 to 0.450, whereas the F1-score values of random prediction range from 0.045 to 0.201. The improvement ratios of our F1-score values vary from ∼3 to ∼10 folds.

Similar to the previous metrics, the ER@20 values achieved by the decision trees of five of the six projects are better than the ER@20 values of random prediction. The only one exception is Chromium with ER@20 values (i.e., 0.473) slightly below the ER@20 of random prediction (i.e., 0.5). We notice that ELBlocker provides a ∼0.9 to ∼1.7 fold improvement over random prediction in terms of ER@20 values.

To summarize, in most cases (except for Chromium) ELBlocker achieves a much better performance compared to random prediction, which improves the F1-score and ER@20 values by ∼2 to ∼9, and ∼0.9 to ∼1.7 folds, respectively. In practice, developers could deploy our proposed tool to help identify blocking bugs. Although the prediction accuracy of our proposed approach is not perfect, it shows much better performance than random prediction.

6.2. ELBlocker vs. Bagging + Random Forest

In our previous section, we use decision trees as the default underlying classifier for Bagging. In this section, we use random

Table 15
Cost effectiveness of ELBlocker and the baseline methods with different numbers of bug reports to check (100, 200, and 500 bug reports, respectively). EL. = ELBlocker, Gar. = Garcia and Shihab's method, SMO. = SMOTE, Bag. = Bagging.

Project	ER@100					ER@200				
	EL.	Gar.	SMO.	OSS	Bag.	EL.	Gar.	SMO.	OSS	Bag.
Freedesktop	0.674	0.623	0.621	0.594	0.542	0.852	0.820	0.781	0.783	0.762
Chromium	0.142	0.109	0.101	0.092	0.003	0.217	0.168	0.150	0.155	0.003
Mozilla	0.807	0.739	0.712	0.668	0.628	0.693	0.640	0.617	0.612	0.551
NetBeans	0.516	0.437	0.455	0.398	0.479	0.389	0.332	0.338	0.291	0.308
OpenOffice	0.688	0.627	0.587	0.587	0.669	0.508	0.463	0.437	0.414	0.461
Eclipse	0.297	0.261	0.216	0.244	0.000	0.256	0.219	0.188	0.206	0.000
Average	0.520	0.466	0.449	0.431	0.387	0.486	0.440	0.419	0.410	0.347

Project	ER@500				
	EL.	Gar.	SMO.	OSS	Bag.
Freedesktop	1.000	1.000	1.000	1.000	1.000
Chromium	0.368	0.325	0.316	0.230	0.016
Mozilla	0.550	0.508	0.470	0.464	0.468
NetBeans	0.506	0.449	0.439	0.380	0.262
OpenOffice	0.602	0.543	0.519	0.465	0.550
Eclipse	0.271	0.231	0.210	0.215	0.000
Average	0.550	0.510	0.492	0.459	0.383

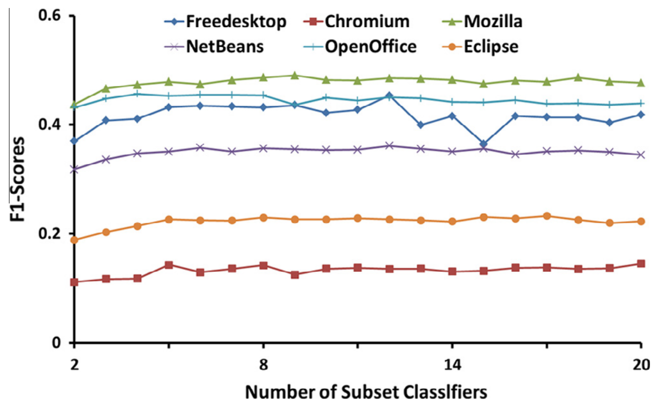


Fig. 17. F1-scores of ELBlocking with number of subset classifiers varied from 2 to 20.

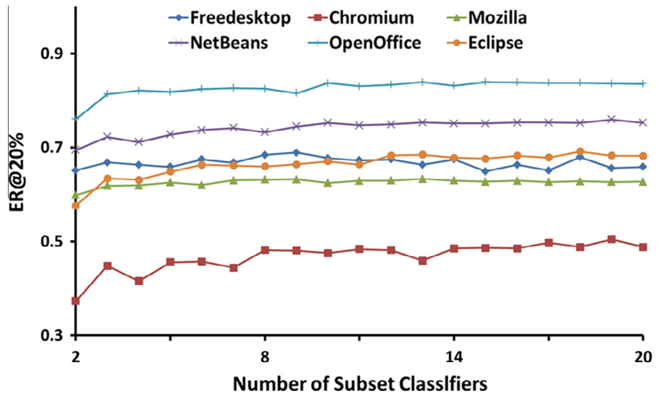


Fig. 18. EffectivenessRatio@20% (ER@20%) of ELBlocking with number of subset classifiers varied from 2 to 20.

forest as the underlying classifier for Bagging, we denote it as Bagging + RA. Table 18 presents the experimental results for ELBlocker compared with Bagging + RA. Once again, we notice that ELBlokcer achieves a better performance than Bagging + RA. On average, across the 6 projects, ELBlocker improves the F1-score and ER@20 of Bagging + RA by 175.05% and 12.83%, respectively.

Table 16
Model building time, and prediction time for ELBlocker, Garcia and Shihab's method, SMOTE, OSS, and Bagging (in seconds).

Project	ELBlocker	Garcia and Shihab's	SMOTE	OSS	Bagging
Prediction time					
Freedesktop	1.18	1.01	2.50	1.78	1.65
Chromium	2.10	2.25	5.52	24.76	14.52
Mozilla	9.24	9.30	169.39	335.03	86.30
NetBeans	4.66	5.85	21.93	119.06	43.88
OpenOffice	6.46	6.32	28.35	106.04	49.62
Eclipse	12.62	16.76	53.40	342.51	151.60
Average	6.04	6.92	46.85	154.86	57.93
Model build time					
Freedesktop	0.02	0.01	0.02	0.02	0.05
Chromium	0.28	0.02	0.02	0.02	0.01
Mozilla	0.69	0.05	0.05	0.03	0.02
NetBeans	0.63	0.05	0.08	0.03	0.01
OpenOffice	0.48	0.03	0.05	0.05	0.02
Eclipse	0.89	0.05	0.06	0.06	0.02
Average	0.50	0.04	0.05	0.04	0.02

6.3. Threats to validity

Threats to internal validity relates to errors in our code and experiment bias. We have double checked our code, still there could be errors that we did not notice. To reduce training set selection bias, we run 10-fold cross-validation 100 times, and record the average performance.

Threats to external validity relates to the generalizability of our results. We have analyzed 402,962 bug reports from 6 projects. In the future, we plan to reduce this threat further by analyzing even more bug reports from additional software projects.

Threats to construct validity refers to the suitability of our evaluation measures. We use F1-score and cost effectiveness which are also used by past studies to evaluate the effectiveness of various automated software engineering techniques

Table 17
Experiment results for ELBlocker compared with random prediction.

Project	ELBlocker				Random prediction			
	Precision	Recall	F1-score	ER@20	Precision	Recall	F1-score	ER@20
Freedesktop	0.417	0.430	0.422	0.658	0.023	0.500	0.045	0.500
Chromium	0.108	0.184	0.136	0.473	0.028	0.500	0.053	0.500
Mozilla	0.437	0.538	0.432	0.628	0.086	0.500	0.150	0.500
NetBeans	0.347	0.361	0.354	0.746	0.125	0.500	0.201	0.500
OpenOffice	0.453	0.447	0.450	0.831	0.032	0.500	0.059	0.500
Eclipse	0.199	0.262	0.226	0.672	0.030	0.500	0.057	0.500

Table 18
Experiment results for ELBlocker compared with Bagging + RA.

Project	ELBlocker				Bagging + RA			
	Precision	Recall	F1-score	ER@20	Precision	Recall	F1-score	ER@20
Freedesktop	0.417	0.430	0.422	0.658	0.702	0.140	0.233	0.597
Chromium	0.108	0.184	0.136	0.473	0.393	0.013	0.025	0.396
Mozilla	0.437	0.538	0.432	0.628	0.701	0.206	0.319	0.527
NetBeans	0.347	0.361	0.354	0.746	0.803	0.093	0.172	0.694
OpenOffice	0.453	0.447	0.450	0.831	0.854	0.213	0.341	0.774
Eclipse	0.199	0.262	0.226	0.672	0.544	0.024	0.050	0.593

[33,17,7,15,9,15,24–27]. Thus, we believe there is little threat to construct validity

7. Related work

7.1. Blocking bug prediction

Garcia and Shihab are the first to propose the problem of blocking bug prediction [17]. They analyze 402,962 bug reports from 6 different open source software communities, and they find that blocking bugs take approximately two to three times longer to be fixed compared to non-blocking bugs. To predict the blocking bugs, they first re-sample the training set to make the number of blocking bugs and non-blocking bugs the same, next a machine learning technique (e.g., random forest) is used to predict whether a new bug is a blocking bug. Our work extends their work by proposing a novel ensemble learning based approach named ELBlocker, which combines multiple classifiers built on different subsets of the training set. The experiment results show ELBlockers achieves a substantial and statistically significant improvement over Garcia and Shihab's method.

7.2. Other studies on bug report management

There have been a number of studies on re-opened bug prediction [15,38,16]. Shihab et al. study re-opened bugs on Eclipse, Apache HTTP, and OpenOffice, and propose prediction models based on decision trees [15]. They also use re-sampling methods to pre-process the training data. Xia et al. investigate the performance of different machine learning methods to predict re-opened bugs, and they find Bagging with decision tree achieves the best performance [38]. In later work, Xia et al. extract more textual features from the bug reports, and propose ReopenPredictor, which combines different classifiers to further improve the performance of reopened bug prediction [39]. Zimmermann et al. also investigate re-opened bugs in Windows [16]. They perform a survey to identify possible root causes of re-opened bugs, and build a logistic regression model to determine the impact of various metrics. Our work is different from the prior work since we focus on different types of bugs, blocking bugs. Since the level of class imbalance in blocking bug prediction is more serious than re-opened bug prediction, blocking bug prediction is a more difficult problem.

Also, there have been a number of studies on bug triaging and developer recommendation [3–6], bug severity/priority assignment [7–9,40], duplicated bug report detection [10,11,41–43], and bug fixing time prediction [12–14]. Our work is orthogonal to the above studies; in this paper, we solve a different problem, blocking bug prediction.

7.3. Imbalanced learning and ensemble learning

There have been a number of imbalanced learning techniques in the machine learning literatures [22,29,28]. Some techniques use majority under-sampling, which addresses the phenomenon of class imbalance by reducing the number of majority instances. One-sided selection (OSS), which removes noisy and redundant instances of the majority class by using the one-nearest-neighbor method, which is one of the state-of-the-art methods [29]. Other techniques use minority over-sampling, which addresses the phenomenon of class imbalance by increasing the number of minority instances. SMOTE which produces new synthetic minority data by extrapolating values from the K nearest neighbors of each of the original minority class instances is one of the state-of-the-art technique [28].

There are a number of ensemble learning techniques in the machine learning literature [30,21,44]. Bagging, which also builds classifiers on subsets of a training data, is one of the ensemble learning techniques which is most similar to our approach. Bagging first samples a subset of training set by using bootstrap sampling method, and builds a classifier on the subset. This process repeats n times, and in total, Bagging builds n classifiers. To determine the label of an instance, Bagging uses a majority voting mechanism. Our ELBlocker is different from Bagging since we do not use bootstrap sampling to select the subsets, rather we randomly divide the training set into multiple disjoint subsets, and we build a classifier on each of these subsets. Moreover, after we build multiple classifiers, we automatically detect an appropriate imbalanced decision threshold; Bagging does not consider a threshold.

In this paper, we choose OSS, SMOTE, and Bagging as baseline methods, and we compare ELBlocker with them. The experiment results show that ELBlocker achieves a substantial and statistically significant improvement over OSS, SMOTE, and Bagging.

8. Conclusion and future work

In this paper, we propose a novel blocking bug prediction approach named *ELBlocker*, which leverages ensemble learning techniques. Considering the *class imbalance phenomenon*, we first divide the training set into multiple disjoint sets, and in each disjoint set, we build a classifier. Next, we combine these multiple classifiers, and automatically determine an appropriate decision boundary to separate blocking bugs from non-blocking bugs. Our experiment on 6 large projects containing a total of 402,962 bug reports show that ELBlocker achieves a substantial and statistically significant improvement over the baseline methods, i.e., Garcia and Shihab's method, SMOTE, OSS, and Bagging. On average ELBlocker improves the F1-scores of Garcia and Shihab's method, SMOTE, OSS, and Bagging by 14.69%, 23.36%, 30.98%, and 171.65%, respectively; and ELBlocker improves the ER@20% of these methods by 8.99%, 15.76%, 22.64%, and 56.82%, respectively.

Considering the class imbalance phenomenon in the bug report collections, predicting blocking bugs is a difficult problem. Our work is one of the first works on identifying blocking bugs. Although the performance of our ELBlocker is not perfect, we hope our work will inspire other researchers to develop more advanced techniques to identify blocking bugs. In the future, we plan to evaluate ELBlocker on datasets from more software projects, and apply some information retrieval and text mining techniques such as topic modeling [45], and extract more features (e.g., code features) to improve the prediction performance further. We also plan to develop an automated tool to tell developers not only whether a bug is a blocking bug, but also the other bugs which are blocked by the blocking bug.

Acknowledgments

This research was supported by the National Basic Research Program of China (the 973 Program) under Grant 2015CB352201, and National Key Technology R&D Program of the Ministry of Science and Technology of China under Grant 2014BAH24F02, and the Fundamental Research Funds for the Central Universities.

References

[1] J.S. Collofello, S.N. Woodfield, Evaluating the effectiveness of reliability-assurance techniques, J. Syst. Softw. 9 (3) (1989) 191–195.
[2] G. Tassey, The economic impacts of inadequate infrastructure for software testing, National Institute of Standards and Technology, RTI Project 7007 (011).
[3] J. Anvik, L. Hiew, G.C. Murphy, Who should fix this bug?, in: Proceedings of the 28th International Conference on Software Engineering, ACM, 2006, pp 361–370.
[4] X. Xia, D. Lo, X. Wang, B. Zhou, Accurate developer recommendation for bug resolution, in: 20th Working Conference on Reverse Engineering (WCRE), IEEE, 2013, pp. 72–81.
[5] G. Jeong, S. Kim, T. Zimmermann, Improving bug triage with bug tossing graphs, in: Proceedings of 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.
[6] W. Wu, W. Zhang, Y. Yang, 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.
[7] T. Menzies, A. Marcus, Automated severity assessment of software defect reports, in: IEEE International Conference on Software Maintenance. ICSM 2008, IEEE, 2008, pp. 346–355.
[8] A. Lamkanfi, S. Demeyer, E. Giger, B. Goethals, Predicting the severity of a reported bug, in: 7th IEEE Working Conference on Mining Software Repositories (MSR), IEEE, 2010, pp. 1–10.
[9] Y. Tian, D. Lo, C. Sun, Drone: predicting priority of reported bugs by multi-factor analysis, in: 29th IEEE International Conference on Software Maintenance (ICSM), IEEE, 2013, pp. 200–209.
[10] C. Sun, D. Lo, S.-C. Khoo, J. Jiang, Towards more accurate retrieval of duplicate bug reports, in: Proceedings of the 26th IEEE/ACM International Conference on Automated Software Engineering, IEEE Computer Society, 2011, pp. 253–262.
[11] N. Jalbert, W. Weimer, Automated duplicate detection for bug tracking systems, in: IEEE International Conference on Dependable Systems and Networks With FTCS and DCC. DSN 2008, IEEE, 2008, pp. 52–61.
[12] L. Marks, Y. Zou, A.E. Hassan, Studying the fix-time for bugs in large open source projects, in: Proceedings of the 7th International Conference on Predictive Models in Software Engineering, ACM, 2011, p. 11.
[13] C. Weiss, R. Premraj, T. Zimmermann, A. Zeller, How long will it take to fix this bug?, in: Proceedings of the Fourth International Workshop on Mining Software Repositories, IEEE Computer Society, 2007, p 1.
[14] L.D. Panjer, Predicting eclipse bug lifetimes, in: Proceedings of the Fourth International Workshop on Mining Software Repositories, IEEE Computer Society, 2007, p. 29.
[15] E. Shihab, A. Ihara, Y. Kamei, W.M. Ibrahim, M. Ohira, B. Adams, A.E. Hassan, K.-i. Matsumoto, Studying re-opened bugs in open source software, Empirical Softw. Eng. 18 (5) (2013) 1005–1042.
[16] T. Zimmermann, N. Nagappan, P.J. Guo, B. Murphy, Characterizing and predicting which bugs get reopened, in: 34th International Conference on Software Engineering (ICSE), IEEE, 2012, pp. 1074–1083.
[17] H.V. Garcia, E. Shihab, Characterizing and predicting blocking bugs in open source projects, in: Proceedings of the Eleventh International Workshop on Mining Software Repositories, IEEE Computer Society, 2014.
[18] R. Barandela, R.M. Valdovinos, J.S. Sánchez, F.J. Ferri, The imbalanced training sample problem: Under or over sampling?, in: Structural, Syntactic, and Statistical Pattern Recognition, Springer, 2004, pp 806–814.
[19] J.R. Quinlan, Induction of decision trees, Mach. Learn. 1 (1) (1986) 81–106.
[20] J. Han, M. Kamber, J. Pei, Data Mining: Concepts and Techniques, Morgan Kaufmann, 2006.
[21] L. Breiman, Random forests, Mach. Learn. 45 (1) (2001) 5–32.
[22] H. He, E.A. Garcia, Learning from imbalanced data, IEEE Trans. Knowl. Data Eng. 21 (9) (2009) 1263–1284.
[23] T.G. Dietterich, Ensemble methods in machine learning, in: Multiple Classifier Systems, Springer, 2000, pp. 1–15.
[24] E. Arisholm, L.C. Briand, M. Fuglerud, Data mining techniques for building fault-proneness models in telecom java software, in: The 18th IEEE International Symposium on Software Reliability. ISSRE'07, IEEE, 2007, pp. 215–224.
[25] F. Rahman, D. Posnett, P. Devanbu, Recalling the imprecision of cross-project defect prediction, in: Proceedings of the ACM SIGSOFT 20th International Symposium on the Foundations of Software Engineering, ACM, 2012, p. 61.
[26] F. Rahman, D. Posnett, I. Herraiz, P. Devanbu, Sample size vs. bias in defect prediction, in: Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering, ACM, 2013, pp. 147–157.
[27] T. Jiang, L. Tan, S. Kim, Personalized defect prediction, in: IEEE/ACM 28th International Conference on Automated Software Engineering (ASE), IEEE, 2013, pp. 279–289.
[28] N.V. Chawla, K.W. Bowyer, L.O. Hall, W.P. Kegelmeyer, Smote: synthetic minority over-sampling technique, J. Artif. Intell. Res. 16 (1) (2002) 321–357.
[29] M. Kubat, S. Matwin, et al., Addressing the curse of imbalanced training sets: one-sided selection, in: Proceedings of the Fourteenth International Conference on Machine Learning (ICML), vol. 97, 1997, pp. 179–186.
[30] L. Breiman, Bagging predictors, Mach. Learn. 24 (2) (1996) 123–140.
[31] X. Xia, Y. Feng, D. Lo, Z. Chen, X. Wang, Towards more accurate multi-label software behavior learning, in: Software Evolution Week-IEEE Conference on Software Maintenance, Reengineering and Reverse Engineering (CSMR-WCRE), 2014, IEEE, 2014, pp. 134–143.
[32] X. Xia, D. Lo, X. Wang, B. Zhou, Tag recommendation in software information sites, in: Proceedings of the Tenth International Working Conference on Mining Software Repositories, IEEE Press, 2013, pp. 287–296.
[33] Y. Tian, J. Lawall, D. Lo, Identifying linux bug fixing patches, in: 34th International Conference on Software Engineering (ICSE), IEEE, 2012, pp. 386–396.
[34] F. Thung, D. Lo, J. Lawall, Automated library recommendation, in: 20th Working Conference on Reverse Engineering (WCRE), IEEE, 2013, pp. 182–191.
[35] M. Hall, E. Frank, G. Holmes, B. Pfahringer, P. Reutemann, I.H. Witten, The weka data mining software: an update, ACM SIGKDD Explor. Newslett. 11 (1) (2009) 10–18.
[36] F. Wilcoxon, Individual comparisons by ranking methods, Biometrics 1 (6) (1945) 80–83.
[37] N. Cliff, Ordinal Methods for Behavioral Data Analysis, Psychology Press, 2014.
[38] X. Xia, D. Lo, X. Wang, X. Yang, S. Li, 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.
[39] X. Xia, D. Lo, E. Shihab, X. Wang, B. Zhou, Automatic, high accuracy prediction of reopened bugs, Autom. Softw. Eng. (2014) 1–35
[40] Y. Tian, D. Lo, X. Xia, C. Sun, Automated prediction of bug report priority using multi-factor analysis, Empirical Softw. Eng. (2014) 1–30
[41] X. Wang, L. Zhang, T. Xie, J. Anvik, J. Sun, An approach to detecting duplicate bug reports using natural language and execution information, in: Proceedings of the 30th International Conference on Software Engineering, ACM, 2008, pp. 461–470.
[42] C. Sun, D. Lo, X. Wang, J. Jiang, S.-C. Khoo, A discriminative model approach for accurate duplicate bug report retrieval, Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering, vol. 1, ACM, 2010, pp. 45–54.
[43] A.T. Nguyen, T.T. Nguyen, T.N. Nguyen, D. Lo, C. Sun, Duplicate bug report detection with a combination of information retrieval and topic modeling, in: Proceedings of the 27th IEEE/ACM International Conference on Automated Software Engineering (ASE), IEEE, 2012, pp. 70–79.
[44] Y. Freund, R.E. Schapire, A decision-theoretic generalization of on-line learning and an application to boosting, J. Comput. Syst. Sci. 55 (1) (1997) 119–139.
[45] D.M. Blei, A.Y. Ng, M.I. Jordan, Latent Dirichlet allocation, J. Mach. Learn. Res. 3 (2003) 993–1022.