Yang XL, Lo D, Xia X *et al.* High-impact bug report identification with imbalanced learning strategies. JOURNAL OF COMPUTER SCIENCE AND TECHNOLOGY 32(1): 181–198 Jan. 2017. DOI 10.1007/s11390-017-1713-3

High-Impact Bug Report Identification with Imbalanced Learning Strategies

Xin-Li Yang [1], David Lo [2], *Member, ACM, IEEE*, Xin Xia [1,*], *Member, CCF, ACM, IEEE*, Qiao Huang [1] and Jian-Ling Sun [1], *Member, CCF, ACM*

[1] *College of Computer Science and Technology, Zhejiang University, Hangzhou 310027, China*
[2] *School of Information Systems, Singapore Management University, Singapore, Singapore*

E-mail: zdyxl@zju.edu.cn; davidlo@smu.edu.sg; {xxia, tkdsheep, sunjl}@zju.edu.cn

Received March 19, 2016; revised September 12, 2016.

Abstract In practice, some bugs have more impact than others and thus deserve more immediate attention. Due to tight schedule and limited human resources, developers may not have enough time to inspect all bugs. Thus, they often concentrate on bugs that are highly impactful. In the literature, high-impact bugs are used to refer to the bugs which appear at unexpected time or locations and bring more unexpected effects (i.e., surprise bugs), or break pre-existing functionalities and destroy the user experience (i.e., breakage bugs). Unfortunately, identifying high-impact bugs from thousands of bug reports in a bug tracking system is not an easy feat. Thus, an automated technique that can identify high-impact bug reports can help developers to be aware of them early, rectify them quickly, and minimize the damages they cause. Considering that only a small proportion of bugs are high-impact bugs, the identification of high-impact bug reports is a difficult task. In this paper, we propose an approach to identify high-impact bug reports by leveraging imbalanced learning strategies. We investigate the effectiveness of various variants, each of which combines one particular imbalanced learning strategy and one particular classification algorithm. In particular, we choose four widely used strategies for dealing with imbalanced data and four state-of-the-art text classification algorithms to conduct experiments on four datasets from four different open source projects. We mainly perform an analytical study on two types of high-impact bugs, i.e., surprise bugs and breakage bugs. The results show that different variants have different performances, and the best performing variants SMOTE (synthetic minority over-sampling technique) + KNN (K-nearest neighbours) for surprise bug identification and RUS (random under-sampling) + NB (naive Bayes) for breakage bug identification outperform the $F1$-scores of the two state-of-the-art approaches by Thung *et al.* and Garcia and Shihab.

Keywords high-impact bug, imbalanced learning, bug report identification

1 Introduction

There have been many defect prediction studies, which aim to help developers to reduce software quality assurance effort[1-5]. Although some of the studies have shown promising performance results in terms of correct defect prediction, their defect prediction models are still not practical enough[6-8]. Actually, traditional defect prediction models identify too much code to review without distinguishing the impact of the defects[9]. Due to tight schedules and limited human resources, developers often do not have enough time to take care of all bugs equally. Anvik *et al.* reported their personal communication with a Mozilla triager who highlights: "Everyday, almost 300 bugs appear that need triaging. This is far too much for only the Mozilla programmers to handle."[10] Given the abundance of bugs and limited resources, developers often need to concentrate on bugs which have high impact.

Regular Paper

A preliminary version of the paper was published in the Proceedings of COMPSAC 2016.

This work is supported by the National Natural Science Foundation of China under Grant Nos. 61602403 and 61402406 and the National Key Technology Research and Development Program of the Ministry of Science and Technology of China under Grant No. 2015BAH17F01.

*Corresponding Author

In recent years, more and more research studies pay close attention to high-impact bugs. Ohira *et al.* created four datasets of high-impact bugs by manually reviewing 4 000 bug reports in four open source projects (Ambari, Camel, Derby and Wicket)[11]. They introduced six kinds of high-impact bugs, i.e., surprise bugs, dormant bugs, blocker bugs, security bugs, performance bugs and breakage bugs. Shihab *et al.* developed a model to predict if a file contains a breakage or surprise bug[9].

In this work, we consider a related but different problem compared with the one considered by Shihab *et al.*[9] Rather than predicting if a file contains a breakage or surprise bug, we identify breakage and surprise bug reports from a collection of bug reports. Since Shihab *et al.*'s approach[9] has very low precision (i.e., 4%∼6%), their approach is not a panacea for dealing with high-impact bugs. Using their proposed approach, it is hard for a developer to fix an unknown high-impact bug given a large list of potentially buggy files with a large number of false positives. This motivates us to consider another direction to tackle the problem with high-impact bugs.

Identifying high-impact bugs early on can help to largely reduce or mitigate the damage caused by these bugs. Unfortunately, considering a large number of bug reports that are received daily by developers, it is often hard for developers to identify those that have high impact. Bug reporters can set the value of the severity field of a bug report to indicate how serious the bug is. Unfortunately, only a minority of bug reporters use this field, and most bug reports have their severity field set to the default value[12]. Moreover, the initial severity fields of many bug reports are wrong and they get corrected later on[13]. Thus, there is a need for an automated technique to help developers identify high-impact bug reports, which is the goal of this work.

Identifying high-impact bug reports is not an easy task. Only a small percentage of bug reports are high-impact ones (for example, in Ohira *et al.*'s dataset[11], only less than 1% of Ambari bug reports are breakage bugs). A bug report dataset is often imbalanced due to the small amount of high-impact bugs in a project. Thus, to identify high-impact bug reports, we leverage a number of imbalanced learning algorithms for high-impact bug prediction. In particular, we investigate four widely used imbalanced learning strategies, i.e., random under-sampling (RUS), random over-sampling (ROS), SMOTE and cost-matrix adjuster (CMA), and four popular classification algorithms, i.e., naive Bayes (NB), naive Bayes multinominal (NBM), support vector machine (SVM) and K-nearest neighbors (KNN), which make totally 16 different combinations (i.e., variants).

We focus on two high-impact bugs, i.e., surprise bugs and breakage bugs, which are first studied by Shihab *et al.*[9] Surprise bugs are bugs which have high impact on developers. These bugs appear in unexpected timing (e.g., in post-release) or locations (e.g., in files that are rarely changed before) and may bring more unexpected effects, catching developers off-guard, and disrupting their already-tight quality assurance schedule and workflow. Breakage bugs are the bugs which have high impact on the customers since these bugs break pre-existing functionality and significantly damage the user experience.

To evaluate our variants of our proposed approach, we use four datasets provided by Ohira *et al.*[11], which contain a total of 2 845 bug reports. To evaluate the performance of different algorithms, we use precision, recall and $F1$-score as evaluation metrics, which are widely used in many software engineering studies[3,14-18]. $F1$-score is a summary measure that combines both precision and recall. A higher $F1$-score means a better performance. The results show that different variants have different performances. We also compare the best performing variants of our approach against two state-of-the-art approaches of Thung *et al.*[19] and Garcia and Shihab[20]. These two approaches were originally proposed to categorize bug reports into bug types and identify blocking bugs respectively, but they can be used to identify surprise and breakage bugs too. We find that the best performing variants of our approach outperform these two approaches too.

This paper extends a preliminary study published as a research paper in a conference[21]. The paper extends the preliminary study in various ways. 1) We add one more type of bugs, i.e., breakage bugs. 2) We add three more classic text classification algorithms, i.e., naive Bayes (NB), support vector machine (SVM) and K-nearest neighbors (KNN). 3) We find the best performing variants of our approach for both surprise bug and breakage bug identification, and we also investigate their effectiveness and stability.

The main contributions of this paper are as follows.

1) We propose a new problem of identifying surprise and breakage bugs. This creates a related but different line of work compared with the prior work by Shihab *et al.* which predicts files that contain high-impact bugs[9].

2) We propose to use imbalanced learning strategies to deal with the problem of identifying surprise and breakage bugs.

3) We conduct an analytical study to investigate the performance of four well-known imbalanced learning strategies built on top of four popular text classification algorithms for high-impact bug prediction.

4) We perform experiments on four software projects. The experimental results show that undersampling is the best imbalanced learning strategy among the four, and naive Bayes multinominal is a better classification algorithm for high-impact bug identification.

The rest of our paper is organized as follows. Section 2 briefly presents high-impact bugs. Section 3 presents the overall framework of our study and elaborates the techniques that we use in our approach. Section 4 describes our experiments and the results. Section 5 presents the threats to validity, and Section 6 discusses miscellaneous points about our work. Section 7 presents related work. Conclusions and future work are presented in Section 8.

2 Overview on High-Impact Bugs

As the name implies, high-impact bugs are bugs that have high impact on developers and users. Based on prior studies, Ohira *et al.* summarized six types of high-impact bugs, i.e., surprise bugs, dormant bugs, blocker bugs, security bugs, performance bugs and breakage bugs[11]. The former three kinds of bugs are in terms of process, while the latter three kinds of bugs are in terms of products.

1) *Surprise Bugs.* Surprise bugs are the bugs which appear in unexpected timings (e.g., in post-release) and locations (e.g., in files that are rarely changed). Shihab *et al.* showed that surprise bugs exist in only 2% of all files[9]. However, surprise bugs may disturb developers' task scheduling greatly.

2) *Dormant Bugs.* Dormant bugs are defined as bugs that "were introduced in one version (e.g., Version 1.1) of a system, yet they are NOT reported until AFTER the next immediate version (i.e., a bug is reported against Version 1.2 or later)"[22].

3) *Blocker Bugs.* Blocker bugs are the bugs that block other bugs from being fixed[20]. Due to this reason, blocker bugs have to be fixed early to not prevent other bugs from getting fixed.

4) *Security Bugs.* Security bugs involve a compromise of the system's confidentiality, integrity, or availability. Thus, they should be fixed as soon as possible.

5) *Performance Bugs.* Performance bugs are bugs that cause significant performance degradation. They may lead to an unresponsive system, low throughput and bad user experience.

6) *Breakage Bugs.* Breakage bugs are bugs that break pre-existing functionality and significantly damage the user experience[9].

Fig.1 and Fig.2 present two examples of high-impact bug reports. In the first bug report (WICKET-5326), the bug report describes a bug appearing in the class $CryptoMapper$. Actually, $CryptoMapper$ is rarely changed and bugs rarely appear in $CryptoMapper$. Therefore, the bug is categorized as a surprise bug since the bug appears in an unexpected location. In the second bug report (AMBARI-3279), "Job Track CPU WIO" dashboard widget has strange behaviour. JobTrackCpu will ignore a fix of an old issue, which significantly damages the user experience. Thus, the bug is categorized as a breakage bug.

Note that it is possible for a bug to be both a breakage bug and a surprise bug. In our work, we predict for one specific type each time. For example, when we identify surprise bugs, we only consider whether a bug is a surprise bug or not, no matter if the bug is also of the other types of high-impact bugs. To predict that a bug is both a breakage bug and a surprise bug, we would use two classifiers to predict that this is the case.

3 Methodology

In this section, we first present our overall framework for high-impact bug identification, and then we describe in detail the individual steps in the overall framework.

3.1 Main Steps

Fig.3 presents the overall framework of our proposed approach. The framework mainly contains two phases: the model building phase and the prediction phase. In the model building phase, we build a classifier (i.e., a statistical model) from a training set of bug reports which have been labeled as surprise (or breakage) bugs or not. In the prediction phase, this classifier would be used to identify if an unlabeled bug report would be a surprise (or breakage) bug or not. We build one classifier for surprise bugs, and another for breakage bugs.

Our framework first extracts a number of features from the training bug reports (step 1). Features are various quantifiable characteristics of the bugs that could potentially distinguish the surprise (or breakage) bugs

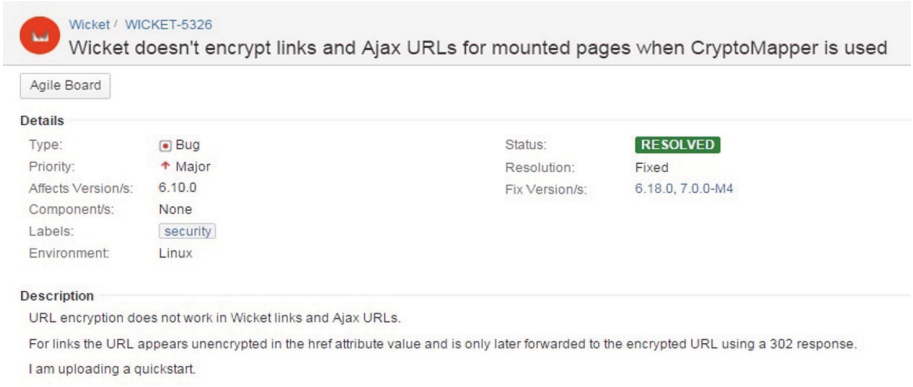


Fig.1. Example of a surprise bug report in Wicket.

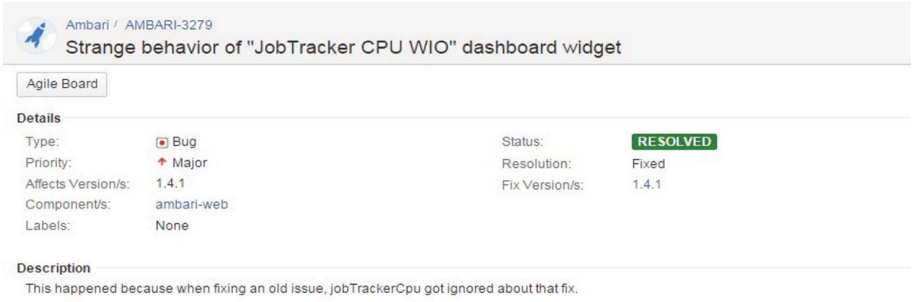


Fig.2. Example of a breakage bug report in Ambari.

from the others. In this paper, we use textual features, which are pre-processed words extracted from the summary and description fields of a bug report (cf. Subsection 3.2). Next, we use some imbalanced learning strategies to handle the class imbalance problem (step 2). We investigate different imbalanced learning strategies (cf. Subsection 3.3) for this step. Finally, we build a classifier based on the extracted features (step 3). We also investigate different classification algorithms (cf. Subsection 3.4) for this step.

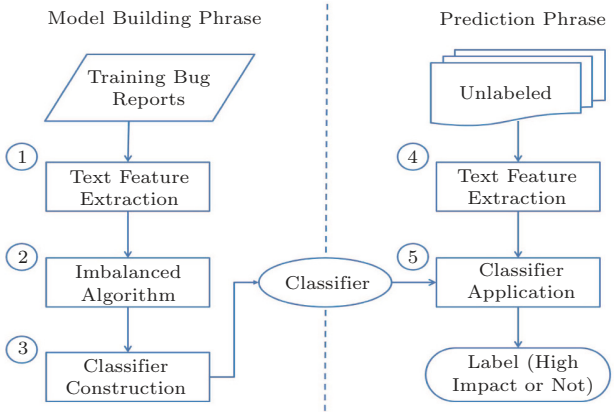


Fig.3. Overall framework of our study.

In the prediction phase, we use the trained classifier to identify whether a bug report with an unknown label is a surprise (or breakage) bug or not. For each of such bug reports, our framework first extracts features from the words in the summary and description fields of the report as we do in the model building phase (step 4). We then input the features to the constructed classifier (step 5). The classifier would output the prediction result which is one of the following labels: surprise (or breakage) bug or not (step 6).

3.2 Feature Extraction

In a bug report, summary and description fields contain most of the useful information for prediction. Therefore, we extract features from these two fields. We first extract all the terms (i.e., words) from the summary and description fields in a bug report. Then, we remove the stop words, numbers and punctuation marks since they provide little information. For the remaining terms, we use Iterated Lovins Stemmer[23] to transform them to their root forms (e.g., "reading" and "reads" are reduced to "read"). We do this stemming step to reduce the feature dimension and to unify similar words into a common representation. Finally, we

calculate the term frequency for each stemmed term. After these steps, a bug report $\boldsymbol{BR}$ is represented as a term frequency vector, i.e., $\boldsymbol{BR} = (t_1, t_2, ..., t_n)$, where t_i denotes the times the i-th term appears in the bug report $\boldsymbol{BR}$. Also, we remove terms which only appear once in one bug report to reduce noise. Table 1 presents the lengths of the term frequency vectors of the four datasets, i.e., Ambari, Camel, Derby, Wicket, after the preprocessing.

Table 1. Lengths of Term Frequency Vectors of the Four Datasets Used in Our Study

Project	Length of Term Frequency Vector
Ambari	1 794
Camel	2 683
Derby	3 585
Wicket	2 658
Average	2 680

3.3 Imbalanced Learning Strategies

Class imbalance is always a big problem in machine learning. It can lead to a classifier having poor performance. Imbalanced learning strategies can be employed to balance the initial imbalanced dataset and help the trained classifier not to be biased to the majority class. Thus, in most cases, it can improve the performance of the classifier[24-25].

There are mainly two classes of imbalanced learning strategies. One is sampling methods, and the other is cost-sensitive methods[26-27]. In our study, we investigate four well-known strategies. Three of them are sampling methods, namely random under-sampling, random over-sampling and SMOTE. The last one is a cost-sensitive method, which adjusts the cost matrix. We refer to it as cost-matrix adjuster. The reason why we select the four strategies is that they are popular and diverse. Random under-sampling (RUS) is an under-sampling method, which shrinks the majority class. Random over-sampling (ROS) and SMOTE are two classic over-sampling methods, which expand the minority class. However, ROS adds duplicated samples to the minority class, while SMOTE adds created artificial samples to the minority class. At last, cost-matrix adjuster (CMA) is a cost-sensitive method.

We introduce the four imbalanced learning strategies briefly below. For simplicity, the subsets of data belonging to the minority class (in our case: high-impact bug reports of a particular category) and the majority class (in our case: all the other bug reports) are denoted by $S_{\min}$ and S_{maj}, respectively.

3.3.1 Random Under-Sampling

Under-sampling is one of the effective sampling methods to deal with the class imbalance problem[26-27]. It deletes data belonging to S_{maj} to shrink its scale. Generally, under-sampling first sets a value p, which is the target ratio of data instances belonging to the majority class to the entire data used for training. Then, it repeats the following two steps until the ratio of S_{maj} to the entire data decreases to p.

Step 1: *Instance Selection.* Select an instance belonging to S_{maj} using a strategy. There are many strategies such as random selection and KNN-based selection.

Step 2: *Instance Deletion.* Delete the instance selected in step 1 from the training dataset.

In our study, we use random under-sampling and set p to 0.5. That is, we randomly delete the data belonging to S_{maj} until the amount of data in S_{maj} is equal to that of $S_{\min}$.

3.3.2 Random Over-Sampling

Over-sampling is another effective approach to deal with the class imbalance problem[26-27]. It duplicates data belonging to $S_{\min}$ to expand its scale. Generally, over-sampling first sets a value p, which is the target ratio of data instances belonging to the minority class to the entire data used for training. Then it repeats the following two steps until the ratio of $S_{\min}$ to the entire data increases to p.

Step 1: *Instance Selection.* Select an instance belonging to $S_{\min}$ using a strategy. There are many strategies such as random selection and cluster-based selection.

Step 2: *Instance Addition.* Add the instance selected in step 1 into the training set.

In our study, we use random over-sampling and set p to 0.5. That is, we randomly duplicate the data belonging to $S_{\min}$ until the scale of $S_{\min}$ is the same as that of S_{maj}.

3.3.3 SMOTE

SMOTE is a more sophisticated over-sampling method, whose full name is Synthetic Minority Oversampling Technique[27-28]. Traditional over-sampling methods duplicate the data belonging to $S_{\min}$, while SMOTE creates some artificial data which can be assumed to belong to $S_{\min}$ based on a specific strategy. Specifically, for each data point x in $S_{\min}$, SMOTE first

finds its K-nearest neighbours (data points) belonging to $S_{\min}$ and links x with each of these k points to form k line segments (in a multidimensional feature space). Then, SMOTE randomly picks a data point on each line segment. The k new data points can be assumed as belonging to the minority class and be added into $S_{\min}$. Therefore, if there are initially n data points in $S_{\min}$, SMOTE will create $k \times n$ artificial data points and add them to $S_{\min}$. By default, k is set to 5.

3.3.4 Cost-Matrix Adjuster

Cost-matrix adjuster is a popular cost-sensitive method to deal with the data imbalance problem[26-27]. Different from the previous three methods, it does not delete or add any data point to S_{maj} or $S_{\min}$. Instead, it changes the cost of misclassifying different training instances belonging to different classes. It makes the cost of misclassifying instances in $S_{\min}$ larger than that of S_{maj} so that the classifier will value $S_{\min}$ more than S_{maj}.

By default, the cost matrix of many classifiers is:

$$\begin{pmatrix} 0 & 1 \\ 1 & 0 \end{pmatrix}.$$

The above cost matrix means that the costs of misclassifying training instances of both classes are the same (i.e., 1), and the costs of correct classification are none (i.e., 0). Cost-matrix adjuster adjusts the cost matrix to achieve better classification performance. For example, when the cost matrix is:

$$\begin{pmatrix} 0 & 1 \\ 2 & 0 \end{pmatrix}.$$

it means that the cost of misclassifying instances of $S_{\min}$ is double compared with that of S_{maj}. In this way, the classifier values the correct classification of instances $S_{\min}$ more than that of S_{maj}.

In our study, given the ratio of the majority and the minority class as $x : y$, we set the cost matrix as follows:

$$\begin{pmatrix} 0 & y \\ x & 0 \end{pmatrix}.$$

3.4 Classification Algorithms

We investigate four popular classification algorithms, i.e., naive Bayes (NB), naive Bayes multinominal (NBM), support vector machine (SVM) and K-nearest neighbors (KNN). All of them are classic algorithms which work well in many text classification tasks[19-20,29]. In addition, the four algorithms are diverse. First, although both NB and NBM are based on Bayes theorem, they represent features in different ways. Second, SVM is a supervised learning model based on structural risk minimization principle. Unlike NB or NBM, SVM is a non-probabilistic binary linear classifier. At last, KNN is a distance-based classification algorithm, which is different from NB and NBM. Also, unlike SVM, KNN does not require a training phase.

3.4.1 Naive Bayes

Naive Bayes (NB) is a probabilistic model based on Bayes theorem for conditional probabilities[26,30]. Naive Bayes assumes that features are independent from one another. Also, all the features are binominal. That is, each feature only has two values of 0 and 1 (in our case, representing whether a word exists in a bug report or not).

Based on the above assumptions, given a bug report $\boldsymbol{BR} = (t_1, t_2, ..., t_n)$ (t_i represents a term in the bug report) and a label c_j (in our case: surprise (or breakage) or not), the probability of $\boldsymbol{BR}$ given the label c_j is:

$$p(\boldsymbol{BR}|C = c_j) = \prod_{i=1}^{n} p(t_i|C = c_j).$$

With Bayes theorem, we can compute the probability of a label c_j given $\boldsymbol{BR}$ as follows:

$$p(C = c_j|\boldsymbol{BR}) = \frac{p(C = c_j) \times \prod_{i=1}^{n} p(t_i|C = c_j)}{p(\boldsymbol{BR})}.$$

Assuming that the probabilities of different labels and the probabilities of different bug reports are uniform, the above equation can be simplified as:

$$p(C = c_j|\boldsymbol{BR}) = \prod_{i=1}^{n} p(t_i|C = c_j).$$

The probability of word t_i given class c_j (i.e., $p(t_i|C = c_j)$) in the above equation can be estimated based on the training data. Next, based on the above equation, we can compute the probability for every label given a new bug report $\boldsymbol{BR}$, and assign the label with the highest probability to it.

3.4.2 Naive Bayes Multinomial

Naive Bayes multinominal (NBM) is very similar to NB[26,30]. However, in NBM the value of each feature is

not restricted to 0 or 1, and it can be any non-negative number (in our case, representing the frequency of a word in a bug report). Since NBM can capture more information, it often outperforms NB.

3.4.3 Support Vector Machine

Given training bug reports, support vector machine (SVM)[26,30] first maps each bug report to a point in a high-dimensional space, in which each feature (in our case: a pre-processed word) represents a dimension. Then, SVM selects the points which have big impact for classification as support vectors. Next, it creates a separating hyperplane as a decision boundary to classify two classes. The separating hyperplane created by SVM has a maximum margin, i.e., it separates the support vectors belonging to the two classes as far as possible. When an unlabeled data instance (in our case: a bug report) needs to be classified, SVM can assign it a label according to the decision boundary.

3.4.4 K-Nearest Neighbors

K-nearest neighbors (KNN) is an instance-based classifier[26,30]. Its principle is intuitive: similar instances have similar class labels. In our setting, KNN mainly contains three steps. First, similar to SVM, KNN maps all the training bug reports to points in a high-dimensional space. Then, for an unlabeled bug report $\boldsymbol{BR}$, we find K nearest points to it based on a specific distance metric. In this paper, we use the Euclidean distance as the metric. Euclidean distance between two points is the length of the line segment connecting them. Finally, we determine the label of $\boldsymbol{BR}$ by the labels of the majority of its K nearest neighbors.

4 Experiments

In our study, the platform is Java, and the classification algorithms we use are built in Weka. We first present our experimental setting and evaluation metrics in Subsection 4.1∼Subsection 4.3. We then present four research questions and our experimental results that answer these questions in Subsection 4.4.

4.1 Datasets

We perform experiments on four datasets from four Apache open source projects, which are Ambari①, Camel②, Derby③, and Wicket④. The projects are selected based on three criteria[11]. First, all the projects have a large number of reported issues, which is essential for a good research in the topic. Second, all the projects use JIRA⑤ as an issue tracking system, which leads to an easier manual labeling process. Third, the projects are different from one another in the application domain, which is essential for a general investigation since the distribution of high-impact bugs can be very different in different application domains.

The four datasets contain a total of 2 845 bug reports. Ohira *et al.* collected and manually categorized them[11]. The labels are generated by four graduate students and four faculty members, in which each pair of graduate student and faculty member are responsible for a single dataset and reach an agreement for the labeling of all the bug reports. In addition, we also manually check the labeling of several samples randomly before performing our experiments and find them to be reasonable.

Table 2 summarizes the statistics of each dataset, containing the total number of bug reports (BRs), the number of surprise BRs, the number of breakage BRs, the corresponding ratios of the two kinds of BRs and the time periods of the BRs. We can see that all the datasets are imbalanced, especially for the breakage class, whose proportions are less than 10% in three out of the four datasets. In Ambari, the ratio of breakage BRs is even less than 1%.

4.2 Experimental Settings

In our study, we investigate the effectiveness of our approach using four imbalanced learning strategies presented in Subsection 3.3, and the four popular classification algorithms for test classification presented in Subsection 3.4. In the experiments, we use the default values of these imbalanced learning strategies and classification algorithms. We first compare the effectiveness of various variants of our proposed approach (using various classification algorithms and imbalanced

① http://ambari.apache.org/, Nov. 2016.
② http://camel.apache.org/, Nov. 2016.
③ http://db.apache.org/derby/, Nov. 2016.
④ https://wicket.apache.org/, Nov. 2016.
⑤ https://www.atlassian.com/software/jira/, Nov. 2016.

Table 2. Statistics of Datasets Used in Our Study

Project	Number of Total BRs	Number of Surprise BRs	Proportion (%)	Number of Breakage BRs	Proportion (%)	Period
Ambari	871	266	30.54	8	0.92	2011.9∼2014.8
Camel	579	228	39.38	47	8.12	2007.7∼2013.9
Derby	731	111	15.18	194	26.54	2004.9∼2014.9
Wicket	663	242	36.50	60	9.05	2006.10∼2014.11

learning strategies). Next, we compare the best performing variants against two baselines, which are two state-of-the-art approaches proposed by Thung *et al.*[19] and Garcia and Shihab[20].

We use stratified 10-fold cross-validation[26] to evaluate the variants of our approach and two baselines. Following stratified 10-fold cross-validation, each dataset is divided into 10 folds, where each fold contains more or less equal proportion of instances belonging to each class. Next, we perform 10 evaluation rounds; in each round, nine folds are used as a training dataset, and the remaining one fold is used as a testing dataset. We aggregate the results of the 10 evaluation rounds and report the overall performance. Stratified cross-validation is a standard evaluation setting, which is widely used in software engineering studies[20,31-34]. Since stratified 10-fold cross-validation involves randomness, to increase the confidence of the results, we repeat it 10 times and report the average results.

We repeat 10-fold cross-validation 10 times for both surprise and breakage bugs. We then report separate results for surprise and breakage bugs.

4.3 Evaluation Metrics

We use precision, recall and $F1$-score as evaluation metrics. These metrics are commonly-used measures to evaluate classification performance[3,26]. They can be derived from a confusion matrix. The confusion matrix lists all four possible classification results. If a bug report is correctly classified as "surprise" (or "breakage"), it is a true positive (TP); if a bug report is misclassified as "surprise" (or "breakage"), it is a false positive (FP). Similarly, there are false negatives (FN) and true negatives (TN). TP, FP, FN, and TN mean the number of TP, FP, FN, and TN respectively. Based on TP, FP, FN, and TN, precision, recall and $F1$-score are calculated.

Precision. Precision is the proportion of correctly predicted "surprise" (or "breakage") bug reports to all bug reports predicted as "surprise" (or "breakage"). Mathematically, precision P is defined as:

$$P = TP/(TP + FP).$$

Recall. Recall is the proportion of the number of correctly predicted "surprise" (or "breakage") bug reports to the actual number of "surprise" (or "breakage") bug reports. Mathematically, recall R is defined as:

$$R = TP/(TP + FN).$$

$F1$-Score. $F1$-score is a summary measure that combines both precision and recall — it evaluates if an increase in precision (recall) outweighs a reduction in recall (precision). Mathematically, $F1$-score F is defined as:

$$F = (2 \times P \times R)/(P + R).$$

4.4 Research Questions

Our experiments are designed to answer the following research questions.

4.4.1 RQ1: Which Variants of Our Proposed Approach Perform the Best for Identifying Surprise and Breakage Bug Reports?

Motivation. When considering four imbalanced learning strategies and four classification algorithms, we can have totally 16 variants (i.e., combinations of one of the imbalanced learning strategies and one of the classification algorithms). Therefore, in the first research question, we want to investigate which variants perform the best for identifying surprise and breakage bug reports.

Approach. To answer this question, we first use the imbalanced learning strategies, which can be any one of the four imbalanced learning strategies (i.e., random under-sampling (RUS), random over-sampling (ROS), SMOTE and cost-matrix adjuster (CMA)), to balance the initial training dataset. Then, we use the classification algorithms, which can be any one of four popular text classification algorithms (i.e., naive Bayes (NB), naive Bayes multinominal (NBM), support vector machine (SVM) and K-nearest neighbors (KNN) method), to build classifiers. We use the three evaluation metrics mentioned above to compare the totally 16 variants.

Results. Table 3 and Table 4 present the $F1$-score values of the top-3 best performing variants for surprise bug and breakage bug identification respectively. The detailed results (i.e., precision, recall and $F1$-score values) are shown in Table 18∼Table 23 in "result.pdf" at "https://github.com/goddding/JCST". From these tables, we can conclude several points.

Table 3. $F1$-Scores of Top-3 Best Performing Variants for Surprise Bug Identification

Project	SMOTE+KNN	CMA+NB	RUS+NBM
Ambari	0.433 2	0.418 9	0.427 0
Camel	0.562 1	0.467 1	0.512 1
Derby	0.266 2	0.378 4	0.245 2
Wicket	0.526 9	0.376 0	0.415 0
Average	0.447 1	0.410 4	0.399 8

Table 4. $F1$-Scores of Top-3 Best Performing Variants for Breakage Bug Identification

Project	RUS+NB	ROS+NB	CMA+NB
Ambari	0.015 1	0.000 0	0.000 0
Camel	0.216 5	0.249 3	0.213 3
Derby	0.636 6	0.640 0	0.641 2
Wicket	0.234 1	0.220 7	0.246 9
Average	0.275 6	0.277 5	0.275 4

First, from Table 3 and Table 4, we can see that SMOTE+KNN achieves the average $F1$-scores of 0.45 for surprise bug identification, and RUS+NB, ROS+NB and CMA+NB achieve the average $F1$-scores of 0.28 for breakage bug identification. They are the best performing variants compared with the other variants.

Second, from Table 18 and Table 19 of the "result.pdf", we can find that for surprise bug identification, most variants achieve the average values of 30%∼35% in terms of precision and the gap between them is relatively small. However, in terms of recall, the different variants have significantly different average values. For example, SMOTE+KNN achieves the highest average recall of 80%, while SMOTE+NB has the lowest average recall of only 20%. Thus, SMOTE+KNN performs the best for surprise bug identification.

Third, from Table 21 and Table 22 of the "result.pdf", we can find that for breakage bug identification, different variants have significantly different average values in terms of both precision and recall. For example, SMOTE+SVM achieves the highest average precision of 32%, but has low average recall of 22%. RUS+NBM achieves the highest average recall of 70%, but has low average precision of 21%. Therefore, both of them cannot be the best performing variants. Among the three best performing variants, i.e., RUS+NB, ROS+NB and CMA+NB, we can see that RUS+NB has much higher average recall than the other two variants (53% vs 33%) and its average precision is only lower than the other two variants a bit (23% vs 25%). Considering the setting of our problem, recall is more important than precision because high-impact bugs should be identified as many as possible. Therefore, RUS+NB shows the best performance for breakage bug identification.

Summary. Different variants have different performances for surprise bug and breakage bug identification. Among them SMOTE+KNN shows the best performance for surprise bug identification, and RUS+NB shows the best performance for breakage bug identification.

4.4.2 RQ2: Can the Best Performing Variants of Our Approach Outperform State-of-the-Art Approaches?

Motivation. Since we have found that SMOTE +KNN performs the best for surprise bug identification and RUS+NB performs the best for breakage bug identification, we want to go further to investigate whether the two best performing variants are effective compared with some state-of-the-art approaches.

Approach. To demonstrate the effectiveness of the two best performing variants, we compare them with two state-of-the-art approaches proposed by Thung *et al.*[19] and Garcia and Shibab[20] respectively. The first baseline is an automatic bug categorization approach based on support vector machine[19]. It is referred to as baseline-1 in the following text. The second baseline is an approach based on decision tree, initially designed for blocking bug characterization and prediction[20]. It is referred to as baseline-2 in the following text. For comparability sake, we use the same features introduced in Subsection 3.2 for all the approaches. We use the evaluation metric $F1$-score to make comparisons.

Results. Table 5 and Table 6 present the $F1$-score values of the two best performing variants compared with those of the two baselines for surprise bug and breakage bug identification, respectively. From the two tables, we can conclude several points.

First, for surprise bug identification, our approach achieves an average $F1$-score of 0.45, while baseline-1 and baseline-2 achieve 0.32 and 0.29 respectively. Compared with the two baselines, our approach improves

the $F1$-scores by 42% and 55% respectively, which is a substantial improvement.

Table 5. $F1$-Scores of the Best Performing Variant and the Two Baselines for Surprise Bug Identification

Project	SMOTE+KNN	Baseline-1	Baseline-2
Ambari	0.433 2	0.355 8	0.277 4
Camel	0.562 1	0.392 1	0.375 8
Derby	0.266 2	0.163 3	0.091 5
Wicket	0.526 9	0.351 7	0.396 7
Average	0.447 1	0.315 7	0.285 4

Table 6. $F1$-Scores of the Best Performing Variant and the Two Baselines for Breakage Bug Identification

Project	RUS+NB	Baseline-1	Baseline-2
Ambari	0.015 1	0.000 0	0.000 0
Camel	0.216 5	0.243 2	0.133 3
Derby	0.636 6	0.570 6	0.579 4
Wicket	0.234 1	0.173 9	0.166 7
Average	0.275 6	0.246 9	0.219 9

Second, in terms of breakage bug identification, our approach achieves an average $F1$-score of 0.28, while baseline-1 and baseline-2 achieve 0.25 and 0.22 respectively. Compared with the two baselines, our approach improves the $F1$-scores by 12% and 25% respectively.

Third, for both surprise bug and breakage bug identification, the two best performing variants outperform the two baselines in all the datasets except for one situation (i.e., baseline-1 has a higher $F1$-score for breakage bug prediction in dataset Camel).

To better demonstrate the superiority of our approach, we perform the Wilcoxon statistical test and compute the p-value. We also compute the Cliff's delta. Wilcoxon statistical test is often used to check if the difference in two means is statistically significant (which corresponds to a p-value of less than 0.05). Cliff's delta is often used to check if the difference in two means is substantial. The range of Cliff's delta is in $[-1, 1]$, where -1 or 1 means all values in one group are smaller or larger than those of the other group, and 0 means the data in the two groups is similar. The mappings between Cliff's delta scores and effectiveness levels are shown in Table 7. By computing the p-value and Cliff's delta, the extent of which our approach improves over the two baselines can be more rigorously assessed.

Table 8 and Table 9 present p-values and Cliff's deltas of the best performing variants compared with the two baselines for surprise and breakage bug identification, respectively. From the four tables, we can see the effectiveness of the best performing variants more clearly. In terms of $F1$-score, the best performing variants improve the performance of the baselines statistically significantly and substantially in all the datasets except only one situation, where RUS+NB is worse than Baseline-1 in the dataset Camel for breakage bug identification.

Table 7. Mappings of Cliff's Delta Values to Effectiveness Levels[35]

Cliff's Delta (δ)	Effectiveness Level
$-1 \leqslant \delta < 0.147$	Negligible
$0.146 \leqslant \delta < 0.33$	Small
$0.33 \leqslant \delta < 0.474$	Medium
$0.474 \leqslant \delta \leqslant 1$	Large

Table 8. p-Value and Cliff's Delta Comparison of SMOTE+KNN and the Two Baselines for Surprise Bug Identification in Terms of $F1$-Score

Project	With Baseline-1		With Baseline-2	
	p-Value	Cliff's Delta	p-Value	Cliff's Delta
Ambari	0.005 889	1 (large)	0.005 889	1 (large)
Camel	0.001 953	1 (large)	0.001 953	1 (large)
Derby	0.001 953	1 (large)	0.001 953	1 (large)
Wicket	0.001 953	1 (large)	0.001 953	1 (large)

Table 9. p-Value and Cliff's Delta Comparison of RUS+NB and the Two Baselines for Breakage Bug Identification in Terms of $F1$-Score

Project	With Baseline-1		With Baseline-2	
	p-Value	Cliff's Delta	p-Value	Cliff's Delta
Ambari	0.001 953	1.0 (large)	0.001 953	1.0 (large)
Camel	0.003 906	−0.8 (negligible)	0.001 953	1.0 (large)
Derby	0.001 953	1.0 (large)	0.001 953	1.0 (large)
Wicket	0.005 859	0.8 (large)	0.003 906	0.8 (large)

Summary. The two best performing variants of our approach are more effective than the two state-of-the-art baselines. They outperform the two baselines in all the datasets for both surprise bug and breakage bug identification except for one situation.

4.4.3 *RQ3: What Is the Effect of Varying the Amount of Training Data for the Best Performing Variants of Our Approach?*

Motivation. For some projects, the amount of training data (i.e., bug reports with known label(s)) can be limited. Since we have found that SMOTE+KNN and RUS+NB have the best performances for surprise bug and breakage bug identification respectively, we want to investigate their stability by varying the amount of training data in this research question.

Approach. For the previous research questions, we perform 10-fold cross-validations, which means that 90% of data are used for training and 10% of data are used for testing. In this research question, we perform 2-fold∼10-fold cross-validations on the datasets. We plot two curves on one chart showing the $F1$-score of surprise bug and breakage bug identification respectively, using 2-fold∼10-fold cross-validations.

Results. Fig.4 presents the $F1$-score of surprise bug (blue dashed line) and breakage bug (red solid line) identification using different k-fold cross-validations. In the figure, the curves are very stable. In terms of $F1$-score, the biggest fluctuation is less than 0.03. For the biggest fluctuation, RUS+NB has the lowest $F1$-score of 0.21 using the 2-fold cross-validation, while it has the highest $F1$-score of 0.24 using the 6-fold cross-validation for breakage bug identification in Wicket. Therefore, we can conclude that the best performing variants of our approach have a good stability and can work well with a wide range of training data.

Summary. The best performing variants of our approach (i.e., SMOTE+KNN for surprise bug identification and RUS+NB for breakage bug identification) are stable and able to work well for the reduced amount of training data.

4.4.4 *RQ4: Does Our Approach Work for High-Impact Bug Report Identification in the Cross-Project Setting?*

Motivation. We have shown that the best performing variants of our approach work well for high-impact bug report identification in the within-project setting. We want to further investigate whether our approach can work for high-impact bug report identification in the cross-project setting.

Approach. To construct the cross-project setting, among the four datasets, we use one as the training dataset and another as the testing dataset. We use the best performing variants of our approach (SMOTE+KNN for surprise bug identification and RUS+NB for breakage bug identification) and record $F1$-score to see whether our approach can work in the cross-project setting.

Results. Table 10 presents the $F1$-scores of surprise bug and breakage bug identification in the cross-project setting. From the table, we can see that in most cases, the $F1$-scores achieved by our approach in the cross-project setting are only a bit worse than those achieved in the within-project setting. In addition, an interesting observation is that in some cases, the $F1$-scores in the cross-project setting are even better than those in the within-project setting. For example, considering the

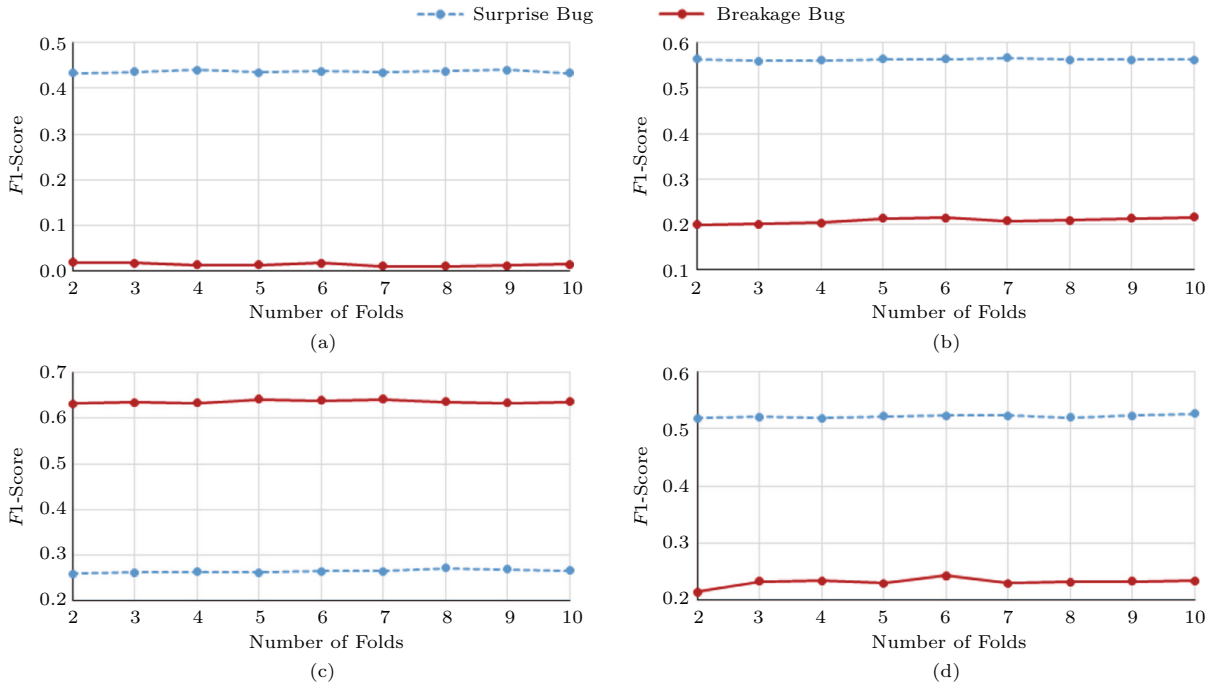


Fig.4. 2-fold∼10-fold validation results for surprise bug and breakage bug identification on four datasets. (a) Ambari. (b) Camel. (c). Derby. (d) Wicket.

performance of our approach in the dataset Ambari, in the within-project setting the $F1$-score is less than 0.02, while in the cross-project setting the $F1$-score is over 0.15. Therefore, we conclude that our approach can work for high-impact bug report identification in the cross-project setting.

Table 10. $F1$-Scores for Surprise Bug and Breakage Bug Identification in the Cross-Project Setting

Project	Surprise	Breakage
Ambari → Camel	0.553 6	0.186 8
Ambari → Derby	0.266 8	0.459 7
Ambari → Wicket	0.495 7	0.197 5
Camel → Ambari	0.460 0	0.031 2
Camel → Derby	0.257 4	0.447 6
Camel → Wicket	0.530 1	0.230 1
Derby → Ambari	0.370 3	0.153 8
Derby → Camel	0.526 9	0.117 6
Derby → Wicket	0.485 2	0.029 0
Wicket → Ambari	0.442 7	0.031 6
Wicket → Camel	0.552 5	0.207 7
Wicket → Derby	0.256 4	0.369 2

Summary. Our approach can work for high-impact bug report identification in the cross-project setting.

4.4.5 *RQ5: Which Words Tend to Discriminate in the Identification of Surprise Bug and Breakage Bug Reports?*

Motivation. Now that we have demonstrated the effectiveness, stability and generalization of our approach for surprise bug and breakage bug identification. We want to further investigate which words tend to discriminate, and which can provide a deeper insight for surprise bug and breakage bug identification.

Approach. We use information gain to find the most discriminant words. Information gain denotes the information that a word can gain to discriminate the surprise/breakage bugs with others. The bigger the information gain value of a word, the more discriminant the word. We first compute the information gain value for each word. And then we sort all the words according to their information gain values. Finally, we choose the top-10 words as the most discriminant words.

Results. Table 11 and Table 12 present the top-10 discriminant words in each dataset for surprise bug and breakage bug identification, respectively. From the tables, we can find that different datasets have different discriminant words. This is the case since the application domain of each of the datasets is different from one another. In addition, the discriminant words are different for different types of high-impact bugs (surprise bugs and breakage bugs), since they have different definitions.

Table 11. Top-10 Discriminant Words for Surprise Bug Identification

Project	Top-10 Words (After Stemming)
Ambari	group, sum, uninst, tick, slider, finish, cleanup, folder, man, multipl
Camel	norm, transport, provider, broker, multicast, fault, occur, part, replac, retrief
Derby	pract, suspect, pack, ignor, crash, unicod, perf, disappear, failur, upload
Wicket	firefox, inform, engl, deriv, upper, startup, breakpoint, rang, patch, map

Table 12. Top-10 Discriminant Words for Breakage Bug Identification

Project	Top-10 Words (After Stemming)
Ambari	thrown, dump, step, ping, suspect, rang, installer, url, cluster, folder
Camel	upgrad, fail, camel, caus, behav, common, impl, support, find, net
Derby	test, failur, junit, sun, protect, run, diff, reflect, org, apach
Wicket	migr, introduc, wick, issu, field, upgrad, debugger, chang, super, render

Summary. The discriminant words are different for different types of high-impact bugs and different datasets.

5 Threats to Validity

Threats to construct validity relate to the suitability of our evaluation metrics. We use precision, recall and $F1$-score which are also used by many past software engineering studies to evaluate the effectiveness of various classification techniques[3,14-18]. Thus, we believe there are few threats to construct validity.

Threats to internal validity relate to several points. The first one is the potential errors in our experiments and the bias due to randomization. To address this threat, we have manually checked the labels of the datasets used in our study, double checked our implementations and repeated all the experiments 10 times to report the average performance. The second one relates to design decisions of our algorithm. In our algorithm, we do not include "component" as a feature since Xia *et al.* showed that the value of the component field is changed in over 30% of all bug reports that they examined, which is a high proportion[13,18]. This means that the value of "component" has a chance of

being faulty in bug reports and using incorrectly assigned component field may have negative impact on the effectiveness of our approach. Third, 10-fold cross-validation may not mimic how our tool would be used in practice. To address this threat, we also use time-based validation to evaluate the effectiveness of our approach, and show that results achieved by time-based validation and 10-fold cross-validations are similar.

Threats to external validity relate to the generalizability of our results. We have evaluated our approach on 2 845 bug reports from four open source projects. We believe the datasets are large enough for an analytical study.

6 Discussion

6.1 Time-Based Validation

In our work, we use 10-fold cross-validation to evaluate the effectiveness of our approach. Ten-fold cross-validation is a popular method that has been used in many past studies[31-32]. However, in practice, when developers use our approach to identify high-impact bugs, they need to train models from historical bug reports. Therefore, we also investigate another validation method named time-based validation. In time-based validation, we sort bug reports based on the time they are submitted. We then use the first half of the data as training data and the last half of the data as testing data. We use the best performing variants of our approach (SMOTE+KNN for surprise bug identification and RUS+NB for breakage bug identification) and record $F1$-scores to see whether our approach can work at least as well in this validation setting as the 10-fold cross-validation setting.

Table 13 presents the $F1$-scores of surprise bug and breakage bug identification using time-based validation. Compared with Table 5 and Table 6, we can see that the average $F1$-scores achieved by our approach using time-based validation are a bit higher than those achieved using 10-fold cross-validation. Specifically, the $F1$-scores achieved using time-based validation are lower than those achieved using 10-fold cross-validation in only three cases (Ambari and Derby for surprise bug identification and Wicket for breakage bug identification). For the other five cases, the $F1$-scores achieved using time-based validation are higher than those achieved using 10-fold cross-validation. Therefore, we can conclude that the results achieved using 10-fold cross-validation do not overestimate the results achieved using time-based validation.

Table 13. $F1$-Scores for Surprise Bug and Breakage Bug Identification Using Time-Based Validation

Project	Surprise	Breakage
Ambari	0.431 5	0.021 3
Camel	0.613 7	0.253 3
Derby	0.193 1	0.661 1
Wicket	0.617 3	0.219 0
Average	0.463 9	0.288 7

6.2 Additional Four Types of High-Impact Bugs

In our work, following Shihab *et al.*[9], we focus on only two types of high-impact bugs. Actually, our algorithm can be applied to the other types of high-impact bugs. We have shown the superiority of our approach in surprise bug and breakage bug identification Subsection 4.4. Now we extend to the other four types of high-impact bugs to better investigate our approach. Note that the experimental setting is the same as the one introduced in Section 4, and we present the detailed results (i.e., precision, recall and $F1$-score values) in Table 24∼Table 35 in "result.pdf" at "https://github.com/goddding/JCST". From the tables, we can see that just as what we achieve in surprise bug and breakage bug identification, each of the other four types of high-impact bugs has a best performing variant (i.e., SMOTE+KNN for dormant bug identification, CMA+NBM for blocker bug identification, CMA+SVM for security bug identification and CMA+NBM for performance bug identification).

We also compare the best performing variants with the two baselines, i.e., baseline-1 and baseline-2. Table 14∼Table 17 present the $F1$-score values of the four best performing variants compared with those of the two baselines for the four types of high-impact bug identification. From the tables, we can see that the best performing variants of our approach achieve better performances than the two baselines for all the four types of high-impact bug identification.

Table 14. $F1$-Scores of the Best Performing Variant and the Two Baselines for Dormant Bug Identification

Project	SMOTE+KNN	Baseline-1	Baseline-2
Ambari	0.000 0	0.000 0	0.000 0
Camel	0.314 1	0.266 1	0.239 5
Derby	0.340 5	0.288 6	0.253 8
Wicket	0.237 4	0.155 8	0.090 9
Average	0.223 0	0.177 6	0.146 1

Table 15. $F1$-Scores of the Best Performing Variant and the Two Baselines for Blocker Bug Identification

Project	CMA+NBM	Baseline-1	Baseline-2
Ambari	0.105 3	0.000 0	0
Camel	0.000 0	0.000 0	0
Derby	0.190 5	0.240 0	0
Wicket	0.372 7	0.000 0	0
Average	0.167 1	0.060 0	0

Table 16. $F1$-Scores of the Best Performing Variant and the Two Baselines for Security Bug Identification

Project	CMA+SVM	Baseline-1	Baseline-2
Ambari	0.142 9	0.139 5	0.000 0
Camel	0.173 9	0.173 9	0.000 0
Derby	0.477 9	0.477 9	0.348 6
Wicket	0.000 0	0.000 0	0.000 0
Average	0.198 7	0.197 8	0.087 2

Table 17. $F1$-Scores of the Best Performing Variant and the Two Baselines for Performance Bug Identification

Project	CMA+NBM	Baseline-1	Baseline-2
Ambari	0.200 0	0.085 1	0.088 9
Camel	0.340 4	0.346 2	0.355 6
Derby	0.439 0	0.425 5	0.073 2
Wicket	0.375 0	0.289 9	0.155 8
Average	0.341 1	0.286 7	0.168 4

6.3 Qualitative Analysis

In Section 4, we notice that there is a huge variation in the performance results from project to project. After a deep look at the results, we find that there is a relation between the degree of dataset imbalance and the performance of our best variants. The more imbalanced the dataset is, the worse the identification result generated using our best variants would be. For example, for surprise bug identification, the most balanced dataset Camel (which has near 40% surprise bug reports) has the highest $F1$-score (0.56) while the most imbalanced dataset Derby (which has only 15% surprise bug reports) has the lowest $F1$-score (0.27). For breakage bug identification, the most balanced dataset Derby (which has near 27% breakage bug reports) has the highest $F1$-score (0.64) while the most imbalanced dataset Derby (which has only less than 1% breakage bug reports) has the lowest $F1$-score (0.02).

Note that our approach has leveraged imbalance learning strategies. Therefore, we think it is legitimate to conclude that although imbalance learning strategies can improve the performance of high-impact bug identification, it can only alleviate the imbalance problem rather than totally solve it.

6.4 Implications

6.4.1 For Practitioners

With our model, many high-impact bug reports can be identified automatically. For project managers, our model can tell them which bugs are likely to have high impact so that they can assign these bugs to developers as soon as possible. For project developers, when they notice the high-impact bugs from a large number of pending bugs with our model, they can focus more on these high-impact bugs. Solving high-impact bugs can better improve project quality.

Our model needs to be trained on historical bug reports. Practitioners can easily collect historical bug reports from issue tracking systems such as JIRA and Bugzilla. Therefore, our model is practical and can be of much benefit.

6.4.2 For Researchers

Our proposed approach pushes the frontier of research in high-impact bug report identification. With imbalance learning strategies, our model achieves a substantial and statistically significant improvement in correctly identifying high-impact bug reports. However, the basic imbalance learning strategies we use have not fully solved the imbalance problem. This highlights an opportunity for further research to extend, customize or invent advanced imbalance learning strategies.

7 Related Work

We classify related work into three parts. The first part is about studies on high-impact bugs. The second part is about studies on bug report management. The last part is about studies that leverage imbalanced learning strategies.

7.1 High-Impact Bugs

The most related studies to ours are the recent studies by Ohira *et al.*[11] and Shihab *et al.*[9] Ohira *et al.* created four datasets of high-impact bugs by manually reviewing 4 000 bug reports in four open source projects (Ambari, Camel, Derby and Wicket)[11]. They introduced six kinds of high-impact bugs, i.e., surprise bugs, dormant bugs, blocker bugs, security bugs, performance bugs and breakage bugs. In addition, they classified them into two types, i.e., process and product. A bug management process in a project will be

impacted by the first three kinds of bugs, while user experience and satisfaction with software products will be affected by the last three kinds of bugs. Shihab *et al.* developed prediction models to identify if a file contains a breakage or surprise bug[9]. In this work, we investigate the usage of text mining and imbalanced learning strategies to identify high-impact bug reports in a collection of bug reports. This is a related but different problem compared with the one considered by Shihab *et al.*[9] Rather than predicting if a file contains a breakage or surprise bug, we identify breakage and surprise bug reports from a collection of bug reports.

Aside from the two studies highlighted above, there are also other studies that are about high-impact bugs[20,36-37]. Zaman *et al.* conducted a case study on the Firefox project to demonstrate the difference between performance and security bugs[36]. Nistor *et al.* studied performance and non-performance bugs from three popular code bases[37]. Garcia and Shihab studied blocking bugs in six open source projects and proposed a model to identify them[20].

In this paper, we propose an approach that can identify bug reports that correspond to surprise and breakage bugs. We evaluate many variants of our approach using four datasets created by Ohira *et al.*[11] We have shown that the best variant of our approach outperforms the state-of-the-art high-impact bug report identification approach by Garcia and Shihab[20].

7.2 Bug Report Management

Aside from studies on high-impact bugs highlighted in Subsection 7.1, there are many other studies that propose ways to improve how bug reports are handled. These studies typically try to automate some existing manual tasks, or to provide additional insights to help developers better resolve bug reports. These studies can be grouped into several categories; in this subsection, we highlight four categories: bug categorization, bug assignment, reopened bug prediction, and severity prediction.

Bug Categorization. A number of studies propose techniques that categorize bug reports[19,38]. Huang *et al.* proposed a novel orthogonal defect classification (ODC) system by integrating experts' experience and domain knowledge[38]. Thung *et al.* proposed a text mining solution that can categorize bugs into various types[19]. They compared six classic classification algorithms and concluded that SVM achieves the best performance for automatic bug categorization. In this paper, we have compared our work against a state-of-the-art study that automatically categorizes bugs, i.e., [19]. Our experiments demonstrate that the best performing variant of our approach which leverages under sampling outperforms that work.

Bug Assignment. There are many studies that propose automated techniques that assign developers to bug reports[29,39-40]. Jeong *et al.* introduced a graph model based on Markov chains, which captures bug tossing history, to improve bug triage[39]. The model reveals developer network and can help better assign developers to bug reports. Anvik and Murphy presented a machine learning approach to create recommenders that assist with a variety of decisions aimed at reducing the effort of bug report triage[29]. Bhattacharya *et al.* employed a comprehensive set of machine learning tools and a probabilistic graph-based model (bug tossing graphs) to assign bug reports to developers[40]. They performed an ablative analysis by unilaterally varying classifiers, features, and learning model to show their relative importance on bug assignment accuracy, and also proposed optimization techniques.

Reopened Bug Report Prediction. There are a number of studies that propose automated approaches that can predict if a bug report would be reopened after it has been closed[33,41-42]. Shihab *et al.* studied reopened bugs on three projects and proposed prediction models based on decision trees[33]. They used sampling methods to handle the imbalanced datasets. Xia *et al.* investigated the performance of different supervised learning algorithms for re-opened bug prediction[41]. They found bagging of decision tree achieves the best performance. In later work, they proposed a novel approach ReopenPredictor which extracts more textual features from the bug reports[42]. The approach automatically estimates thresholds to maximize the prediction performance.

Severity Prediction. There are several studies that predict the severity of bug reports to help developers prioritize bug reports[12,34,43]. Menzies and Marcus proposed a novel automated method called SEVERIS[34]. The method is based on text mining and machine learning techniques and it is applied to predict the severity of bug reports from NASA. Lamkanfi *et al.* investigated whether the severity of a reported bug can be accurately predicted by analyzing its textual description using text mining algorithms[12]. Different from Menzies and Marcus, Lamkanfi *et al.* focused on coarse-grained severity levels (i.e., severe and not-severe) rather than fine-grained ones. In later work, they went further to

compare four well-known text mining algorithms to accurately predict the severity of bug reports[43].

Similar to the above studies, the goal of this work is also to help developers better manage bug reports, which are often too many for developers to deal with[10]. We consider an orthogonal concern compared with the above studies though, namely the identification of high-impact bug reports.

7.3 Imbalanced Learning Strategies

There are a number of software engineering studies which leverage imbalanced learning strategies[24,44-45]. Kamei *et al.* investigated the effectiveness of over- and under-sampling strategies on fault-prone module detection[24]. They evaluated the performance of four sampling methods applied to four fault-prone detection models. They concluded that all the four sampling methods can improve the prediction performance. Wang and Yao used class imbalance learning for software defect prediction[44]. They investigated different types of imbalanced learning strategies and proposed a dynamic version of AdaBoost.NC, which is an ensemble learning method that automatically adjusts its parameters during training. Pelayo and Dick evaluated the effectiveness of SMOTE sampling technique for software defect prediction[45]. Their results show that SMOTE can improve the average performance by at least 23% on four benchmark datasets.

Similar to the above approaches, we also employ imbalanced learning algorithms, while we consider a different problem, namely the identification of high-impact bug reports in a collection of bug reports.

8 Conclusions

In this paper, we leveraged imbalanced learning strategies to identify high-impact bug reports⑥. We investigated four widely used imbalanced learning strategies (i.e., random under-sampling, random over-sampling, SMOTE and cost-matrix adjuster) and four popular text classification algorithms (i.e., naive Bayes, naive Bayes multinominal, support vector machine and K-nearest neighbors) to perform experiments on datasets from four different open source projects. We focused on two types of high-impact bugs, i.e., surprise bugs and breakage bugs, which are first studied by Shihab *et al.*[9] The results showed that different variants have different performances, and the best performing variants outperform the $F1$-scores of the two baseline approaches by Thung *et al.*[19] and Garcia and Shihab[20], respectively. In addition, we investigated the stability of the best performing variants.

In the future, we plan to continue improving the $F1$-score of our proposed approach by introducing additional technical contributions. We also plan to perform experiments on more datasets to reduce the threats to external validity.

References

[1] D'Ambros M, Lanza M, Robbes R. An extensive comparison of bug prediction approaches. In *Proc. the 7th IEEE Working Conference on Mining Software Repositories* (*MSR*), May 2010, pp.31-41.

[2] Rahman F, Devanbu P. How, and why, process metrics are better. In *Proc. the 35th International Conference on Software Engineering*, May 2013, pp.432-441.

[3] Nam J, Pan S J, Kim S. Transfer defect learning. In *Proc. the 35th International Conference on Software Engineering*, May 2013, pp.382-391.

[4] Kamei Y, Shihab E, Adams B, Hassan A E, Mockus A, Sinha A, Ubayashi N. A large-scale empirical study of just-in-time quality assurance. *IEEE Transactions on Software Engineering*, 2013, 39(6): 757-773.

[5] Yang X, Lo D, Xia X, Zhang Y, Sun J. Deep learning for just-in-time defect prediction. In *Proc. the IEEE International Conference on Software Quality, Reliability and Security* (*QRS*), August 2015, pp.17-26.

[6] Hassan A E. The road ahead for mining software repositories. In *Proc. the Frontiers of Software Maintenance*, September 28-October 4, 2008, pp.48-57.

[7] Godfrey M W, Hassan A E, Herbsleb J, Murphy G C, Robillard M, Devanbu P, Mockus A, Perry D E, Notkin D. Future of mining software archives: A roundtable. *IEEE Software*, 2009, 26(1): 67-70.

[8] Shihab E, Jiang Z M, Ibrahim W M, Adams B, Hassan A E. Understanding the impact of code and process metrics on post-release defects: A case study on the Eclipse project. In *Proc. the ACM-IEEE International Symposium on Empirical Software Engineering and Measurement*, September 2010.

[9] Shihab E, Mockus A, Kamei Y, Adams B, Hassan A E. High impact defects: A study of breakage and surprise defects. In *Proc. the 19th ACM SIGSOFT FSE and the 13th ESEC*, September 2011, pp.300-310.

[10] Anvik J, Hiew L, Murphy G C. Coping with an open bug repository. In *Proc. the OOPSLA Workshop on Eclipse Technology eXchange*, October 2005, pp.35-39.

[11] Ohira M, Kashiwa Y, Yamatani Y, Yoshiyuki H, Maeda Y, Limsettho N, Fujino K, Hata H, Ihara A, Matsumoto K. A dataset of high-impact bugs: Manually-classified issue reports. In *Proc. the 12th IEEE Working Conference on Mining Software Repositories* (*MSR*), May 2015, pp.518-521.

⑥The source code and datasets of our work are available at: "https://github.com/goddding/JCST", Nov. 2016.

[12] Lamkanfi A, Demeyer S, Giger E, Goethals B. Predicting the severity of a reported bug. In *Proc. the 7th IEEE Int. Working Conference on Mining Software Repositories* (*MSR*), May 2010, pp.1-10.

[13] Xia X, Lo D, Wen M, Shihab E, Zhou B. An empirical study of bug report field reassignment. In *Proc. the Software Evolution Week — IEEE Conference on Software Maintenance, Reengineering, and Reverse Engineering*, February 2014, pp.174-183.

[14] Kim S, Whitehead E J, Zhang Y. Classifying software changes: Clean or buggy? *IEEE Transactions on Software Engineering*, 2008, 34(2): 181-196.

[15] Rahman F, Posnett D, Devanbu P. Recalling the "imprecision" of cross-project defect prediction. In *Proc. the 20th ACM SIGSOFT International Symposium on the Foundations of Software Engineering*, Nov. 2012, Article No. 61.

[16] Canfora G, De Lucia A, Di Penta M, Oliveto R, Panichella A, Panichella S. Multi-objective cross-project defect prediction. In *Proc. the 6th IEEE International Conference on Software Testing, Verification and Validation* (*ICST*), March 2013, pp.252-261.

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

[18] Xia X, Lo D, Shihab E, Wang X. Automated bug report field reassignment and refinement prediction. *IEEE Transactions on Reliability*, 2016, 65(3): 1094-1113.

[19] Thung F, Lo D, Jiang L. Automatic defect categorization. In *Proc. the 19th Working Conference on Reverse Engineering* (*WCRE*), October 2012, pp.205-214.

[20] Garcia H V, Shihab E. Characterizing and predicting blocking bugs in open source projects. In *Proc. the 11th Working Conference on Mining Software Repositories*, May 31-June 1, 2014, pp.72-81.

[21] Yang X, Lo D, Huang Q, Xia X, Sun J. Automated identification of high-impact bug reports leveraging imbalanced learning strategies. In *Proc. the 40th IEEE Annual Computer Software and Applications Conference* (*COMPSAC*), June 2016, pp.227-232.

[22] Chen T H, Nagappan M, Shihab E, Hassan A E. An empirical study of dormant bugs. In *Proc. the 11th Working Conference on Mining Software Repositories*, May 31-June 1, 2014, pp.82-91.

[23] Lovins J B. Development of a stemming algorithm. *Mechanical Translation and Computational Linguistics*, 1968, 11: 22-31.

[24] Kamei Y, Monden A, Matsumoto S, Kakimoto T, Matsumoto K. The effects of over and under sampling on fault-prone module detection. In *Proc. the 1st International Symposium on Empirical Software Engineering and Measurement*, September 2007, pp.196-204.

[25] Khoshgoftaar T M, Yuan X, Allen E B. Balancing misclassification rates in classification-tree models of software quality. *Empirical Software Engineering*, 2000, 5(4): 313-330.

[26] Han J, Kamber M. Data Mining: Concepts and Techniques. Morgan Kaufmann, 2006.

[27] He H, Garcia E A. Learning from imbalanced data. *IEEE Transactions on Knowledge and Data Engineering*, 2009, 21(9): 1263-1284.

[28] Chawla N V, Bowyer K W, Hall L O, Kegelmeyer W P. SMOTE: Synthetic minority over-sampling technique. *Journal of Artificial Intelligence Research*, 2002, pp.321-357.

[29] Anvik J, Murphy G C. Reducing the effort of bug report triage: Recommenders for development-oriented decisions. *ACM Transactions on Software Engineering and Methodology*, 2011, 20(3): 10.

[30] Aggarwal C. Data Mining: The Textbook. Springer International Publishing, 2015.

[31] Xia X, Feng Y, Lo D, Chen Z, Wang X. Towards more accurate multi-label software behavior learning. In *Proc. the Software Evolution Week — IEEE Conference on Software Maintenance, Reengineering and Reverse Engineering* (*CSMRWCRE*), Feb. 2014, pp.134-143.

[32] Xia X, Lo D,Wang X, Zhou B. Tag recommendation in software information sites. In *Proc. the 10th Working Conference on Mining Software Repositories*, May 2013, pp.287-296.

[33] Shihab E, Ihara A, Kamei Y, Ibrahim W M, Ohira M, Adams B, Hassan A E, Matsumoto K. Studying re-opened bugs in open source software. *Empirical Software Engineering*, 2013, 18(5): 1005-1042.

[34] Menzies T, Marcus A. Automated severity assessment of software defect reports. In *Proc. the IEEE International Conference on Software Maintenance*, May 2008, pp.346-355.

[35] Cliff N. Ordinal Methods for Behavioral Data Analysis. Psychology Press, 2014.

[36] Zaman S, Adams B, Hassan A E. Security versus performance bugs: A case study on Firefox. In *Proc. the 8th Working Conference on Mining Software Repositories*, May 2011, pp.93-102.

[37] Nistor A, Jiang T, Tan L. Discovering, reporting, and fixing performance bugs. In *Proc. the 10th Working Conference on Mining Software Repositories*, May 2013, pp.237-246.

[38] Huang L, Ng V, Persing I, Geng R, Bai X, Tian J. AutoODC: Automated generation of orthogonal defect classifications. In *Proc. the 26th IEEE/ACM International Conference on Automated Software Engineering* (*ASE*), November 2011, pp.412-415.

[39] Jeong G, Kim S, Zimmermann T. Improving bug triage with bug tossing graphs. In *Proc. the 7th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering*, August 2009, pp.111-120.

[40] Bhattacharya P, Neamtiu I, Shelton C R. Automated, highly-accurate, bug assignment using machine learning and tossing graphs. *Journal of Systems and Software*, 2012, 85(10): 2275-2292.

[41] Xia X, Lo D, Wang X, Yang X, Li S, Sun J. A comparative study of supervised learning algorithms for re-opened bug prediction. In *Proc. the 17th European Conference on Software Maintenance and Reengineering* (*CSMR*), March 2013, pp.331-334.

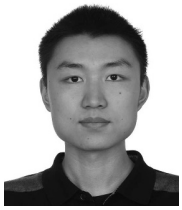
[42] Xia X, Lo D, Shihab E, Wang X, Zhou B. Automatic, high accuracy prediction of reopened bugs. *Automated Software Engineering*, 2014, 22(1): 75-109.

[43] Lamkanfi A, Demeyer S, Soetens Q D, Verdonck T. Comparing mining algorithms for predicting the severity of a reported bug. In *Proc. the 15th European Conference on Software Maintenance and Reengineering* (*CSMR*), March 2011, pp.249-258.
[44] Wang S, Yao X. Using class imbalance learning for software defect prediction. *IEEE Transactions on Reliability*, 2013, 62(2): 434-443.
[45] Pelayo L, Dick S. Applying novel resampling strategies to software defect prediction. In *Proc. the Annual Meeting of the North American Fuzzy Information Processing Society*, June 2007, pp.69-72.

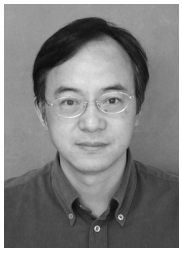
Xin-Li Yang is a Ph.D. candidate in the College of Computer Science and Technology, Zhejiang University, Hangzhou. His research interests include mining software repository and empirical study.

David Lo received his Ph.D. degree in computer science from the School of Computing, National University of Singapore, Singapore, in 2008. He is currently an assistant professor in the School of Information Systems, Singapore Management University, Singapore. He has close to 10 years of experience in software engineering and data mining research and has more than 130 publications in these areas. He received the Lee Foundation Fellow for Research Excellence from the Singapore Management University in 2009. He has won a number of research awards including an ACM Distinguished Paper Award for his work on bug report management. He has published in many top international conferences in software engineering, programming languages, data mining and databases, including ICSE, FSE, ASE, PLDI, KDD, WSDM, TKDE, ICDE, and VLDB. He has also served on the program committees of ICSE, ASE, KDD, VLDB, and many others. He is a steering committee member of the IEEE International Conference on Software Analysis, Evolution, and Reengineering (SANER) which is a merger of the two major conferences in software engineering, namely CSMR and WCRE. He also served as the general chair of ASE 2016. He is a leading researcher in the emerging field of software analytics and has been invited to give keynote speeches and lectures on the topic in many venues, such as the 2010 Workshop on Mining Unstructured Data, the 2013 Génie Logiciel Empirique Workshop, the 2014 International Summer School on Leading Edge Software Engineering, and the 2014 Estonian Summer School in Computer and Systems Science.

Xin Xia received his Ph.D. degree in computer science from the College of Computer Science and Technology, Zhejiang University, Hangzhou, in 2014. He is currently a research assistant professor in the College of Computer Science and Technology at Zhejiang University, Hangzhou. His research interests include software analytic, empirical study, and mining software repository.



Qiao Huang received his B.Eng. and M.Eng. degrees in computer science and technology in the College of Computer Science and Technology, Zhejiang University, Hangzhou, in 2012 and 2016 respectively. He is currently a Ph.D. student in the College of Computer Science and Technology, Zhejiang University, Hangzhou. His research interests include mining software repositories, natural language processing and empirical software engineering.



Jian-Ling Sun received his Ph.D. degree in computer science from the College of Computer Science and Technology, Zhejiang University, Hangzhou, in 1992. He is currently a professor in the College of Computer Science and Technology, Zhejiang University, Hangzhou. His research interests include database and web technology.