Hindawi
Complexity
Volume 2020, Article ID 8509821, 14 pages
https://doi.org/10.1155/2020/8509821

WILEY | Hindawi

Research Article

Surprise Bug Report Prediction Utilizing Optimized Integration with Imbalanced Learning Strategy

Hui Li,[1,2] **Yang Qu**,[1] **Shikai Guo**,[1,2] **Guofeng Gao**,[1] **Rong Chen**,[1] **and Guo Chen**[3]

[1]*Information Science and Technology College, Dalian Maritime University, Dalian 116026, China*
[2]*Collaborative Innovation Center for Transport Studies of Dalian Maritime University, Dalian 116026, China*
[3]*Marine Electrical Engineering College, Dalian Maritime University, Dalian 116026, China*

Correspondence should be addressed to Shikai Guo; shikai.guo@dlmu.edu.cn

Received 30 July 2019; Revised 2 December 2019; Accepted 24 January 2020; Published 19 February 2020

Academic Editor: Saleh Mobayen

In software projects, a large number of bugs are usually reported to bug repositories. Due to the limited budge and work force, the developers often may not have enough time and ability to inspect all the reported bugs, and thus they often focus on inspecting and repairing the highly impacting bugs. Among the high-impact bugs, surprise bugs are reported to be a fatal threat to the software systems, though they only account for a small proportion. Therefore, the identification of surprise bugs becomes an important work in practices. In recent years, some methods have been proposed by the researchers to identify surprise bugs. Unfortunately, the performance of these methods in identifying surprise bugs is still not satisfied for the software projects. The main reason is that surprise bugs only occupy a small percentage of all the bugs, and it is difficult to identify these surprise bugs from the imbalanced distribution. In order to overcome the imbalanced category distribution of the bugs, a method based on machine learning to predict surprise bugs is presented in this paper. This method takes into account the textual features of the bug reports and employs an imbalanced learning strategy to balance the datasets of the bug reports. Then these datasets after balancing are used to train three selected classifiers which are built by three different classification algorithms and predict the datasets with unknown type. In particular, an ensemble method named optimization integration is proposed to generate a unique and best result, according to the results produced by the three classifiers. This ensemble method is able to adjust the ability of the classifier to detect different categories based on the characteristics of different projects and integrate the advantages of three classifiers. The experiments performed on the datasets from 4 software projects show that this method performs better than the previous methods in terms of detecting surprise bugs.

1. Introduction

With the rapidly increasing complexity in the organizations of software projects, utilizing software repositories to test and maintain software systems gains popularity in software engineering domain [1, 2]. The developers need to obtain feedback about defects which exists in the released system through bug reports. However, the number of bug reports is so large that the developers could hardly manage [3]. For example, one of the famous crowdsourced testing platforms, named Baidu Crowdsourced Testing Platform, releases approximately 100 projects per month and receives 1000 test reports per day on average [4]. At the same time, huge amounts of test reports bring the rapidly increasing test tasks for the software projects. It is reported that inspecting 1000 bug reports takes nearly half a working week for a developer on average at present [5]. Due to tight schedules and limited human resources, developers often do not have enough time to inspect all bugs equally. Developers often need to concentrate on bugs which have high impact (have greater threat to the software system), which are used to refer to the bugs which appear at unexpected time or locations and bring more unexpected effects (i.e., surprise bugs). Therefore, it is unrealistic to carefully review all the bug reports and assign them to the suitable developers [6, 7]. Ostrand et al. reported that about 20% of the code files contains 80% of the defects and different defects have different negative impact on software systems [8]. Thus, in order to alleviate the working

pressure of the developers, providing an effective method that assisting developers to detect which bugs have greater threat to the software systems is very essential [9–14]. These bugs with higher priority to be fixed are named high-impact bugs [15].

In recent years, more and more approaches have been proposed to detect high-impact bugs. For further investigating the classification method of detecting high-impact bugs, Ohira et al. manually sorted out high-impact bugs from the datasets of four open-source projects (Ambari, Camel, Derby and Wicket) [16]. These datasets contain six kinds of bugs which are surprise bugs, breakage bugs, dormant bugs, security bugs, blocker bugs, and performance bugs. Shihab et al. proposed a model to identify if a code file contains a breakage or surprise bug [17]. Yang et al. proposed a method to identify high-impact bugs through combinations of classification algorithms and imbalanced learning strategies [18]. Although many classification methods have been able to predict high-impact bugs, the performance of the models are still unsatisfied, and thus, these methods are not suitable to be applied to software projects now [19–23].

Among the high-impact bugs, surprise bugs can bring developers a huge negative impact on the development and maintenance of software systems, though surprise bugs usually appear accidentally in timings and locations and only account for 2% of all the bugs [17]. Aiming to assist developers to detect bugs that are more threat to software systems, we focus on the surprise bugs and propose a classification method through Optimized Integration with Imbalanced Learning Strategy (OIILS) to identify surprise bugs in this paper. As we know, the users usually utilize short or long text descriptions to illustrate the bugs encountered in the software system. Thus, we extract the textual features from the textual descriptions available in the bug reports as training data or testing data. In addition, it is reported that the surprise bugs only account for a small percentage of all the bugs, and thus, the impact on software systems represents imbalance distribution in bug reports datasets [8]. Obviously, imbalanced category distribution has a negative impact on the performance of a classifier [24, 25]. In our method, we adopt imbalanced learning strategy named SMOTE to balance the training data and feed the processed training data to the classifiers. We also pay attention to the gap between the prediction performances of different classifiers for the same dataset. Thus, we utilize different classification algorithms to predict surprise bug reports.

The prediction results show that the classifiers represent various prediction abilities to discover different categories (for instance, if a classifier has a strong ability to detect one category in binary classification, the ability to detect another category is usually weak), and the different classifiers represent various prediction abilities to the same datasets. Therefore, the ensemble method of integrating the advantage of different classifiers and balancing the ability of each classifier becomes a good solution for imbalance data. In other words, one can assign higher weight to improve the weak ability of the classifier to detect one category and assign lower weight to reduce the stronger ability in order to get better classification results. Yang et al. choose 4 widely used strategies (SMOTE, Random Undersampling (RUS), Random Oversampling (ROS), and Cost-Matrix Adjuster (CMA),) for dealing with imbalanced data and 4 text classification algorithms (Naive Bayes (NB), Naive Bayes Multinominal (NBM), Support Vector Machine (SVM), and K-nearest Neighbors (KNN)) to identify high-impact bug reports [18]. In the method of OIILS, we evaluate 7 strategies (NB, J48, KNN, Random Tree (RT), Random Forest (RF), NBM, and SVM) to build the classifiers and then choose three classifiers (KNN, J48, and NBM) based on the experimental results and balance the ability of each classifier. Specifically, we first utilize training data to feed the objective classifier and predict hypothetically unlabeled bugs in training data in the weight training phase. We obtain the probability of each category for each bug and utilize weights to adjust them, respectively. To obtain a higher accuracy, we treat the process of detecting the most suitable weights as a linear programming problem, and the constraint solver named CPLEX is used to address this problem [27]. In the weight adjustment phase, we adjust the probability of each category through the weights obtained in the above step and obtain an optimized result for each classifier. Finally, we infer an ultimate result through the results produced by three classifiers, based on the principle of the minimum.

We investigate the validity of OIILS experimentally on four datasets provided. These datasets are, respectively, from 4 open source projects and contains 2844 bugs. The details of these datasets and the related category distributions are shown in Table 1, including the number of surprise bugs (surprise), the number of ordinary bugs (Ordinary), the totally number of the bugs (Total), and the high-impact bugs as a percentage of all the bugs (Percentage). In addition, it is revealed that the surprise bugs rarely appear in all bugs and category distribution in different projects are unique. The experimental results show the classification algorithms utilized in our method achieve better prediction performance. Meanwhile, the comparison experiments with the possible combinations of all classification algorithms and imbalanced learning strategies show that the combination of algorithms and imbalanced learning strategy presented in this work achieves the best performance. Finally, it is proved that the ensemble method we propose outperforms other classic ensemble methods and the average improvement of F-Measure is between 6.29% and 26.6%.

The contributions of this work are as follows:

(i) We combine imbalanced learning strategy with multiclassification algorithms to overcome imbalanced problem of datasets and take advantage of different abilities of three classification algorithms.

(ii) We propose an ensemble method named Optimized Integration with imbalanced learning strategy that can balance the ability of detecting different categories for each classifier based on the characteristics of the experimental data and integrate advantages of different classification algorithms. The ensemble method considers the optimization weight problem as a linear programming problem and utilizes the

Table 1: The distribution of categories in different projects.

Projects	Surprise	Ordinary	Total	Percentage
Ambari	266	605	871	30.54
Camel	228	351	579	39.38
Derby	111	620	731	15.18
Wicket	242	421	663	36.50

constraint solver named CPLEX to obtain the most suitable weights for higher accuracy of the results.

(iii) We evaluate our method based on 4 open-source projects which contain 2844 bugs. And the results of experiments show that each part of our method is feasible and outperforms other corresponding classification algorithms.

The rest of this paper is constructed as follows. Background is presented in Section 2. Section 3 describes the method of optimized multiclassifier integration. In Section 4, we illustrate the experimental datasets, evaluation metrics, 4 research questions, and corresponding experimental settings. Section 5 presents the results of each research question and analysis based on the experimental results. Finally, conclusion and threats to validity are illustrated in Section 6 and Section 7, respectively.

2. Background

2.1. Related Work. In order to ensure the quality of software systems, the software companies invest a large number of work force and capital costs for software testing and debugging. Thus, many software projects, including open-source projects and commercial software projects, utilize software bug tracking systems in order to conveniently and efficiently manage the bug reports [28]. As software testing progresses, a great number of bug reports are submitted daily to the bug tracking system. For example, during the ten-year period from October 2001 to October 2010, Bugzilla received a total of 333371 bug reports on Eclipse projects, which were submitted by 34917 testers participating in the project. Meanwhile, in Mozilla project, a total of 643615 bug reports were submitted as of December 31, 2011. Due to the large number of bug reports submitted everyday, the task of inspecting bug reports has become more and more time-consuming. Thus the researchers put forward many methods to reduce the lengthy process of reporting in the past ten years. We illustrated these methods mainly as follows.

2.1.1. Content Optimization. Optimization is a widely used method to solve quantitative problems in many disciplines [29–31].

Beetenburg et al. studied the contents of test reports, collected the contents of test reports submitted by people with different roles and abilities in different projects or platforms, and then reported that there are some differences between these bug reports [32]. Thus, the researchers established a supervised learning model based on the content of the developer submitting the bug report, aiming to obtain the information from the bug reports, provide suggestions to the developers on the content of the bug reports, and improve the quality of the test reports. Demeyer and Lankanfi noticed that the specific bug reports fields may contain some errors, such as wrong components [33]. Thus, they utilized data mining techniques to predict possible errors in test reports. Wu et al. reported that the bug report information is usually incomplete; thus, they proposed a method named BUGMINER to determine the key information in bug reports and utilize it to check whether the information of newly submitted bug report is complete [34]. Though many researchers propose many methods to optimize the content of bug reports, the applicability of these methods still unsatisfied, and overcoming limitation of methods is one of the challenges in the future work.

2.1.2. Severity Prediction. Severity is the estimate to the importance of the bugs by its own feedback [22]. Some systems demand high security and fault tolerant, so it is necessary to accurately assess the severity of the defects (bugs) possibly lying in the systems [35, 36]. Aiming to achieve this goal, Menzies and Marcus proposed and applied an automated predictive approach using text-mining and machine learning for mission-critical systems [37]. According to the severity labels utilized in NASA, they divided all bugs into five levels. They first preprocess words extracted from the descriptions of bug reports, such as removing stop words and remaining stemming. Then they select Top-k words by information gain and treated them as features to represent bug reports. At last, they use RIPPER rule learner to classify bug reports with unlabeled severity. Lamkanfi et al. used four classification algorithms for comparing the severity of prediction [38]. Tian et al. summarized the work of their predecessors and proposed an information retrieval method which uses text and nontextual character information to predict the severity of bug reports [39]. In the process of this method, the severity is assigned by submitters subjectively. Each submitter considers the severity of the bug reports by different experience and understanding, but this inevitably brings the subjective and inaccurate evaluation to the severity of the bug reports. Thus a reasonable specification is required to improve accuracy that submitters assign severity.

2.1.3. Priority Prediction. Evaluating the priority of the bug reports is a very important work in software testing. Yu et al. utilized neural networks to speed up the training process in order to reduce the error rate during the evaluation [40]. Kanwal and Mapbool conducted prioritization of bug reports based on SVM and Naive–Bayes classification. Meanwhile, they reported that SVM performs better in predicting the priority of bug reports by utilizing textual features [41]. Tian et al. presented a method of automatically selecting the appropriate classification algorithm to predict the priority of the bug report, based on machine learning framework and classification standard [42]. Gao et al. presented an integration method to identify high-priority bug reports [43].

Besides the methods above, some methods ware presented to alleviate the heavy work of bug report processing, such as detecting duplicate bug reports [44–48], working on misclassification about bug reports [49, 50], and bug report assignment [51, 52]. However, previous studies only use single classifier for prediction, but ignore the differences in the ability of each classifier to detect various projects and categories. In this work, considering the various performance of classifiers and unique distribution of category in projects, we propose a method to make use of the complementary characteristics between classifiers. This method is also optimized according to the data distribution of different projects to achieve the best effect.

2.2. Motivation. Ohira et al. collected the datasets of high-impact bugs by reviewing 4000 bug reports [16]. They categorize datasets into six different kinds of bugs, which are surprise bugs, dormant bugs, blocker bugs, security bugs, performance bugs, and breakage bugs [15]. In this work, we pay attention to surprise bugs.

The surprise bugs we investigate are collected from four Apache open-source projects, which are Ambari, Camel, Derby, and Wicket. The four datasets contain 2844 bugs totally. As shown in Table 1, it can be seen that surprise bugs account for only about one-third or less of the entire datasets. Obviously, imbalanced data distribution in datasets has a negative impact on the classification performance of the classifier [15]. In order to solve this problem, the ensemble method OIILS is designed.

3. Methods

In this section, the method of OIILS will be described. First, we demonstrate the overall framework of OIILS. Then, we divide our method into four submodules, including text feature extraction module, data balancing module, multiclassifier module, and optimization integration module.

In feature extraction module, we extract text futures which can describe the information of the bugs in order to train the machine learning classifiers. In data balancing module, we adopt imbalanced learning strategy to balance the imbalanced datasets, aiming to help classification algorithms promote the prediction results. In multiclassifier module, we use various classification algorithms to build classifiers, respectively, and train the datasets which have been balanced in data balancing module, then utilize these classifiers to produce prediction results. In optimization integration module, we present the ensemble method of OIILS to predict the impact of each bug report based on the results generated by multiclassifier module. In other words, we double the number of training data belonging to minority class.

3.1. Overall Framework. The overview of OIILS is illustrated in Figure 1. OIILS runs in two phases: training phase and prediction phase.

In the training phase, we collect a number of bug reports with known type as input. And then, we extract the text features that can represent the information of each bug report. Second, we utilize imbalanced learning strategy to balance the training data. Finally, we employ three classification algorithms to build classifiers and utilize them to train the training data balanced in data balancing module, respectively.

In the prediction phase, OIILS obtains some bug reports with unknown type. We utilize text feature extraction module to extract the text features of the unlabeled reports. Then, different classifiers that we trained in the training phase are used to predict these bug reports, respectively, and we separately obtain the prediction results of these classifiers. At last, we adopt the ensemble method to produce the final prediction of each bug reports.

Then the details of each module will be demonstrated in the following four subsections.

3.2. Text Feature Extraction Module. The target of text feature extraction module is to collect the text features that could characterize each bug report. First, we extract text descriptions from summary and description fields of bug reports since these two fields could provide useful information about bugs. Then, we utilize word segmentation to segment descriptions into words. In order to reduce noisy data, we remove stop words, numbers, and punctuation marks that contain little meaning. It is reported that developers generally utilize semantically related words that consist the same root word [42]. Thus, we finally apply Iterated Lovins Stemmer [53] to collapse various forms of the same word to their stem for harmonizing words with similar meanings.

After the steps illustrated above, the number of words contained in each bug report is obviously less than before. We treat each stemmed word that appears in bug reports as a textual feature and transform these words to textual feature vector. For bug report set BR, we express the i^{th} bug report br_i by the following formula:

$$\text{br}_i = (t_1, t_2, t_3, \ldots, t_n), \tag{1}$$

where t_j denotes the existence of the j^{th} word. If the j^{th} word exists, the value of t_j is 1; otherwise, 0. And n denotes the number of textual features.

3.3. Data Balancing Module. Imbalanced distribution of category in datasets is an important problem for machine learning. In generally, imbalanced data causes poor performance in almost all classification algorithms. Thus, many investigators adopt imbalanced learning strategies to process the imbalanced dataset in order to assist classifiers in avoiding to bias the majority class when training dataset [54, 55]. In addition, some previous studies have proved that the classification results achieve better after utilizing imbalanced learning strategy to preprocess the datasets in most cases [56, 57].

Imbalanced learning strategies mainly contain two types, which are sampling methods and cost-sensitive methods. In OIILS, we adopt a classic oversampling method named

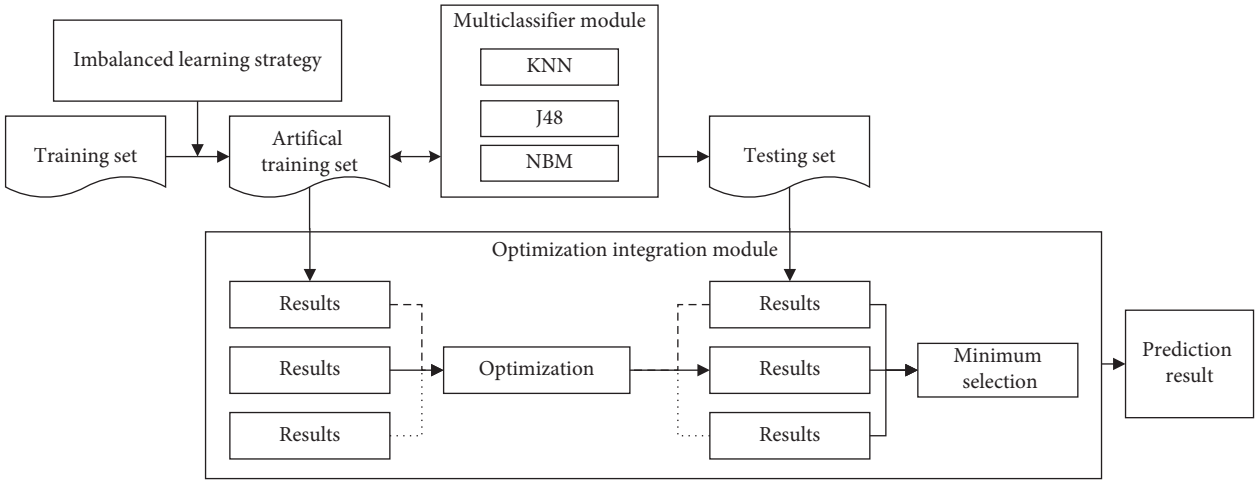

Figure 1: Overview of OIILS.

SMOTE (short for Synthetic Minority Oversampling Technique). This method is more sophisticated than traditional oversampling methods because it could create some artificial data which belongs to minority class based on specific strategy [58]. Firstly, SMOTE identifies its K most similar neighbors for each bug belonging to minority class according to the value of textual feature vector. Next, SMOTE links this bug and its K neighbors in the multidimensional feature space and selects a point randomly in these line segments, respectively. These K artificial points are assumed to be the new data which belongs to minority category. Thus, SMOTE will produce $K * n$ artificial points to assist classifiers to train the features of minority class, if a dataset contains n bugs belonging to minority class. In our method, we let $K = 1$.

3.4. Multiclassifier Module. As we know, different classification algorithms perform different prediction abilities on different datasets. And the effect of utilizing different classification algorithms to predict the same dataset is also different. In addition, there is a certain randomness in the new artificial data constructed by the imbalanced learning strategy named SMOTE. In order to improve the classification stability, we integrate different classification algorithms. In training phase, we employ these algorithms to build classifiers, respectively. And then we use them to train the datasets which are balanced by SMOTE in data balancing module. In the testing phase, we utilize these classifiers to predict bug reports with unknown type and obtain the probability of each category.

3.5. Optimization Integration Module. In the above step, we have introduced that OIILS contains various classification algorithms and gain different prediction results. In order to integrate the advantages of each classification algorithm effectively, we propose an ensemble method named Optimization Integration. Optimization Integration consists of three phases which are the weight training phase, weight adjustment phase, and minimum selection phase.

The input of Algorithm 1 is the training data, the probability of majority class and minority class for each bug, and the constraint solving model. The output of this algorithm is the category of each bug.

Lines 3–20 is the weight training phase. In this phase, the most suitable weights are obtained by constraint solving. In lines 21–25, we use the weights to adjust the original probability on majority class and minority class, respectively, of each bug. Finally, we choose the maximize one among different probabilities about majority class and minority class as the class probability of each bug and use this to determine the category of each bug.

3.5.1. Weight Training Phase. Due to the different abilities of detecting minority category for different classification algorithms, the constraint solver named CPLEX is used to adjust the weight of the abilities for different classification algorithms to identify the priority of the bug report and then improve the prediction accuracy of each classifier [59].

It is well-known that the classifier determines the category of the target by comparing the probabilities that the target belongs to different categories. The objective function that we establish is to determine the weights W_0, W_1 that can balance the ability of detecting two categories in a classifier so as to correspond the prediction results of bug reports to the probability distribution above as much as possible. We utilize the data after balancing in the data balancing module as training data. Firstly, we extract the features of the training data and fed these features to the classifier. Then we convert the category of br_i in training data to the corresponding value C_i according to the following equation:

$$C_i = \begin{cases} -1, & \text{if } \mathrm{BR}_i \text{ belongs to minority class;} \\ 1, & \text{if } \mathrm{BR}_i \text{ belongs to majority class.} \end{cases} \quad (2)$$

As shown in equation (2), if a bug belongs to minority class, the number of this category corresponds to −1; otherwise, 1. Then, we assume the bugs in training data are unlabeled and utilize the classifier to predict them. Given p_i^0, p_i^1 are defined as the probability of two categories which

Require: Training data (the data after balancing), the probability P_i^0 and P_i^1 of each bug, the constraint solving model ILP = OBJ, CONSTRAINTS.
Ensure: The category of each bug.
(1) $C_i = 0$; //Initialization
(2) $W_0 = 0.5$, $W_1 = 0.5$; //Initializes the weights.
(3) /∗Weight training phase.∗/
(4) **for** each BR_i
(5) **if** BR_i belongs to minority class
(6) $C_i = -1$;
(7) **else**
(8) $C_i = 1$;
(9) **end if**
(10) **end for**
(11) OBJ ⟵ maximize $\sum_{i=1}^{n} \mathrm{SubOBJ}_i$
(12) /∗Let the objective function be maximizing the highest achievable accuracy of the classifier.∗/
(13) /∗Generate all the CONSTRAINTS.∗/
(14) CONSTRAINTS ⟵ $W_0 + W_1 = 1$;
(15) CONSTRAINTS ⟵ $0 < W_0 < 1$;
(16) CONSTRAINTS ⟵ $0 < W_1 < 1$;
(17) /∗The ILP formulation is built successfully.∗/
(18) **for** each classifier
(19) W_0, W_1 ⟵ ILP; // Obtain the most suitable weights by optimization.
(20) **end for**
(21) /∗Weight adjustment phase.∗/
(22) **for** each BR_i**do**
(23) $W_0 \times P_i^0$; // P_i^0 is the original majority probability of BR_i.
(24) $W_1 \times P_i^1$; // P_i^1 is the original majority probability of BR_i.
(25) **end for**
(26) /∗Minimum selection∗/
(27) **for** each BR_i**do**
(28) $P_0^{\min}$ ⟵ minimum value of majority class probabilities;
(29) $P_1^{\min}$ ⟵ minimum value of majority class probabilities;
(30) **if** $P_0^{\min} > P_1^{\min}$
(31) Category of BR_i ⟵ majority class;
(32) **else**
(33) Category of BR_i ⟵ minority class;
(34) **end if**
(35) **end for**
(36) **return** the category of each bug report

Algorithm 1: OIILS algorithm.

produced by the classifier predicting the i^{th} bug report, where p_i^0 is the probability of bug report br_i belonging to majority class and p_i^1 denotes the probability of minority class. Finally, we establish the subobjective function as

$$\mathrm{SubOBJ}_i = \frac{(W_0 * p_i^0 - W_1 * p_i^1) * C_i}{\left|W_0 * p_i^0 - W_1 * p_i^1\right|}. \tag{3}$$

It indicates that the subobjective function only contains two results. If the prediction result of the i^{th} bug after adjusting is true, the value of SubOBJ_i is 1; otherwise, the value of SubOBJ_i is −1. Thus, we demonstrate the objective function in the following equation:

$$\mathrm{OBJ} = \sum_{i=1}^{n} \mathrm{SubOBJ}_i. \tag{4}$$

After establishing the objective function in the linear programming problem, we need to obtain the maximum value of the weights in order to harvest the highest achievable accuracy of the classifier. To address the problem, CPLEX is used to obtain the most suitable weights. In addition, the value of weight should be within a reasonable range, and the weights we set is to balance the ability of a classifier essentially. In other words, we need to enhance the ability to detect a category and simultaneously reduce the ability to detect the other category for a classifier. Thus, some constraints are added for constraining the weights. The constraints are given as follows:

$$\begin{aligned} W_0 + W_1 &= 1, \\ 0 < W_0 &< 1, \\ 0 < W_1 &< 1. \end{aligned} \tag{5}$$

3.5.2. Weight Adjustment Phase. After training the weights, we obtain the most suitable weights W_0 and W_1 for each classifier. Then, the weights are used to adjust the prediction

results generated by corresponding classifiers. We utilize corresponding W_0 to adjust the probability p_i^0 of the majority class and utilize corresponding W_1 to adjust the probability p_i^1 of the minority class for the i^{th} bug. Table 2 presents the details of adjustment.

3.5.3. Minimum Selection Phase. After the adjustment, we obtain three sets of predicted results and each set contains the probabilities of two categories. Thus, there are some different probabilities about majority class and different probabilities about minority class for a bug. We set $p_0^{\min}$ to present the minimum value among all of the probabilities about majority class and set $p_1^{\min}$ to present the minimum value of minority class probabilities. Thus, each bug contains $p_0^{\min}$ and $p_1^{\min}$ about majority class and minority class. At last, OIILS utilizes the category represented by the maximum value of $p_0^{\min}$ and $p_1^{\min}$ to determine the final type of this bug report.

4. Experiment Design

In this section, we evaluate OIILS through a serial of experiments by four open-source projects, and the experimental datasets are introduced in Section 2.2. We use stratified sampling to split surprise bugs of each projects into five segments, and 4 of them are used as training dataset randomly, while the remaining one is used as the testing dataset. The evaluation metrics are shown in Section 4.1. Then, we design four research questions to investigate the performance of OIILS in Section 4.2. Finally, we describe the experiment design to figure out these questions in the last subsection.

4.1. Evaluation Metrics. In order to measure the effectiveness of surprise bug classification methods, we utilize three metrics: *precision*, *recall*, and *F-Measure* [60]. The three metrics are widely used for evaluating classification algorithms.

Table 3 shows four results of classification. True positive (TP) denotes a bug is predicted correctly as a surprise bug. False positive (FP) denotes a bug is predicted mistakenly as a surprise bug. The definitions of true negative (TN) and false negative (FN) are similar with true positive and false positive. TP, FP, TN, and FN in mathematical formulas below express the total number of each classification result, respectively.

Precision signifies the proportion of bugs observed as surprise bugs that are predicted correctly. *Precision* can be expressed as

$$\text{Precision} = \frac{\text{TP}}{\text{TP} + \text{FP}}. \tag{6}$$

Recall signifies the proportion of bugs predicted as surprise bugs to all bugs observed as surprise bugs. *Recall* can be expressed as

$$\text{Recall} = \frac{\text{TP}}{\text{TP} + \text{FN}}. \tag{7}$$

Table 2: The rule of utilizing weights to adjust prediction result.

	Majority class	Minority class
Weight	W_0	W_1
Original probability	p_i^0	p_i^1
Adjusted probability	$W_0 * p_i^0$	$W_1 * p_i^1$

Table 3: The definitions of precision, recall, and F-Measure for surprise bugs.

	Predicted surprise bug	Predicted ordinary bug
Observed surprise bug	TP	FN
Observed ordinary bug	FP	TN

F-Measure is the harmonic mean of precision and recall. It is used to balance the discrepancy between *precision* and *recall*. *F-Measure* can be expressed as

$$F - \text{Measure} = \frac{2 * \text{Precision} * \text{Recall}}{\text{Precision} + \text{Recall}}. \tag{8}$$

4.2. Research Questions. In the experiment, we evaluate OIILS by addressing the following research questions.

RQ1: Which classification algorithm works well in classifying surprise bugs?

The answer to RQ1 helps us evaluate whether classification algorithms are appropriate for identifying surprise bugs in all surprise bugs and explains one of the reasons why we select J48, KNN, and NBM in OIILS.

RQ2: Can the combination of classification algorithms of OIILS perform better than other combinations of classification algorithms?

The answer to RQ2 helps us assess the performance of each combination of different classification algorithms and shows that the combination of classification algorithms in OIILS performs best.

RQ3: Can OIILS outperform classification algorithm with imbalanced learning strategy?

The answer to RQ3 helps us determine whether OIILS outperform different combinations of classification algorithms and imbalanced the learning strategy.

RQ4: How accurate is the ensemble method of OIILS compared with the classic ensemble methods, including Adaboost, Bagging, and Vote?

The answer to RQ4 helps us determine whether OIILS performs better than classic ensemble methods, such as Adaboost, Bagging, and Vote. The result can demonstrate that OIILS is able to integrate the advantages of each classification algorithm.

4.3. Experiment Setting

4.3.1. RQ1 Experiment Setting. We first employ NB, J48, KNN, RT, RF, NBM, and SVM to build the classifiers and feed the training datasets generated to them, respectively.

Then, we use different classifiers to classify the testing datasets which are randomly generated. Finally, we evaluate each classification algorithm by three metrics which are *precision*, *recall*, and *F-Measure*, and select some appropriate classification algorithms for integrating.

4.3.2. RQ2 Experiment Setting. We select 5 classification algorithms based on the result of RQ1, which are NB, J48, KNN, RT, and NBM. Then, we randomly select three classification algorithms in five classification algorithms mentioned above to combine. All of the combinations are shown in Table 4. We utilize these 10 groups to replace the combination of algorithms in OIILS, respectively, and predict the generated testing datasets. Finally, we evaluate each group of algorithms through three metrics mentioned above and determine whether the combination of classification in OIILS represents the best performance.

4.3.3. RQ3 Experiment Setting. To compare OIILS with different combinations of imbalanced learning strategy and classification algorithm, we concentrate on four imbalanced learning strategies which are random undersampling (RUS), random oversampling (ROS), cost-matrix adjuster (CMA) and SMOTE [55, 61]. Then we utilize the imbalanced learning strategies mentioned above to balance the generated training datasets. Next, we select the five classification algorithms which are NB, J48, KNN, RT, and NBM to build classifiers. We utilize five classifiers to train the four types of balanced training datasets, respectively, and predict the generated testing datasets. Finally, we evaluate 20 prediction results by *precision*, *recall*, and *F-Measure* and investigate whether OIILS outperforms the methods that simply use the combination of imbalanced learning strategies and classification algorithms.

4.3.4. RQ4 Experiment Setting. We compare OIILS with some classic ensemble methods [62–65]. First, we use SMOTE to balance the generated training datasets and utilize KNN as the basic classification algorithm, because the combination of SMOTE and KNN achieves the best performance among all the combinations. Then, we use two ensemble methods which are Adaboost and Bagging in Weka to integrate the basic classification algorithm, respectively, and predict the testing datasets generated randomly. We also use three classification algorithms which are the same as classification algorithms in OIILS to predict the testing dataset, respectively, and produce a final prediction by Vote. We compare these results and the results generated by OIILS through three metrics mentioned above and determine whether OIILS can make more use of the advantages of each classification algorithm than other ensemble methods.

5. Results and Discussions

In this section, we analyze the four experimental results to demonstrate the answers of four research questions.

Table 4: Combinations of classification algorithms.

Group	Algorithms	Group	Algorithms
G1	NB, J48, KNN	G6	NB, RT, NBM
G2	NB, J48, RT	G7	J48, KNN, RT
G3	NB, J48, NBM	G8	J48, KNN, NBM
G4	NB, KNN, RT	G9	J48, RT, NBM
G5	NB, KNN, NBM	G10	KNN, RT, NBM

5.1. Addressing RQ1. RQ1: Which classification algorithm works well in classifying surprise bugs?

Table 5 illustrates the performance of seven classification algorithms. All of the classification algorithms are demonstrated in the first row, and three evaluation metrics of prediction results about four testing projects and average values are shown in the first column.

Based on the average values of each classification algorithm, it can be seen that using NB to predict surprise bugs can achieve 0.347 of *F-Measure* on average and outperforms other classification algorithms. In addition, RF achieves the worst performance except SVM, only 0.196 of *F-Measure* on average. The investigations on the gap between RF and other algorithms in terms of *F-Measure* also show that although RF performs better in *precision*, its value of *recall* is lower. We believe that detecting more surprise bugs from all the bugs has more practical significance than detecting little surprise bugs with high accuracy for helping developer improve efficiency. Finally, we can see that SVM is linear and not applicable for predicting datasets with fuzzy boundary. In this experiment, SVM seems to have no ability to detect surprise bugs.

According to the prediction results of each project by different classification algorithms, it can be seen that the prediction performance by using classification algorithms (except SVM) is different. Therefore, it is unsuitable to predict all surprise bugs with one single classification algorithm. The comparison of seven classification algorithms shows that NB performs more stable on *F-Measure* than others and its range is 30%–40%. Besides, we can see that a severe fluctuation exists between classification results of RT. RT achieves minimum *F-Measure* which is only 9.1%, and the maximum is 51.5%.

In addition, it can be seen that the results of Camel outperform other projects, and the results of Derby have the worst performance based on the comprehensive performance of all the projects. We investigated the gap between results of each project and found that the distribution of category in each project is significantly different. We can see from Table 1 that Camel represents the most balanced distribution among four projects. Camel contains 228 surprise bugs, accounting for 39.38% of the total number. But the category distribution in the project named Derby is the most imbalanced, and the surprise bugs account for only 15.18%. Extremely imbalanced data bring a great challenge of predicting, and it is the reason why Derby produces the poor classification results.

In summary, the performance of classification algorithms is still not satisfied for us due to the instability and low accuracy. Thus, it is difficult to select the most

Table 5: The performance of seven classification algorithms.

Projects	Evaluation	NB	J48	KNN	RT	RF	NBM	SVM
Ambari	*Precision*	0.273	0.178	0.268	0.389	**0.412**	0.303	Null
	Recall	0.34	0.151	0.208	**0.396**	0.132	0.189	0
	F-Measure	0.303	0.163	0.234	**0.393**	0.2	0.233	Null
Camel	*Precision*	0.457	0.271	0.357	**0.49**	0.482	0.392	Null
	Recall	0.348	0.283	0.326	**0.544**	0.283	0.435	0
	F-Measure	0.395	0.277	0.341	**0.516**	0.356	0.412	Null
Derby	*Precision*	0.333	0.278	0.182	0.095	0.333	**0.5**	Null
	Recall	**0.409**	0.227	0.091	0.091	0.046	0.136	0
	F-Measure	**0.367**	0.25	0.121	0.093	0.08	0.214	Null
Wicket	*Precision*	0.341	0.373	0.311	0.35	0.263	**0.5**	Null
	Recall	0.306	0.388	0.286	**0.429**	0.102	0.245	0
	F-Measure	0.323	0.38	0.298	**0.385**	0.147	0.329	Null
avg*Precision*		0.351	0.275	0.28	0.331	0.372	**0.424**	Null
avg*Recall*		0.351	0.262	0.228	**0.365**	0.141	0.251	0
avg*F-Measure*		**0.347**	0.268	0.249	0.347	0.196	0.297	Null

appropriate classification algorithm to detect surprise bugs. In addition, we can see that RF and SVM are not appropriate for predicting surprise bugs according to the results of RF and SVM which achieve lower performance than other algorithms. Therefore, in the following experiments, we focus on 5 classification algorithms, which are NB, J48, KNN, RT, and NBM.

5.2. Addressing RQ2. RQ2: Can the combination of classification algorithms in OIILS perform better than other combinations of classification algorithms?

Table 6 illustrates the performance of 10 groups of classification algorithms. We demonstrate each combination of classification algorithms in the first row and demonstrate prediction performance of four projects through *precision*, *recall*, and *F-Measure* in the first column.

According to the average results of 10 groups, the combination of algorithms in OIILS (the group of G8) achieves 0.490 of *F-Measure* on average and achieves the best performance in terms of *F-Measure*. We compare G8 with the classification algorithm named NB which achieves best performance (0.347) among all of the algorithms mentioned above in Table 5. The comparison result shows that G8 increases by 156.13% in terms of *recall* on average and 41.21% in terms of *F-Measure* on average than NB, and it indicates that the combination for OIILS can improve the ability of detecting surprise bugs with single algorithm substantially. Additionally, it can be seen that different combinations of classification algorithms represent different performance on predicting surprise bugs. For instance, G3 only achieves 0.328 of average *F-Measure* and the performance is even worse than the performance of simply utilizing NB. It follows that improper combination of classification algorithms may worsen the performance of basic algorithms.

According to the performance results that each group classifies Ambari project, we can see that G9 achieves the best performance (0.492) in terms of *F-Measure*, while the performance of G3, G5, and G6 is worse than other groups. The investigation on these groups shows that only these groups contain NB and NBM. The poor performance of these groups may result from the weak complementarity between NB and NBM, and it is difficult to promote each other.

Meanwhile, it can be seen that the performance of all the groups predicting the projects named Camel is better than other projects. This is caused by two reasons. The one is that each algorithm predicts that Camel is better than other projects, while the other is that algorithm integration further expands the advantages of the classification algorithms. Additionally, it can be seen that G1, G4, G5, G7, and G10 achieves 1 of *recall* in terms of *recall*. In other words, the algorithms of these groups can cover all of the surprise bugs.

After the above analysis, it can be seen that the combined performance of the three classification algorithms used in OIILS is the best in all combinations.

5.3. Addressing RQ3. RQ3: Can OIILS outperform classification algorithm with imbalanced learning strategies?

Table 7 presents the prediction results of OIILS and 4 combinations by every 2 classification algorithms that perform best under each imbalanced learning strategies. The combinations are listed in the first row, and the evaluation of prediction results for each project and average is demonstrated in the first column.

According to the average results of each combination of classification algorithm and imbalanced learning strategy, SMOTE + KNN achieves the best performance in terms of *recall* and *F-Measure* among all the combinations. SMOTE + KNN achieves 0.820 of *recall* and 0.456 of *F-Measure* on average, and they are higher than those of RUS + J48 by 53.55% and 14.29%, respectively. For each projects, SMOTE + KNN achieves 0.7 ~ 0.9 of *recall* and performs better than other combinations substantially. Thus, SMOTE + KNN is more appropriate for predicting surprise bugs than other combinations.

Based on the analysis above, we compare the performance of SMOTE + KNN with OIILS. It can be seen that

Table 6: The performance of 10 groups of classification algorithms.

Projects	Evaluation	G1	G2	G3	G4	G5	G6	G7	G8	G9	G10
Ambari	*Precision*	0.308	0.378	0.382	0.305	0.28	0.329	0.313	0.311	**0.425**	0.301
	Recall	0.83	0.698	0.491	0.868	0.698	0.509	**0.887**	0.793	0.585	0.83
	F-Measure	0.449	0.49	0.43	0.451	0.4	0.4	0.463	0.447	**0.492**	0.442
Camel	*Precision*	0.414	0.433	0.385	0.414	0.414	0.424	0.414	0.409	**0.439**	0.414
	Recall	**1**	0.63	0.326	**1**	**1**	0.609	**1**	0.978	0.63	**1**
	F-Measure	**0.586**	0.513	0.353	**0.586**	**0.586**	0.5	**0.586**	0.577	0.518	**0.586**
Derby	*Precision*	0.211	0.143	0.143	0.208	0.218	0.182	0.215	**0.229**	0.206	0.222
	Recall	0.826	0.304	0.217	**0.87**	0.826	0.348	**0.87**	0.826	0.304	**0.87**
	F-Measure	0.336	0.194	0.172	0.336	0.346	0.239	0.345	**0.359**	0.246	0.354
Wicket	*Precision*	0.398	0.392	0.37	0.378	0.397	0.397	0.383	**0.405**	0.385	0.381
	Recall	**1**	0.633	0.347	0.98	0.98	0.592	**1**	**1**	0.51	0.98
	F-Measure	0.57	0.484	0.358	0.546	0.565	0.475	0.554	**0.577**	0.439	0.549
avg*Precision*		0.333	0.336	0.32	0.326	0.328	0.333	0.331	0.339	**0.364**	0.33
avg*Recall*		0.914	0.566	0.345	**0.929**	0.876	0.514	0.939	0.899	0.508	0.92
avg*F-Measure*		0.485	0.421	0.328	0.48	0.474	0.404	0.487	**0.49**	0.424	0.483

Table 7: The prediction results of OIILS and 4 combinations.

Projects	Evaluation	RUS		ROS		CMA		SMOTE		OIILS
		J48	RT	RT	NBM	NB	NBM	KNN	NBM	
Ambari	*Precision*	0.316	0.326	0.275	**0.339**	0.307	0.293	0.283	0.301	0.311
	Recall	0.453	0.585	0.264	0.415	0.434	0.415	0.672	0.362	**0.793**
	F-Measure	0.372	0.419	0.269	0.373	0.359	0.344	0.398	0.328	**0.447**
Camel	*Precision*	0.429	0.389	0.404	**0.45**	0.447	0.423	0.389	0.428	0.409
	Recall	0.522	0.457	0.457	0.587	0.37	0.652	0.896	0.604	**0.978**
	F-Measure	0.471	0.42	0.429	0.509	0.405	0.513	0.542	0.501	**0.577**
Derby	*Precision*	0.173	0.165	0.286	0.308	0.143	0.235	0.223	**0.358**	0.229
	Recall	0.609	0.565	0.261	0.174	0.174	0.174	0.8	0.13	**0.826**
	F-Measure	0.269	0.255	0.273	0.222	0.157	0.2	0.349	0.191	**0.359**
Wicket	*Precision*	0.429	0.333	0.333	0.447	0.365	0.447	0.381	**0.472**	0.405
	Recall	0.551	0.469	0.327	0.347	0.388	0.429	0.914	0.42	**1**
	F-Measure	0.482	0.39	0.33	0.391	0.376	0.438	0.537	0.445	**0.577**
avg*Precision*		0.337	0.303	0.324	0.386	0.316	0.35	0.319	**0.39**	0.339
avg*Recall*		0.534	0.519	0.327	0.381	0.341	0.418	0.82	0.379	**0.899**
avg*F-Measure*		0.399	0.371	0.325	0.374	0.324	0.374	0.456	0.366	**0.49**

OIILS achieves 0.899 in terms of *recall* on average and 0.49 in terms of *F-Measure* on average and increases by 9.63% on average *recall* and 7.46% on average *F-Measure* than SMOTE + KNN, respectively. We can see that OIILS achieves 0.793, 0.978, 0.826, and 1.0 in terms of *recall* for each projects, and increases by 18.01%, 9.15%, 3.25% and 9.41% than SMOTE + KNN respectively. In addition, OIILS achieves 0.447, 0.577, 0.359, and 0.577 in terms of *F-Measure* and increases by 12.31%, 6.46%, 2.87%, and 7.45% compared to SMOTE + KNN, respectively. It also can be seen from Table 7 that OIILS achieves about 1.0 in terms of *recall* when predicting Camel and Wicket projects and achieves 0.9 and 0.8 in terms of *recall* in other projects. In other words, OIILS can detect all the surprise bugs in projects which are relatively balanced and can detect most surprise bugs in projects which are relatively imbalanced.

From the experiment results, we could see that each part of our method is feasible and outperforms other corresponding classification algorithms, because OIILS could combine imbalanced learning strategy with multiclassification algorithms to overcome imbalance of datasets and take advantage of different abilities of three classification algorithms.

5.4. Addressing RQ4. RQ4: How accurate is the ensemble method of OIILS compared with the classic ensemble methods, including Adaboost, Bagging, and Vote?

Table 8 illustrates the performance of four ensemble methods. We demonstrate Adaboost, Bagging, Vote, and OIILS in the first row and demonstrate the prediction results of the four projects and the corresponding average values in the first column.

We compare OIILS with three classic ensemble methods named Adaboost, Bagging, and Vote. It can be seen from Table 8 that OIILS, respectively, achieves 0.339 and 0.899 in terms of *precision* and *recall* on average, improves the best *precision* of other ensemble methods by 5.61%, and improves the highest value of *recall* among classic ensemble methods by 6.01%. And OIILS achieves 0.49 in terms of *F-Measure* on

Table 8: The performance of four ensemble methods.

Projects	Evaluation	Adaboost	Bagging	Vote	OIILS
Ambari	*Precision*	0.298	0.295	0.276	**0.311**
	Recall	0.736	0.717	0.396	**0.793**
	F-Measure	0.424	0.418	0.326	**0.447**
Camel	*Precision*	0.387	0.383	0.373	**0.409**
	Recall	0.891	0.891	0.674	**0.978**
	F-Measure	0.539	0.536	0.481	**0.577**
Derby	*Precision*	0.213	0.207	0.222	**0.229**
	Recall	**0.826**	0.783	0.261	**0.826**
	F-Measure	0.339	0.327	0.24	**0.359**
Wicket	*Precision*	0.38	0.4	**0.405**	**0.405**
	Recall	0.939	0.939	0.653	**1**
	F-Measure	0.541	0.561	0.5	**0.577**
avg*Precision*		0.32	0.321	0.319	**0.339**
avg*Recall*		0.848	0.833	0.496	**0.899**
avg*F-Measure*		0.461	0.461	0.387	**0.49**

average and improves the best *F-Measure* of other methods by 6.29%. In addition, we can see that the ensemble method named Vote achieves the worst performance among the four ensemble methods. Obviously, the values of *recall* and *F-Measure* that Vote achieves are only about 60% of other ensemble methods.

For the purpose of finding out the reason of the worse performance of other ensemble methods, such as Vote, we investigate which classification algorithms are integrated in these methods. As mentioned in Section 5.1, different classification algorithms represent different prediction performance. For Vote, it is integrated by three algorithms which are J48, KNN, and NBM, and it can be seen the experimental results that KNN outperforms other algorithms obviously after adopting SMOTE. Thus, J48 and NBM bring a negative impact on the way of voting due to the poor performance of classifying.

Then we investigate why Vote does not perform well. As discussed in Section 5.1, different classification algorithms produce different prediction results based on same datasets. In this experiment, we use Vote to integrate three algorithms which are J48, KNN, and NBM, and we can see that KNN outperforms other algorithms obviously after adopting SMOTE according to the experimental results. Thus, J48 and NBM bring a negative impact on the way of voting due to the poor performance of classifying.

The experimental results show that OIILS could balance the ability of detecting different categories for each classifier based on the characteristics of the experimental data and integrate advantages of different classification algorithms. The ensemble method considers the optimization weight problem as a linear programming problem and utilizes the constraint solver named CPLEX to obtain the most suitable weights for higher accuracy of the results.

6. Conclusions

In this paper, we present a method named OIILS to identify surprise bugs. We consider textual feature of bug reports and utilize imbalanced learning strategy to assist classifiers for prediction. Then we use three classification algorithms to build classifiers, utilize these classifiers to train the same datasets which are balanced, and predict and test by the testing datasets. We also preset an ensemble method named Optimization Integration to combine the advantages of each classifier. Firstly, we set the weights to adjust the ability of detecting different categories based on the characteristics of projects for each classifier. Then, we adjust the probabilities of predicted results in order to obtain higher accuracy. Finally, we assign a label to each bug report to describe the extent of its impact according to the principle of the minimum value.

We have compared many basic classification algorithms for predicting imbalanced datasets to prove that classification algorithms used in OIILS is optimal. Then we have listed all the combinations of different classification algorithms, and the prediction results prove that the combination of OIILS achieves the best performance. Next, we have utilized four different imbalanced learning strategies and five classification algorithms to combine for predicting the testing datasets obtained in this experimental study. The prediction results show that the SMOTE used in OIILS outperforms other combinations. Finally, experimental results also prove that OIILS performs with higher accuracy of integration than classic ensemble methods.

With the help of OIILS, software project managers can identify the surprise bugs and assign these bugs to the developers to fix them in priority. Once the developers receive bug repair tasks, they will repair the surprise bugs as soon as possible and check the related code rigorously. It is sure that the software system quality can be better improved if the surprise bugs are repaired. OIILS can be easily used in software projects. The project managers can collect historical bug reports as training datasets, train the model, check the validity, and then apply it to identify surprise bugs from the newly submitted bug reports.

Additionally, OIILS could be benefit to the study of surprise bug identifications in software practices. Assembled with optimized integration and imbalance learning strategy, OIILS can improve the performance of surprise bug

identification, based on the experimental results on the datasets of real software projects. However, the performance of OIILS is different for the selected software projects more or less, and the problem has not been solved thoroughly. Thus further studies on surprise bug identification are still needed in the future.

In the future work, we plan to perform experiments with more imbalanced datasets and more open-source projects. We also plan to improve accuracy of OIILS without losing recall of minority category.

Finally, we plan to employ or design a more stable imbalanced learning strategy to make up the instability of SMOTE because the artificial datasets are produced randomly according to the data which belongs to minority category.

7. Threats to Validity

Our method and experiment design still contain some threats. We illustrate these threats as follows.

7.1. Conclusion Validity. The experiments evaluate OIILS though three metrics which are *precision*, *recall*, and *F-Measure*. Although utilizing accuracy to evaluate the performance of predicting imbalanced datasets is lack of practical significance, it is still an important indicator to assess the classification ability. Thus, the evaluation metrics in the experiments that completely ignore accuracy may not evaluate each prediction performance comprehensively.

7.2. Internal Validity. There are four open-source projects used as datasets in the experiments, and we divide each dataset into five parts randomly. For all of the experiments, we fix one of these parts as testing data and the remaining parts as training data. However, the performance of a classification algorithm predicting different datasets is different. Thus, the fixed training data and testing data used in these experiments may not completely show the performance of each classification algorithm.

7.3. Construct Validity. The type of bugs we concerned in this work is surprise bugs, and the surprise bugs are from four open-source projects. We investigated the other types of bugs in these projects and noticed that the category distribution of surprise bugs is more balanced than other types of bugs. Meanwhile, different category distribution may cause different performance for imbalanced learning strategy. Therefore, evaluating the performance of predicting imbalanced data through only one type of bugs may not be enough.

7.4. External Validity. We adopt imbalanced learning strategy named SMOTE in OIILS to balance the training datasets. As we know, SMOTE generates artificial data randomly based on initial data which belongs to minority category. However, the artificial data generated are different in each experiment. Thus, the performance of OIILS may randomly fluctuate due to SMOTE.

Data Availability

The data are available by contacting Dr. Guo via email at shikai.guo@dlmu.edu.cn.

Conflicts of Interest

The manuscript has no conflicts of interest.

Authors' Contributions

The idea of this work is provided by Shikai Guo; the code of this model is written by Guofeng Gao; data collection and experiments are conducted by Yang Qu; and the manuscript is written by Hui Li. Prof. Rong Chen and Prof. Chen Guo provided many suggestions on model design and manuscript revisions.

Acknowledgments

This work was supported by the National Natural Science Foundation of China (Nos. 61602077, 61902050, 61672122, 61771087, 51879027, 51579024, and 71831002), Program for Innovative Research Team in University of Ministry of Education of China (No. IRT17R13), High Education Science and Technology Planning Program of Shandong Provincial Education Department (Nos. J18KA340 and J18KA385), the Fundamental Research Funds for the Central Universities (Nos. 3132019355, 3132019501, and 3132019502), and Next-Generation Internet Innovation Project of CERNET (Nos. NGII20181205, NGII20190627, and NGII20181203).

References

[1] X. Xia, D. Lo, X. Wang, and B. Zhou, "Accurate developer recommendation for bug resolution," in *Proceedings of the Working Conference on Reverse Engineering*, pp. 72–81, Koblenz, Germany, October 2013.

[2] W. Pan, H. Ming, C. K. Chang, Z. Yang, and D.-K. Kim, "ElementRank: ranking java software classes and packages using a multilayer complex network-based approach," *IEEE Transactions on Software Engineering*, 2019.

[3] T. Xie, L. Zhang, X. Xiao, Y.-F. Xiong, and D. Hao, "Cooperative software testing and analysis: advances and challenges," *Journal of Computer Science and Technology*, vol. 29, no. 4, pp. 713–723, 2014.

[4] Q. Wang, S. Wang, Q. Cui, Q. Wang, M. Li, and J. Zhai, "Local-based active classification of test report to assist crowdsourced testing," in *Proceedings of the IEEE/ACM International Conference on Automated Software Engineering*, pp. 190–201, Singapore, September 2016.

[5] J. Wang, Q. Cui, Q. Wang, and S. Wang, "Towards effectively test report classification to assist crowdsourced testing," in *Proceeding of the ACM/IEEE International Symposium on Empirical Software Engineering and Measurement*, Ciudad Real, Spain, September 2016.

[6] M. R. Karim, A. Ihara, X. Yang, H. Iida, and K. Matsumoto, "Understanding key features of high-impact bug reports," in

Proceedings of the International Workshop on Empirical Software Engineering in Practice, pp. 53–58, Tokyo, Japan, 2017.

[7] H. Li, G. Gao, R. Chen, X. Ge, and S. Guo, "The influence ranking for testers in bug tracking systems," *International Journal of Software Engineering and Knowledge Engineering*, vol. 29, no. 1, pp. 1–21, 2019.

[8] T. J. Ostrand, E. J. Weyuker, and R. M. Bell, "Where the bugs are," in *Proceedings of the ACM Sigsoft International Symposium on Software Testing and Analysis*, pp. 86–96, Boston, MA, USA, 2004.

[9] M. D'Ambros, M. Lanza, and R. Robbes, "An extensive comparison of bug prediction approaches," in *Proceedings of the IEEE Working Conference on Mining Software Repositories*, pp. 31–41, Cape Town, South Africa, May 2010.

[10] F. Rahman and P. Devanbu, "How, and why, process metrics are better," in *Proceedings of the International Conference on Software Engineering*, pp. 432–441, San Francisco, CA, USA, 2013.

[11] J. Nam, S. J. Pan, and S. Kim, "Transfer defect learning," in *Proceedings of the International Conference on Software Engineering*, pp. 382–391, San Francisco, CA, USA, May 2013.

[12] Y. Kamei, E. Shihab, B. Adams et al., "A large-scale empirical study of just-in-time quality assurance," *IEEE Transactions on Software Engineering*, vol. 39, no. 6, pp. 757–773, 2013.

[13] W. Pan, B. Song, K. Li, and K. Zhang, "Identifying key classes in object-oriented software using generalizedk-core decomposition," *Future Generation Computer Systems*, vol. 81, pp. 188–202, 2018.

[14] X. Yang, D. Lo, X. Xia, Y. Zhang, and J. Sun, "Deep learning for just-in-time defect prediction," in *Proceedings of the IEEE International Conference on Software Quality, Reliability and Security*, pp. 17–26, Vancouver, Canada, August 2015.

[15] X. Yang, D. Lo, Q. Huang, X. Xia, and J. Sun, "Automated identification of high impact bug reports leveraging imbalanced learning strategies," in *Proceedings of the Computer Software and Applications Conference*, pp. 227–232, Atlanta, GA, USA, June 2016.

[16] M. Ohira, Y. Kashiwa, Y. Yamatani et al., "A dataset of high impact bugs: manually-classified issue reports," in *Proceedings of the IEEE Working Conference on Mining Software Repositories*, pp. 518–521, Florence, Italy, May 2015.

[17] E. Shihab, A. Mockus, Y. Kamei, B. Adams, and A. E. Hassan, "High-impact defects: a study of breakage and surprise defects," in *Proceedings of the ACM Sigsoft Symposium & the European Conference on Foundations of Software Engineering*, pp. 300–310, Szeged, Hungary, September 2011.

[18] X.-L. Yang, D. Lo, X. Xia, Q. Huang, and J.-L. Sun, "High-impact bug report identification with imbalanced learning strategies," *Journal of Computer Science and Technology*, vol. 32, no. 1, pp. 181–198, 2017.

[19] A. Lamkanfi, S. Demeyer, E. Giger, and B. Goethals, "Predicting the severity of a reported bug," in *Proceedings of the IEEE Working Conference on Mining Software Repositories*, pp. 1–10, Cape Town, South Africa, May 2010.

[20] X. Xia, D. Lo, M. Wen, E. Shihab, and B. Zhou, "An empirical study of bug report field reassignment," in *Proceedings of the IEEE Conference on Software Maintenance, Reengineering, and Reverse Engineering*, pp. 174–183, Antwerp, Belgium, 2014.

[21] S. Guo, R. Chen, M. Wei, H. Li, and Y. Liu, "Ensemble data reduction techniques and multi-RSMOTE via fuzzy integral for bug report classification," *IEEE ACCESS*, vol. 6, pp. 45934–45950, 2018.

[22] S. Guo, R. Chen, H. Li, T. Zhang, and Y. Liu, "Identify severity bug report with distribution imbalance by CR-SMOTE and ELM," *International Journal of Software Engineering and Knowledge Engineering*, vol. 29, no. 6, pp. 139–175, 2019.

[23] R. Chen, S. K. Guo, X. Z. Wang, and T. L. Zhang, "Fusion of multi-RSMOTE with fuzzy integral to classify bug reports with an imbalanced severity distribution," *IEEE Transactions on Fuzzy Systems*, vol. 27, no. 12, pp. 2406–2420, 2019.

[24] V. López, A. Fernández, S. García, V. Palade, and F. Herrera, "An insight into classification with imbalanced data: empirical results and current trends on using data intrinsic characteristics," *Information Sciences*, vol. 250, no. 11, pp. 113–141, 2013.

[25] H. Li, X. Yang, Y. Li, L.-Y. Hao, and T.-L. Zhang, "Evolutionary extreme learning machine with sparse cost matrix for imbalanced learning," *ISA Transactions*, 2019.

[26] N. Bhargava and G. Sharma, "Decision tree analysis on J48 algorithm for data mining," *International Journal of Advanced Research in Computer Science and Software Engineering*, vol. 3, no. 6, pp. 1114–1119, 2013.

[27] IBM ILOG CPLEX Optimizer, 2018, https://www.ibm.com/analytics/data-science/prescriptive-analytics/cplex-optimizer/.

[28] J. Anvik, L. Hiew, and G. C. Murphy, "Who should fix this bug?," in *Proceedings of the International conference on Software Engineering*, pp. 361–370, Shanghai, China, 2006.

[29] W. Deng, J. Xu, and H. Zhao, "An improved ant colony optimization algorithm based on hybrid strategies for scheduling problem," *IEEE ACCESS*, vol. 7, pp. 20281–20292, 2019.

[30] W. Deng, H. Zhao, X. Yang, J. Xiong, M. Sun, and B. Li, "Study on an improved adaptive PSO algorithm for solving multi-objective gate assignment," *Applied Soft Computing*, vol. 59, pp. 288–302, 2017.

[31] W. Deng, H. Zhao, L. Zou, G. Li, X. Yang, and D. Wu, "A novel collaborative optimization algorithm in solving complex optimization problems," *Soft Computing*, vol. 21, no. 15, pp. 4387–4398, 2017.

[32] N. Bettenburg, S. Just, R. Premraj, and T. Zimmermann, "Quality of bug reports in eclipse," in *Proceedings of the OOPSLA Workshop on Eclipse Technology Exchange*, pp. 21–25, Montreal, Canada, 2007.

[33] S. Demeyer and A. Lamkanfi, "Predicting reassignments of bug reports—an exploratory investigation," in *Proceedings of the European Conference on Software Maintenance and Reengineering*, pp. 327–330, Genova, Italy, March 2013.

[34] L. Wu, B. Xie, G. E. Kaiser, and R. J. Passonneau, "BUGMINER: software reliability analysis via data mining of bug reports," in *Proceedings of the International Conference on Software Engineering Knowledge Engineering*, pp. 95–100, Miami Beach, FL, USA, 2011.

[35] L.-Y. Hao, H. Zhang, W. Yue, and H. Li, "Quantized sliding mode control of unmanned marine vehicles: various thruster faults tolerated with a unified model," *IEEE Transactions on Systems, Man, and Cybernetics: Systems*, 2020.

[36] L.-Y. Hao, H. Zhang, W. Yue, and H. Li, "Fault-tolerant compensation control based on sliding mode technique of unmanned marine vehicles subject to unknown persistent ocean disturbances," *International Journal of Control, Automation, and Systems*, 2019.

[37] T. Menzies and A. Marcus, "Automated severity assessment of software defect reports," in *Proceedings of the IEEE International Conference on Software Maintenance*, pp. 346–355, Beijing, China, September-October 2008.

[38] A. Lamkanfi, S. Demeyer, Q. D. Soetens, and T. Verdonck, "Comparing mining algorithms for predicting the severity of a reported bug," in *Proceedings of the European Conference on Software Maintenance and Reengineering*, pp. 249–258, Oldenburg, Germany, 2011.

[39] Y. Tian, D. Lo, and C. Sun, "Information retrieval based nearest neighbor classification for fine-grained bug severity prediction," in *Proceedings of the Working Conference on Reverse Engineering*, pp. 215–224, Kingston, Canada, 2012.

[40] L. Yu, W. T. Tsai, W. Zhao, and F. Wu, "Predicting defect priority based on neural networks," in *Proceedings of the International Conference on Advanced Data Mining & Applications*, pp. 356–367, Chongqing, China, November 2010.

[41] J. Kanwal and O. Maqbool, "Bug prioritization to facilitate bug report triage," *Journal of Computer Science and Technology*, vol. 27, no. 2, pp. 397–412, 2012.

[42] Y. Tian, D. Lo, and C. Sun, "Drone: predicting priority of reported bugs by multi-factor analysis," in *Proceedings of the IEEE International Conference on Software Maintenance*, pp. 200–209, Eindhoven, Netherlands, September 2013.

[43] G. Gao, H. Li, R. Chen, X. Ge, and S. Guo, "Identification of high priority bug reports via integration method," in *Proceedings of the CCF Conference on Big Data*, pp. 336–349, Xi'An, China, 2018.

[44] A. Sureka and P. Jalote, "Detecting duplicate bug report using character N-gram-based features," in *Proceeding of the Asia Pacific Software Engineering Conference*, pp. 366–374, Sydney, Australia, 2011.

[45] C. Sun, D. Lo, S. C. Khoo, and J. Jiang, "Towards more accurate retrieval of duplicate bug reports," in *Proceedings of the IEEE/ACM International Conference on Automated Software Engineering*, pp. 253–262, Lawrence, KS, USA, November 2011.

[46] Y. Tian, C. Sun, and D. Lo, "Improved duplicate bug report identification," in *Proceedings of the European Conference on Software Maintenance and Reengineering*, pp. 385–390, Szeged, Hungary, 2012.

[47] L. Feng, L. Song, C. Sha, and X. Gong, "Practical duplicate bug reports detection in a large web-based development community," in *Proceedings of the Asia-Pacific Web Conference*, pp. 709–720, Sydney, Australia, 2013.

[48] A. Alipour, A. Hindle, and E. Stroulia, "A contextual approach towards more accurate duplicate bug report detection," in *Proceedings of the Working Conference on Mining Software Repositories*, pp. 183–192, San Francisco, CA, USA, 2013.

[49] N. Pingclasai, H. Hata, and K. I. Matsumoto, "Classifying bug reports to bugs and other requests using topic modeling," in *Proceedings of the Asia-Pacific Software Engineering Conference*, pp. 13–18, Bangkok, Thailand, 2014.

[50] K. Herzig, S. Just, and A. Zeller, "It's not a bug, it's a feature: how misclassification impacts bug prediction," in *Proceedings of the International Conference on Software Engineering*, pp. 392–401, San Francisco, CA, USA, 2013.

[51] M. Alenezi, K. Magel, and S. Banitaan, "Efficient bug triaging using text mining," *Journal of Software*, vol. 8, no. 9, pp. 2185–2190, 2013.

[52] R. Shokripour, J. Anvik, Z. M. Kasirun, and S. Zamani, "Why so complicated? Simple term filtering and weighting for location-based bug report assignment recommendation," in *Proceedings of the Working Conference on Mining Software Repositories*, pp. 2–11, San Francisco, CA, USA, 2013.

[53] J. B. Lovins, "Development of a stemming algorithm," *Mechanical Translation and Computational Linguistics*, vol. 11, no. 6, pp. 22–31, 1968.

[54] S. Wang and X. Yao, "Using class imbalance learning for software defect prediction," *IEEE Transactions on Reliability*, vol. 62, no. 2, pp. 434–443, 2013.

[55] H. He and E. A. Garcia, "Learning from imbalanced data," *IEEE Transactions on Knowledge Data Engineering*, vol. 21, no. 9, pp. 1263–1284, 2009.

[56] Y. Kamei, A. Monden, S. Matsumoto, T. Kakimoto, and K. I. Matsumoto, "The effects of over and under sampling on fault-prone module detection," in *Proceedings of the International Symposium on Empirical Software Engineering and Measurement*, pp. 196–204, Madrid, Spain, 2007.

[57] T. M. Khoshgoftaar, X. Yuan, and E. B. Allen, "Balancing misclassification rates in classification-tree models of software quality," *Empirical Software Engineering*, vol. 5, no. 4, pp. 313–330, 2000.

[58] N. V. Chawla, K. W. Bowyer, L. O. Hall, and W. P. Kegelmeyer, "SMOTE: synthetic minority over-sampling technique," *Journal of Artificial Intelligence Research*, vol. 16, no. 1, pp. 321–357, 2002.

[59] Y. Liu, X. Wang, Z. Zhai, R. Chen, B. Zhang, and Y. Jiang, "Timely daily activity recognition from headmost sensor events," *ISA Transactions*, vol. 94, pp. 379–390, 2019.

[60] C. G. Weng and J. Poon, "A new evaluation measure for imbalanced datasets," in *Proceedings of the Australasian Data Mining Conference*, pp. 27–32, Glenelg, Australia, 2008.

[61] J. Han and M. Kamber, *Data Mining: Concepts and Techniques'*, Morgan Kaufmann Publishers Inc., Burlington, MA, USA, 2nd edition, 2006.

[62] S. B. Kotsianti and D. Kanellopoulos, "Combining bagging, boosting and dagging for classification problems," in *Proceedings of the Knowledge-Based Intelligent Information and Engineering Systems and Italian Workshop on Neural Networks*, pp. 493–500, Vietri sul Mare, Italy, 2007.

[63] L. Rokach, "Ensemble-based classifiers," *Artificial Intelligence Review*, vol. 33, no. 1-2, pp. 1–39, 2010.

[64] G. Wang, J. Sun, J. Ma, K. Xu, and J. Gu, "Sentiment classification: the contribution of ensemble learning," *Decision Support Systems*, vol. 57, no. 1, pp. 77–93, 2014.

[65] A. G. Karegowda, M. A. Jayaram, and A. S. Manjunath, "Cascading k-means with ensemble learning: enhanced categorization of diabetic data," *Journal of Intelligent Systems*, vol. 21, no. 3, pp. 237–253, 2012.