applied sciences

MDPI

Article

Identify High-Impact Bug Reports by Combining the Data Reduction and Imbalanced Learning Strategies

Shikai Guo [1,2,3,4], Miaomiao Wei [2], Siwen Wang [2], Rong Chen [2], Chen Guo [1,*] and Hui Li [2,3] and Tingting Li [4]

1 The College of Marine Electrical Engineering, Dalian Maritime University, Dalian 116026, China
2 The College of Information Science and Technology, Dalian Maritime University, Dalian 116026, China
3 Collaborative Innovation Center for Transport Studies of Dalian Maritime University, Dalian 116026, China
4 Key Laboratory of Symbolic Computation and Knowledge Engineering of Ministry of Education, Jilin University, Changchun 130012, China
* Correspondence: dmuguoc@126.com

Received: 20 August 2019; Accepted: 1 September 2019; Published: 4 September 2019

Abstract: As software systems become increasingly large, the logic becomes more complex, resulting in a large number of bug reports being submitted to the bug repository daily. Due to tight schedules and limited human resources, developers may not have enough time to inspect all the bugs. Thus, they often concentrate on the bugs that have large impacts. However, there are two main challenges limiting the automation technology that would help developers to become aware of high-impact bug reports early, namely, low quality and class distribution imbalance. To address these two challenges, we propose an approach to identify high-impact bug reports that combines the data reduction and imbalanced learning strategies. In the data reduction phase, we combine feature selection with the instance selection method to build a small-scale and high-quality set of bug reports by removing the bug reports and words that are redundant or noninformative; in the imbalanced learning strategies phase, we handle the imbalanced distributions of bug reports through four imbalanced learning strategies. We experimentally verified that the method of combining the data reduction and imbalanced learning strategies could effectively identify high-impact bug reports.

Keywords: high-impact bug reports; class imbalance; feature selection; instance selection

1. Introduction

Bug tracking systems, such as Bugzilla [1] and JIRA [2], can help developers to manage the bug reports collected from various sources, including development teams, testing teams, and end users [3]. Due to the increased scale and complexity of software projects, a large number of bug reports are received daily by bug tracking systems. For example, the Mozilla bug repository receives an average of 135 new bug reports each day [4]. Due to the large number of bugs submitted to the bug repository every day, in order to correctly verify the severity of the bug report and solve the problems of manual bug classification, such as high time consumption and low accuracy, it is becoming increasingly important to automatically identify high-impact bug reports. However, we face the following two challenges: low quality data and data distribution imbalances. Since bug reports are submitted by people from all over the world and each person's description in their natural language and understanding of bugs are different, there is excessive noise in the data [5]. Noisy data may mislead the data analysis techniques, and large-scale data may increase the cost of data processing [6]. The low quality bugs accumulate in bug repositories and grow in scale. Furthermore, most of the bug reports are not high impact bug reports; in other words, the training set often has an imbalanced distribution. This is a disadvantage for most of the existing classification approaches, which have

largely been developed under the assumption that the underlying training set is evenly distributed. These two challenges, the large amount and low quality of the data, will affect the performance of the bug classification. To solve the problems of software projects quickly, developers often need to prioritize and concentrate on the high-impact bug reports. Therefore, an automated technique to tell developers whether or not a bug report is a high-impact bug report, would be preferable to augment productivity.

However, although a large number of bug reports are submitted daily by people from all over the world, only a small percentage of the bug reports are high-impact bug reports, and each person's description in their natural language and understanding of bugs are different, thus resulting in excessive noise [5]. Noisy data may mislead the data analysis techniques, while large-scale data may increase the cost of data processing [6–8]. The low quality bugs accumulate in the bug repositories and grow in scale. The time cost of manual bug classification is costly and has low accuracy. In manual bug classification in Eclipse, 44% of the bugs were misallocated, and the time cost between opening a bug and its first classification averaged 19.3 days [9]. To avoid the expensive cost of manual bug classification, Anvik et al. [10] proposed an automatic bug classification method, which uses text classification technologies to predict the developers of bug reports. In this method, a bug report is mapped to a document, and the relevant developer maps to the label of the document. Then, a text classifier is used to automatically resolve the bug reports, such as Naive Bayes [11]. Based on the results of the text categorization, manual categorizers assign new bugs by combining their professional abilities. However, the low-quality and class imbalance distribution of the bug reports in bug repositories will have an adverse effect on automatic bug classification techniques. Since the software bug reports are free-form text datasets (generated by the developer), it is necessary to process the bug datasets into high-quality bugs to facilitate the application [6]. Xuan et al. proposed a method that combines the feature selection with instance selection to reduce bug datasets to obtain high-quality bug data and to determine the performance of the bug classification by changing the order of reduction. Due to the huge size and noise of the bug repository, it is still difficult to reduce the bug data set to a completely high-quality data set. Yang [12] investigated four widely used imbalanced learning strategies (i.e., random under-sampling (RUS), random over-sampling (ROS), synthetic minority over-sampling technique (SMOTE) and cost-matrix adjuster (CMA)) to solve the class imbalance distribution of bug reports from four different open source projects. However, the bug reports are written in natural language, which includes much noise. Therefore, the internal methods (CMA) are unsuitable to balance the distribution of bug reports [13]. In addition, RUS could cause under-fitting, whereas ROS could cause over-fitting [14–17]. Additionally, the SMOTE is based on low-dimensional random sampling, which has a poor generalization ability [18,19]. Furthermore, random sampling (RUS, ROS, SMOTE) could bring a certain degree of uncertainty, and some sampling results are not in accord with the real distribution of the dataset.

In this paper, we propose an approach for identifying high-impact bug reports by combining data reduction, named Data Reduction based on Genetic algorithm (DRG), with imbalanced learning strategies. In data reduction phase, we use four feature selection algorithms (FS) (i.e., one rule (OneR), information gain (IG), chi-square (CHI), and filtered selection (Relief) (Described in Appendix A.2)) to extract 10%, 20%, 30%, 40%, 50%, 60%, 70%, 80% and 90% of the attributes from bug reports with reduced bug data dimensions and word dimensions to solve the low-quality bug reports challenge and obtain 36 feature selection schemes, i.e., feature reduction (GA(FS)), and use instance selection (IS) algorithms (i.e., condensed nearest neighbor (CNN), edited nearest neighbor (ENN), minimal consistent set (MCS), and iterative case filter (ICF)) to randomly generate 36 instance selection schemes, i.e., instance reduction (GA(IS)). Simultaneous feature and instance reduction (i.e., GA (FS+IS)) is a combined extraction scheme of 36 features and instance strings; imbalanced learning strategies are applied to the dataset after the reduction operation. In the imbalanced learning strategies phase, we investigate four widely used imbalanced learning strategies, i.e., RUS, ROS, SMOTE and CMA, and four popular classfication algorithms, i.e., naive Bayes (NB), naive Bayes multinomial

(NBM), support vector machine (SVM) and K-nearest neighbors (KNN) (Described in Appendix A.1), and combine them to make a total of 16 different combinations (i.e., variants) to identify high-impact bug reports for reduced datasets. Comprehensive experiments have been conducted on public datasets obtained from real-world bug repositories (Mozilla [20], GCC [21] and Eclipse [22]). In addition, we also verify the performance of our method in the case of different combinations of FS and IS, data reduction and imbalanced learning strategies.

The main contributions of this paper are as follows:

- We present a data reduction method based on the genetic algorithm to reduce the data for identifying high-impact bug reports. This problem aims to augment the data set by identifying two aspects of the high-impact bug reports, namely, (a) to simultaneously reduce the scales of the bug report dimension and the word dimension and (b) to improve the ability to identifying high-impact bug reports.
- We verify the ability of our method to identify the high-impact bug reports in the case of different combinations of FS and IS, data reduction and imbalanced learning strategies.
- Five evaluation criteria are used in experimental part to evaluate the proposed method. The results from three public bug repository datasets (Mozilla, GCC and Eclipse) show that our method could effectively identify the high-impact bug reports.

The remainder of this paper is organized as follows: the related studies and the motivation of our approach are discussed in Section 2, the design of our approach is discussed in Section 3, the experimental design and results are presented in Sections 4 and 5, and the conclusions are discussed in Section 6.

2. Background Knowledge and Motivation

In our work, we propose a high-impact bug report identification approach that combines data reduction, i.e., Data Reduction based on the Genetic algorithm (DRG), and imbalanced learning strategies. Thus, in this section, we introduce some background knowledge concerning bug report management. Moreover, we present the motivation of our study.

Bug reports are valuable to software maintenance activities [23–26]. Automatic support for bug report classification can facilitate understanding, resource allocation, and planning.

Antoniol et al. applied text mining techniques to the description of bug reports to determine whether a bug report is a real bug or a feature request [27]. They use techniques such as decision trees and logistic regression as well as the Naive Bayes classifier to achieve this purpose. Menzies et al. use rule learning techniques to predict the severity of bug reports [28]. Their approach was applied to five projects supplied by NASA's Independent Verification and Validation Facility. Tian et al. [29] use the overall framework of machine learning to predict the bug report priority by considering factors such as time, text, author, related reports, severity, and product. Hooimeijer et al. [30] established a predictive model to identify high-quality bug reports. The method effectively distinguishes between high-quality bug reports and low-quality bug reports based on extracting relevant descriptive information about the bug reports, products, operating systems, and bug report submitters. Runeson et al. [31] proposed a detection method for redundant bug reports based on information retrieval technology. This method treats each bug report as a document and obtains bug reports similar to the current bug report by calculating the similarity between the current bug report and the existing bug report. Sun et al. [32] proposed a feature-based partitioning model that identified the similarities between bug reports. Subsequently, Sun et al. [33] proposed a retrieval model based on multifeature information, which can continue to match the most similar features in bug reports.

Xia et al. [34] found that for approximately 80% of bug reports, the values of one or more of their fields (including the severity and fixer fields) are reassigned. They also show that redistributing field values for bug reports with existing field values consumes more time and cost than bug reports for no field reassignment. To resolve this problem, it is necessary to develop automatic approaches to perform

severity predictions and semiautomatic fixer recommendations. Zhang [35] proposed a new approach for severity predictions and semiautomatic fixer recommendations to take the place of manual work by developers. In this approach, the top k nearest neighbors of a new bug report are extracted by computing a similarity measure called REP topic, which is an enhanced version of REP. Next, the severity prediction and semiautomatic fixer recommendation approaches are implemented based on the characteristics of these k neighbors, such as participating developers and textual similarities with the given bug report.

Feng et al. [36] applied test report prioritization methods to crowdsourced testing. They designed dynamic strategies to select the most dangerous and diverse test reports for inspection in each iteration. In their subsequent article, Feng et al. [37] proposed a new technology to prioritize test reports for inspection by software developers. This approach combined image-understanding techniques with traditional text-based techniques, especially for the crowdsourced testing of mobile applications. They proposed prioritization approaches that are based on text descriptions, screenshot images, and a combination of both sources of information. Under the premise of obtaining rich training data, Wang et al. [38] proposed a cluster-based classification approach for crowdsourcing report classification. However, sufficient training data are often not available. Subsequently, Wang et al. [39] proposed an approach called local-based active classification (LOAF) to address the local bias problem and the lack of labeled historical data that exist in the automated crowdsourced testing report classification.

3. Methodology

In this section, we will describe in detail the methods for identifying bug reports with an imbalanced distribution.

3.1. Overview

In this part, we propose a high-impact bug report identification approach by combining data reduction, i.e., Data Reduction based on Genetic algorithm (DRG), and imbalanced learning strategies. The model consists of the following three parts: (1) Data preprocessing: We use a text categorization technique to convert each bug report into a word vector based on the vector space model that is mentioned in [11,40]. (2) Data reduction phase: Low quality bug reports may cause the classification approach to assign bug reports to the wrong category [41–44]. To obtain high-quality bug reports, we use our proposed DRG approach to obtain a reduced dataset. There are three types of reduction methods, as follows: feature reduction (GA(FS)), instance reduction (GA(IS)) and the reduction of features and instances simultaneously (GA(FS+IS)). (3) Imbalance learning strategies: To eliminate the imbalance of datasets, we use four imbalanced processing strategies, i.e., RUS, ROS, SMOTE and CMA, to balance the processing of datasets and solve the class imbalance problem [19,45–47]. Figure 1 shows the overall framework of our proposed method.

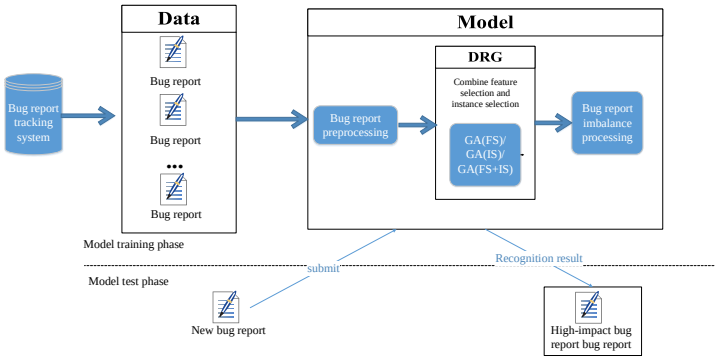

Figure 1. The framework of our model.

3.2. DRG Algorithm

The DRG algorithm mainly includes the following six steps: gene coding, population initialization, selection, crossover, mutation and termination criteria judgment. Its execution process is shown in Figure 2:

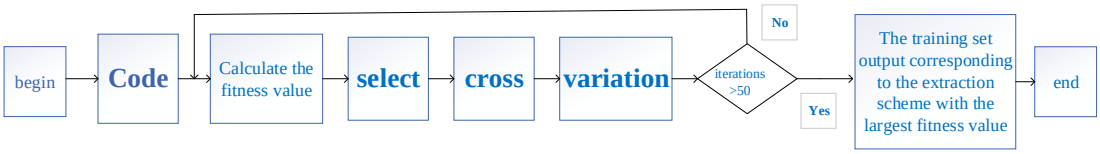

Figure 2. DRG algorithm execution process.

Gene coding: The feature sequence in the dataset is represented as a vector FD of size $1 \times \mathrm{N}$, where N is the total number of features. We define NP as the number of populations, it is different in GA(FS), GA(IS) and GA(FS+IS). A combination of selected features is represented as a binary string, $F_{np} = \{\mathrm{f}_1, \mathrm{f}_2, \ldots, \mathrm{f}_i, \ldots, \mathrm{f}_N\}$, $i \in [1, \mathrm{N}]$, $np \in [1, \mathrm{NP}]$. F_{np} represents a feature selection scheme. Each feature F_i is a $0-1$ variable with two values: when $\mathrm{f_i} = 0$, it means that the feature in FD corresponding to F_i is not selected, and $\mathrm{f_i} = 1$ indicates that the feature in FD corresponding to F_i is selected. Similarly, the instance sequence in the dataset is represented as a vector SD of size $1 \times \mathrm{M}$, where M is the total number of instances (that is the number of bug reports). Each combination of selected instances is a instance selection scheme, it is represented as a binary string, $S_{np} = \{\mathrm{s}_1, \mathrm{s}_2, \ldots, \mathrm{s}_j, \ldots, \mathrm{s}_M\}$, $j \in [1, \mathrm{M}]$, $np \in [1, \mathrm{NP}]$. Each instance s_j is a $0-1$ variable. If the instance in SD corresponding to s_j is not selected, $\mathrm{s_j} = 0$. And $\mathrm{s_j} = 1$ means that the instance in SD corresponding to s_j is selected.

Population initialization. The population initialization in GA(FS) is the generation of initial feature selection scheme. The method is to use four basic feature selection algorithms (IG, CHI, OneR, Relief) to sort the dataset according to the importance of the features from high to low, and then take 10%, 20%, 30%, 40%, 50%, 60%, 70%, 80% and 90% of the sorted dataset respectively, and thus 36 feature selection schemes are obtained. The population initialization in GA(IS) is the generation of the initial instance selection scheme. The method is to randomly generate a random number between [0, 1] at each s_j in each s_{np} . If the random number is greater than or equal to 0.5, $\mathrm{s_j} = 1$, otherwise $\mathrm{s_j} = 0$, thus obtaining 36 instance selection schemes. The method of generating the initial population of GA(FS+IS) has two steps: Firstly, 36 extraction schemes are generated according to the initial population generation methods in GA(FS) and GA(IS); Then, the generated F_{np} and S_{np} were separately combined to obtain 36 extraction schemes in the initial population.

Selection: The fitness value of each extraction scheme is calculated according to the fitness function. The extraction scheme with the largest fitness value is copied aNP times, and they will be used as aNP schemes in the next generation population (it guarantees that the extraction scheme with high fitness can be retained). Then, using the roulette selection method, the remaining $(1-a)NP$ extraction schemes in the next generation population is generated.

Crossover: Divide all extraction schemes in the population into two groups: $[1, \mathrm{mid}]$ and $[mid+1, NP]$, and mid is equal to $NP/2$. The extraction schemes in the population are combined in order, and then a single point crossover is performed. We define a crossover probability $Cross\_Ratio$, it is randomly generated value. If $Cross\_Ratio$ is greater than or equal to the crossover probability lower bound $Cross\_L$ and less than or equal to the crossover probability upper bound $Cross\_H$, the crossover of these two extraction schemes will be carried out, otherwise they will not crossover. If crossover is performed, a crossover point is randomly generated, then all the gene positions after this point in both two extraction schemes will be exchanged.

Mutation. We define a mutation rate $Variation\_Ratio$. For each extraction scheme in population, a mutation possibility is randomly generated, if the mutation possibility is less than the $Variation\_Ratio$, the mutation operation is performed. If mutation, $Variation\_Num$ mutation gene positions are

randomly selected in the extraction scheme, if the value of a gene position is 0, it is changed to 1, and if it is 1, it is changed to 0.

Termination criteria judgment. The iteration is terminated when the defined number of iterations T is reached.

3.3. Reduction Algorithm (GA)

In Sections 3.3 and 3.4, the input parameters and the meaning of each parameter in the algorithm are as follows: *Train* : The training set. NP : Population size. T : The number of iterations. N: The number of features. M : The number of bug reports. *Cross_L*: The lower bound of the crossover probability. *Cross_H*: The upper bound of the crossover probability. *Variation_Ratio* : Mutation rate. *Variation_Num* : The number of mutated genes.

Algorithm 1 indicates that only the feature selection (GA(FS)) or only instance selection (GA(IS)) is performed on the dataset by the genetic algorithm. In the first line of the algorithm, both the population and the best extraction scheme (*best_individual*) are initialized to null. In lines 2–5, if the algorithm performs feature selection, Algorithm 2 is called to initialize the population; if the algorithm performs instance selection, Algorithm 3 is called to initialize the population. In lines 7–8, the fitness value of each extraction scheme in the population is calculated by the fitness function defined in Section 3.5, and the extraction scheme with the largest fitness value is recorded. Lines 9–11 represent Selection operation, the extraction scheme with the largest fitness function value is copied, and it is passed to the next generation as aNP of the population. Then, the roulette selection method is used to generate the remaining $(1-a)NP$ extraction schemes in the population. Lines 12–23 describe the Crossover operation, lines 24–30 describe the mutation operation, and the specific methods of the crossover and mutation are detailed in Section 3.1. The mutation loci represent the number of mutant genes. In lines 30–31, all extraction schemes update the population after performing the genetic operation. In line 32, after performing T iterations, the obtained optimal extraction scheme is decoded into a corresponding dataset *Reduced_Train*. In line 33, return the reduced dataset *Reduced_Train*.

Algorithm 2 represents the population initialization for feature selection using genetic algorithms. In line 1, initial the population to null. In line 2,metric is used to represent the four feature selection methods (IG, CHI, OneR, and Relief). In lines 3–9, the training set is sorted by the four feature selection algorithms according to importance of all features from high to low. The first 10%, 20%, 30%, 40%, 50%, 60%, 70%, 80%, and 90% of the sorted feature set are added to the feature selection scheme respectively added to the feature selection scheme (*Initialize_Population_FS*). In line 10, the corresponding column of the available feature is set to 1, and the other to 0, return *Initialize_Population_FS*.

Algorithm 3 represents the population initialization for the instance selection using genetic algorithms. In line 1, the initial population is set to null. In lines 2–13, first, each instance selection scheme is organized into a binary string of size $1 * M$.Then, a number between 0 and 1 is randomly generated for each gene position in each extraction scheme; if this number is greater than or equal to 0.5, its corresponding gene position is set to 1, otherwise, it is set to 0, and each instance selection generated will be added into *Initialize_Population_IS*. In line 14, all the extraction schemes produced will be returned.

3.4. GA(FS+IS) Algorithm

GA(FS+IS) indicates that the feature and instance selection are simultaneously performed on the dataset by using the genetic algorithm. Algorithm 4 details the method of simultaneous feature selection and instance selection. In line 1, the population and the best extraction scheme (*best_individual* are initialized to null. In line 2 to 5, the initialized feature selection scheme and the instance selection scheme are combined. Lines 7–8 indicate that in each iteration, the fitness value of each extraction scheme in the population is calculated by the fitness function described in Section 3.5, and the extraction scheme with the largest fitness value will be recorded in (*best_individual* Lines 9–11 are the Selection operation; the extraction scheme with the largest fitness value will be copied for

inheritance to the next generation, and the number of this scheme accounts for a of the population. The other $1 - a$ of the population will be generated by the roulette selection method. Lines 12–22 are the crossover operation; the crossover is performed on feature selection scheme and instance selection. In lines 23–35, the mutation is performed separately on the feature selection scheme and instance selection. Line 46 indicates that the population needs to be updated after each iteration. Line 38 indicates that after completing T iterations, the best extraction scheme is retained by (*best_individual* and it is decoded into the corresponding dataset *Reduced_Train* $(FS + IS)$. The algorithm finally returns the reduced dataset.

Algorithm 1 Reduction algorithm(GA)

Input:

Train, NP, T, N, M, $Cross\_L$, $Cross\_H$, $Variation\_Ratio$, $Variation\_Num$

Output:

Reduced_Train

1: Initialize: $Population \leftarrow \emptyset, best\_individual \leftarrow \emptyset, t = 0$//$t$ represents the current iteration
2: **IF** feature select **then**
3: Population = $Initialize\_FS(Train, NP, N)$
4: **ELSE IF** instance select **then**
5: Population = $Initialize\_IS(Train, NP, M)$
6: **While** $t < T$ **do**:
7: Calculate the fitness value of each individual by Fitness function
8: $best\_individual \leftarrow (The\ individual\ with\ the\ largest\ fitness\ value)$
9: //Selection
10: Copy aNP individuals with the highest fitness value to the next generation.
11: Use the roulette method produces the remaining $(1 - a)NP$ individuals.
12: //Crossover
13: Divide the population into two groups, $[1, NP/2]$ and $[NP/2, NP]$
14: **For** all i from 1 to $NP/2$ **do**:
15: Rate=Random(0,1)
16: **IF** (Cross_L$<=$ Rate$<=$ Cross_H) **then**
17: **IF** feature select **then**
18: cross_point = Random(0, N).
19: **ELSE IF** instance select **then**
20: cross_point = Random(0, M).
21: Individuals i and $(NP/2 + i)$ are crossed (the gene exchange after crossing points).
22: **ELSE IF** feature select **then**
23: **ELSE IF** instance select **then**
24: //Mutation
25: **For** all i from 1 to aNP **do**:
26: rate =Random(0,1).
27: **IF** (rate$<=$ $Variation\_Ratio$) **then**
28: Randomly generate $Variation\_Num$ mutation loci.
29: **For** all j from 1 to $Variation\_Num$ **do**:
30: If the locus is 0, set it to 1, and if it is 1 then set it to 0.
31: Updating Population.
32: The binary string corresponding to best_individual is decoded as *Reduced_Train*.
33: **return** *Reduced_Train*.

3.5. Fitness Function

Individuals need to be selected according to the fitness function value when the data set is reduced by the genetic algorithm. This function is used to measure the ability of the individual classification.

The higher the fitness function value is, the better the individual is. The fitness function is defined as follows [48]:

$$J(x) = S_b - S_w \tag{1}$$

where S_b represents the fuzzy distance between different categories, and S_w represents the fuzzy distance within the same category. The samples can be separated because they are located in different regions of the feature space. The larger the S_b and the smaller S_w, the better the classification effect is. The specific calculation method is described as follows:

Algorithm 2 Initialize_FS

Input:

T*rain*, NP, N

Output:

Initialize_Population_FS

Initialize_Population_FS ← ∅.
metrics ← [*IG*, *CHI*, *OneR*, *Relief*]
For each *metric* in *metric* **do**:
    Order features by *metric*
    For all i from 1 to 9 **do**:
        Take the first $i * 10$ of N as fs
        Initialize_Population_FS.add(fs)
    End for
End for
return Initialize_Population_FS.

Algorithm 3 Initialize_IS

Input:

T*rain*, NP, M

Output:

Initialize_Population_IS

Initialize_Population_IS ← ∅.
For all i from 1 to NP **do**:
    Treat individual i as *ins*.
    For *ins* gene location j from 1 to M **do**:
        rate=Random(0,1)
        If (rate> = 0.5) **then**
            Change the j locus of the ith individual to 1.
        else
            Change the j locus of the ith individual to 0.
        End if.
    End for.
    Initialize_Population_IS.add(ins)
End for.
return Initialize_Population_IS.

We adopt the following Euclidean distance [24]:

$$N(A,B) = \sqrt{\sum_{i=1}^{n} \left(u_A(x_i) - u_B(x_i)\right)^2} \tag{2}$$

When calculating the distance S_b between different categories, $u_A(x_i)$ and $u_B(x_i)$ represent the mean vectors of *severity* and *non_severity*, respectively. The mean vector can be obtained by the following formula:

$$c_i = \frac{1}{n_i} \sum_{x \in w_i} x, (i = 1, 2) \tag{3}$$

where w_i represents the two categories of *severity* and *non_severity*, c_i is the category center feature vector of the category i, and there are n_i bug reporters in w_i category.

Algorithm 4 GA(FS+IS)

Input:

$Train, NP, T, N, M, Cross\_L, Cross\_H, Variation\_Ratio, Variation\_Num$

Output:

$Reduced\_Train(FS + IS)$

Initialize: $Population \leftarrow \emptyset$,$best\_individual \leftarrow \emptyset$, $t = 0$//t represents the current iteration.
For all i from 1 to NP **do**:
    Initialize_Population_FS = Initialize_FS ($Train, NP$, N)
    Initialize_Population_IS = Initialize_IS ($Train, NP$, M)
    $Population \leftarrow Combine(Initialize\_Population\_FS, Initialize\_Population\_IS)$.
While $t < T$ **do**:
    Calculate the fitness value of each individual by the fitness function.
    $best\_individual \leftarrow$ The individual with the largest fitness value.
    //Selection
        Copy individuals with the highest fitness value for aNP times to the next generation.
        Use the roulette method produces the remaining $(1 - a)NP$ individuals.
    //Crossover
        Divide the population into two groups, $[1, NP/2]$ and $[NP/2, NP]$
        For all i from 1 to $NP/2$ **do**:
            Rate FS=Random(0,1).
            If (Cross L$<$ $=$ Rate FS$<$ $=$ Cross H) **then**
                corss point=Random (0, N).
                Individuals i and $(NP/2 + i)$ are crossed (the gene exchange after crossing points).
            Rate IS=Random(0,1).
            If (Cross L$<$ $=$ Rate IS$<$ $=$ Cross H) **then**
                corss point=Random (0, M).
                Individuals i and $(NP/2 + i)$ are crossed (the gene exchange after crossing points).
    // Mutation
        For all i from 1 to NP **do**:
            Rate FS=Random(0,1).
            If (Rate FS$<$ $=$ Variation_Ratio) **then**
                Randomly generate $N/100$ mutation loci.
                For all j from 1 to $N/100$ **do**:
                    If the locus is 0, set it to 1, and if it is 1 then set it to 0.
        For all i from 1 to NP **do**:
            Rate IS=Random(0,1).
            If (Rate IS$<$ $=$ Variation_Ratio) **then**
                Randomly generate $M/100$ mutation loci.
                For all j from 1 to $M/100$ **do**:
                    If the locus is 0, set it to 1, and if it is 1 then set it to 0.
    Updating Population.
End while.
The binary string corresponding to best_individual is decoded as $Reduced\_Train(FS + IS)$.
return $Reduced\_Train(FS + IS)$.

When calculating the distance S_w between the same category, $u_A(x_i)$ and $u_B(x_i)$ represent two different bug reports in the same category. For two different bug reports A and B in the same category, the inner class distance should be calculated, then, the inner class distance of the two category is added to obtain S_w . The definition is as follows:

$$N(A,B) = \frac{1}{P+Q}\sqrt{\sum_{i=1}^{n}\left(u_A(x_i) - u_B(x_i)\right)^2} \quad (4)$$

where $P = \sum_{i=1}^{m_1-1} i$, m_1 represents the number of *severity* bug reports; $P = \sum_{i=1}^{m_2-1} i$, m_2 represents the number of *nonseverity* bug reports;

3.6. Feature Selection Approach

Since many feature selection algorithms have been investigated for text categorization, we select four typical algorithms (OneR, IG, CHI, Relief) in our work [49].

4. Experimental Design

The experimental design used to validate the performance of our approach is described in this section.

4.1. Experimental Datasets

To demonstrate the effectiveness of the proposed approach, we carry out a series of experiments on the bug repositories of three large open source projects, namely, Mozilla, GCC, and Eclipse. Portions of the bug reports (that is, the repaired state is fixed) are selected as the experimental data. There are seven types of labels corresponding to the severity of the datasets, as follows: normal, enhancement, major, critical, blocker, trivial, and minor. The major, critical, and blocker tags are severe bug reports. The trivial and minor tags are *nonseverity* bug reports. Statistics information on the three datasets is shown in Table 1, which contains the total number of bug reports, the number of severe bug reports, the number of *nonseverity* bug reports, and imbalance ratio. We can see that all the datasets are imbalanced. In Mozilla, the proportion of severe bug reports is 1.3015. In Eclipse, the proportion of severe bug reports is 2.311197, and in GCC, the proportion of severe bug reports is 3.6735. In Table 1, the second column represents the size of the data set. The third and fourth columns indicate that the severity labels are major, critical, blocker, respectively. The *nonseverity* tags are trivial and minor. The fifth and sixth columns represent the number of severe bug reports and the number of *nonseverity* bug reports, respectively. The last column indicates the imbalance ratio of the bug report.

Table 1. The original datasets for Mozilla, GCC, and Eclipse.

Project	Number of Total Bug Reports (Row × Column)	Severity *Tag*	Nonseverity *Tag*	Number of Severity Bug Reports	Number of Nonseverity Bug Reports	Imbalance Ratio
Mozilla	18793 × 11682	*major*	*trivial*	1528	1174	1.3015
GCC	13964 × 16768	*critical*	*minor*	2149	585	3.6735
Eclipse	41799 × 27141	*blocker*		5073	2232	2.31197

We analyze and process the Mozilla, GCC and Eclipse datasets and remove the bug reports labeled normal and environment from each dataset, leaving only the bug reports labeled major, critical, blocker, trivial, and minor; then, text preprocessing is performed. The description information of each

bug report in the datasets is segmented, the stop words are removed, and the word stem is processed into a text matrix. Each row in the matrix represents a bug report, and each column represents a word. We delete words whose word frequency is less than 5. The datasets after word frequency reduction are divided into parts in chronological order; the first parts are used as training sets, and the last parts are used as test sets. We obtained the final standard experimental data sets, and the results are shown in Table 2. In Table 2, the second column represents the size of the data set with a word frequency less than 5, the third column represents the proportion of the deleted bug report, and the last two columns represent the number of bug reports with severe and nonsevere tags in the training and test sets, respectively.

Table 2. The preprocessed datasets for Mozilla, GCC, and Eclipse.

Project	CiPin5 (Row*Column)	The Proportion of Bugs Deleted	Training Sets (8 Copies)		Test Sets (2 Copies)	
			Severity	*Nonseverity*	*Severity*	*Nonseverity*
Mozilla	2702 × 2285	85.62%	1222	938	306	236
GCC	2734 × 3292	80.42%	1720	464	429	121
Eclipse	7305 × 4697	82.20%	4090	1750	983	482

4.2. Experimental Parameter Setting

The genetic algorithm is used to reduce the attributes and instances. The specific parameters of GA(FS), GA(IS) and GA(FS+IS) are shown in Table 3. The parameters in Table 3 are the parameter settings in the DRG algorithm.

Table 3. The parameters of the GA(FS), GA(IS), GA(FS+IS).

Algorithm	NP	T	a	Cross_L	Cross_H	Variation_Ratio	Variation_Num
GA(FS)	36	50	one-sixth	0.5	1	0.5	The total number of features /100
GA(IS)	30	50	one-sixth	0.5	1	0.5	The total number of instances /100
GA(FS+IS)	36	50	one-sixth	0.5	1	0.5	__

4.3. Evaluation Metrics

We use precision, recall, the F-measure and Area Under the Curve (AUC) as our evaluation metrics. These metrics are commonly used measures for evaluating classification performance [24,50]. They can be derived from the confusion matrix, which captures all four possible classification results, as presented in Table 4. The number of true positives (TP) is the number of low impact bug reports that are correctly divided into low impact bug reports. The number of false positives (FP) is the number of high impact bug reports that are incorrectly divided into low impact bug reports. The number of false negatives (FN) is the number of low impact bug reports that are incorrectly divided into high impact bug reports. The number of true negatives (TN) is the number of high impact bug reports that are correctly divided into high impact bug reports. Where, the sum of TP and TN is the number of correctly classified bug reports and the sum of FP and FN is the number of correctly classified bug reports. Based on the values of TP, FP, FN, and TN, the precision, recall, F-measure and AUC are calculated as follows.

Table 4. Confusion matrix, which can be used to calculate many evaluation metrics.

Confusion Matrix		Actual	
		Low Impact	**High Impact**
Predicted	low impact	TP:true positives	FP:false positives
	high impact	FN:false negatives	TN:true negatives

Accuracy: The accuracy of the model is the number of correct classifications divided by the total number of classifications. The accuracy is defined as follows:

$$Accuracy = \frac{TP + TN}{TP + FP + TN + FN} \times 100\% \tag{5}$$

Precision: The percentage of bug reports that are predicted to be either nonhighimpact or highimpact and are correctly predicted. We thus consider a separate precision for each severity. For a bug report severity of either nonhighimpact or highimpact, we define the precision more formally as follows:

$$Precision_{low} = \frac{TP}{TP + FP} \times 100\% \tag{6}$$

$$Precision_{high} = \frac{TN}{TN + FN} \times 100\% \tag{7}$$

Recall: The percentage of all bug reports that are actually nonhighimpact or highimpact and are correctly predicted to be nonhighimpact or highimpact, respectively. As for the precision, we also consider a separate recall for each severity. For a bug report severity of either nonhighimpact or highimpact, we define the recall more formally as follows:

$$Recall_{low} = \frac{TP}{TP + FN} \times 100\% \tag{8}$$

$$Recall_{high} = \frac{TN}{TN + FP} \times 100\% \tag{9}$$

F-measure: Usually, the precision and recall are not discussed in isolation. Instead, either the values of one measure are compared for a fixed value of the other measure or both are combined into a single measure, such as the *F-measure*, which is the weighted harmonic mean of the *recall* and *precision*. The *F-measure* has the property that if either the *recall* or *precision* is low, the *F-measure* also decreases. Therefore, the *F-measure* can be used as an effective evaluation criterion for the classification of imbalanced datasets.γ represents the weighting parameters; if γ is greater than 1, the (*precision*) is more important than *recall*, whereas if it is less than 1, the *recall* is a more important one. In this case study, we set γ equal to 1 to define the *F-measure* such that the *recall* and (*precision*) are equally weighted.

$$F - \text{measure}_{low} = \frac{\left(1 + \gamma^2\right) \times Precision_{low} \times Recall_{low}}{\gamma^2 \times Recall_{low} + Precision_{low}} \times 100\% \tag{10}$$

$$F - \text{measure}_{high} = \frac{\left(1 + \gamma^2\right) \times Precision_{high} \times Recall_{high}}{\gamma^2 \times Recall_{high} + Precision_{high}} \times 100\% \tag{11}$$

To comprehensively consider the total *F-measure* for a bug repository dataset, we combine $F - \text{measure}_{low}$ and $F - \text{measuree}_{high}$ into a single measure *F-measure*, which is defined as follows, where α and β represent weighting parameters such that $\alpha = \frac{TP+FN}{TP+FP+TN+FN}$ and $\beta = \frac{TN+FP}{TP+FP+TN+FN}$:

$$Precision = \alpha \times Precision_{low} + \beta \times Precision_{high} \tag{12}$$

$$Recall = \alpha \times Recall_{low} + \beta \times Recall_{high} \tag{13}$$

$$F-\text{measure} = \alpha \times F-\text{measure}_{low} + \beta \times F-\text{measure}_{high} \quad (14)$$

AUC(Area Under Curve): The ROC curve is commonly used for classifier evaluation. However, comparing curves visually can be cumbersome, especially when the curves are close together. Therefore, the AUC is calculated to serve as a single number expressing the accuracy. The AUC is the area of a two-dimensional graph in which $Recall_{low}$ or $Recall_{high}$ is plotted on the Y axis and FPR_{low} or FPR_{high} is plotted on the X axis over a distinct threshold T of possibility values. If the AUC is close to 0.5, then the classifier is practically random, whereas a value close to 1.0 means that the classifier makes practically perfect predictions. This value enables more rational discussions when comparing the accuracies of different classifiers [51]:

$$AUC_{low} = \int_{\infty}^{-\infty} Recall_{low}(T)\, FPR'_{low}(T)\, dT \quad (15)$$

$$AUC_{high} = \int_{\infty}^{-\infty} Recall_{high}(T)\, FPR'_{high}(T)\, dT \quad (16)$$

where

$$FPR'_{low} = \frac{FP}{FP+TN} \times 100\% \quad (17)$$

$$FPR'_{high} = \frac{FN}{FN+TP} \times 100\% \quad (18)$$

To comprehensively consider the total AUC for a bug repository dataset, we combine AUC_{low} and AUC_{high} into a single measure AUC, which is defined as follows, where α and β represent weighting parameters such that $\alpha = \frac{TP+FN}{TP+FP+TN+FN}$ and $\beta = \frac{TN+FP}{TP+FP+TN+FN}$:

$$AUC = \alpha \times AUC_{low} + \beta \times AUC_{high} \quad (19)$$

5. Experimental Results

In this section, the experimental results are discussed in relation to the specific research questions.

5.1. RQ1:What Are the Better Classification Results for the Reduced Datasets and the Unreduced Datasets?

In the first research question, we want to investigate which of the reduced datasets and the unreduced datasets have better classification performance for Mozilla, GCC and Eclipse projects.

To answer this question, first, we use NB, NBM, KNN, and SVM to classify the original datasets and record the experimental results. Then, we use the genetic algorithm for the feature selection for the original datasets and set the iteration to 50 times. After finding the optimal individual GA(FS), we use the basic filter feature selection method, i.e., IG, CHI, OneR, or Relief, to reduce the datasets to the same scale as the genetic algorithm and the statistical results. Finally, we use the genetic algorithm for the instance selection for the original datasets and set the iteration to 50 times. After finding the optimal individual GA(IS), we use the basic instance selection method, i.e., CNN, ENN, MCS, or ICF, to reduce the datasets to the same scale as the genetic algorithm and the statistical experiment result. We use four evaluation metrics mentioned above (accuracy, precision, recall, F-measure and AUC) to analyze the experimental results. Tables 5–7 present the performance of the feature selection approaches and instance selection approaches with four classifiers on Mozilla, GCC, and Eclipse, respectively.

Table 5. The results of feature selection and instance selection in the Mozilla dataset compared to the results of the raw Mozilla data.

Datasets	Classifiers	Evalution Metrics	Origin	Feature Reduction					Instance Reduction				
				GA(FS)	IG	CHI	OneR	Relief	GA(IS)	CNN	ENN	MCS	ICF
Mozilla	NB	Accuracy	0.6882	0.6694	0.6957	0.6883	0.6469	0.6531	0.7085	0.6568	0.6863	0.7048	0.6863
		Fmeasure	0.6883	0.6684	0.6915	0.6836	0.6422	0.6398	0.7097	0.6568	0.6873	0.7056	0.6871
		Precision	0.711	0.669	0.7491	0.7426	0.6497	0.7412	0.7165	0.6791	0.701	0.7215	0.7037
		Recall	0.6882	0.6694	0.6957	0.6883	0.6469	0.6531	0.7085	0.6568	0.6863	0.7048	0.6863
		AUC	0.755	0.7082	0.8279	0.8298	0.656	0.7933	0.7754	0.7414	0.7489	0.7897	0.7514
	NBM	Accuracy	0.7565	0.7444	0.7607	0.7681	0.649	0.7087	**0.8026**	0.7454	0.7306	0.7583	0.7712
		Fmeasure	0.755	0.7439	0.7615	0.7689	0.6476	0.7069	**0.803**	0.7436	0.7276	0.7571	0.7703
		Precision	0.7969	0.7523	0.7772	0.7847	0.6559	0.749	**0.8238**	0.7867	0.7794	0.7963	0.8076
		Recall	0.7565	0.7444	0.7607	0.7681	0.649	0.7087	**0.8026**	0.7454	0.7306	0.7583	0.7712
		AUC	0.8249	0.8082	0.8447	0.851	0.732	0.7985	**0.8545**	0.8006	0.8067	0.8409	0.8271
	KNN	Accuracy	0.6476	0.6471	0.7032	0.7013	0.649	0.6735	0.6494	0.6107	0.6162	0.6199	0.631
		Fmeasure	0.6473	0.6472	0.7041	0.7026	0.6483	0.6746	0.6509	0.596	0.6178	0.6041	0.6321
		Precision	0.671	0.6476	0.7204	0.7129	0.6486	0.6894	0.6587	0.6809	0.6243	0.6979	0.6343
		Recall	0.6476	0.6471	0.7032	0.7013	0.649	0.6735	0.6494	0.6107	0.6162	0.6199	0.631
		AUC	0.6828	0.6487	0.7177	0.7264	0.6594	0.7021	0.6861	0.6656	0.5918	0.6181	0.6203
	SVM	Accuracy	0.7804	0.714	0.8145	0.8219	0.7041	0.7718	0.7583	0.7472	0.7565	0.7694	0.786
		Fmeasure	0.7814	0.7093	0.8153	0.8227	0.6998	0.7724	0.7592	0.7483	0.757	0.7697	0.7868
		Precision	0.7881	0.7206	0.827	0.8326	0.7102	0.7917	0.7634	0.7536	0.7584	0.7911	0.7919
		Recall	0.7804	0.714	0.8145	0.8219	0.7041	0.7718	0.7583	0.7472	0.7565	0.7694	0.786
		AUC	0.7847	0.708	0.8223	0.8288	0.6995	0.7826	0.7603	0.75	0.7552	0.7807	0.7891

Table 6. The results of feature selection and instance selection in the GCC dataset compared to the results of the raw GCC data.

Datasets	Classifiers	Evalution Metrics	Origin	Feature Reduction					Instance Reduction				
				GA(FS)	IG	CHI	OneR	Relief	GA(IS)	CNN	ENN	MCS	ICF
GCC	NB	Accuracy	0.6273	0.5883	0.5748	0.5748	0.5066	0.5527	0.6509	0.6436	0.6255	0.6273	0.6545
		Fmeasure	0.6592	0.6227	0.6096	0.6096	0.5402	0.5853	0.6798	0.6739	0.6569	0.6591	0.6818
		Precision	0.7705	0.7689	0.7709	0.7709	0.7703	0.7749	0.7559	0.7643	0.7348	0.7778	0.7449
		Recall	0.6273	0.5883	0.5748	0.5748	0.5066	0.5527	0.6509	0.6436	0.6255	0.6273	0.6545
		AUC	0.7337	0.7482	0.7562	0.7561	0.7214	0.7431	0.6977	0.7321	0.6692	0.7393	0.6818
	NBM	Accuracy	0.7582	**0.7814**	0.7755	0.7755	0.7865	0.7291	0.76	0.7673	0.7782	0.7255	0.7636
		Fmeasure	0.7719	**0.7867**	0.7796	0.7796	0.7746	0.7419	0.7751	0.7803	0.79	0.7465	0.7766
		Precision	0.7994	**0.7937**	0.7846	0.7846	0.7874	0.7624	0.809	0.8069	0.8136	0.8032	0.8019
		Recall	0.7582	**0.7814**	0.7755	0.7755	0.7685	0.7291	0.76	0.7673	0.7782	0.7255	0.7376
		AUC	0.7769	**0.8**	0.7928	0.7928	0.7703	0.7374	0.7872	0.7811	0.7834	0.7532	0.7702
	KNN	Accuracy	0.7109	0.7577	0.7536	0.7536	0.7647	0.8018	0.7218	0.7091	0.7545	0.6291	0.7382
		Fmeasure	0.7059	0.7581	0.7533	0.7533	0.7516	0.7977	0.7097	0.697	0.7213	0.6604	0.7294
		Precision	0.7013	0.7585	0.7529	0.7529	0.7431	0.7946	0.7004	0.6874	0.7083	0.7937	0.7225
		Recall	0.7109	0.7577	0.7536	0.7536	0.7647	0.8018	0.7218	0.7091	0.7545	0.6291	0.7382
		AUC	0.5625	0.6511	0.6417	0.6417	0.5947	0.6955	0.5576	0.5406	0.5519	0.6972	0.5905
	SVM	Accuracy	0.8091	0.8033	0.8047	0.8029	0.797	0.8036	0.7818	0.7745	0.7982	0.7364	0.8087
		Fmeasure	0.803	0.7591	0.7611	0.7597	0.7477	0.749	0.7753	0.7833	0.7918	0.7544	0.8008
		Precision	0.7993	0.7809	0.7824	0.7782	0.7594	0.7957	0.7707	0.7975	0.7876	0.7956	0.7969
		Recall	0.8091	0.8033	0.8047	0.8029	0.797	0.8036	0.7818	0.7745	0.7982	0.7364	0.8073
		AUC	0.6937	0.586	0.5869	0.5857	0.561	0.5715	0.6554	0.719	0.6778	0.7212	0.6896

Table 7. The results of feature selection and instance selection in the Eclipse dataset compared to the results of the raw Eclipse data.

Datasets	Classifiers	Evalution Metrics	Origin	Feature Reduction					Instance Reduction				
				GA(FS)	IG	CHI	OneR	Relief	GA(IS)	CNN	ENN	MCS	ICF
Eclipse	NB	Accuracy	0.5911	0.5503	0.5679	0.5685	0.4684	0.4905	0.6055	0.587	0.5952	0.6089	0.5891
		Fmeasure	0.6006	0.5486	0.5689	0.5689	0.4257	0.4714	0.6156	0.5954	0.6051	0.619	0.5989
		Precision	0.6865	0.7102	0.714	0.7161	0.7037	0.695	0.6928	0.691	0.6865	0.6954	0.6817
		Recall	0.5911	0.5503	0.5679	0.5685	0.4684	0.4905	0.6055	0.587	0.5952	0.6089	0.5891
		AUC	0.6931	0.7129	0.7173	0.7177	0.7251	0.6829	0.6972	0.6857	0.6908	0.7006	0.6772
	NBM	Accuracy	0.5549	0.5745	0.5768	0.5788	**0.6399**	0.5693	0.5823	0.5597	0.5481	0.5427	0.5611
		Fmeasure	0.5486	0.5776	0.5783	0.5804	**0.6467**	0.5718	0.5822	0.5528	0.5387	0.5323	0.5571
		Precision	0.7331	0.7184	0.7225	0.7221	**0.7348**	0.7157	0.737	0.743	0.738	0.7342	0.7298
		Recall	0.5549	0.5745	0.5768	0.5788	**0.6399**	0.5693	0.5823	0.5597	0.5481	0.5427	0.5611
		AUC	0.723	0.7295	0.7307	0.7331	**0.7707**	0.705	0.7351	0.7218	0.5125	0.7202	0.7109
	KNN	Accuracy	0.6355	0.6848	0.7318	0.7212	0.7903	0.6847	0.6608	0.5782	0.6205	0.6198	0.5857
		Fmeasure	0.6437	0.6895	0.732	0.7215	0.7009	0.6851	0.6614	0.59	0.6315	0.6312	0.5981
		Precision	0.6599	0.6969	0.7323	0.7217	0.6986	0.6854	0.662	0.6177	0.6636	0.6711	0.6384
		Recall	0.6355	0.6848	0.7318	0.7212	0.7093	0.6847	0.6608	0.5782	0.6205	0.6198	0.5857
		AUC	0.6183	0.6617	0.7077	0.6906	0.6628	0.6531	0.6274	0.5688	0.6199	0.6293	0.5923
	SVM	Accuracy	0.759	0.7545	0.7558	0.7568	0.721	0.7164	0.7352	0.729	0.7502	0.714	0.7447
		Fmeasure	0.7498	0.7199	0.7211	0.727	0.6859	0.6546	0.7275	0.7327	0.74	0.7226	0.7372
		Precision	0.7504	0.7624	0.7669	0.7614	0.7146	0.723	0.7256	0.739	0.7405	0.7637	0.7357
		Recall	0.759	0.7545	0.7558	0.7568	0.721	0.7164	0.7352	0.729	0.7502	0.714	0.7447
		AUC	0.6978	0.6466	0.6492	0.6574	0.6219	0.5804	0.6768	0.7092	0.6864	0.7351	0.6871

From Tables 5–7, it can be found that the ability to identifying high-impact bug reports by using feature selection and instance selection to select datasets after reduction works much better than the original experimental datasets. For example, from Table 5, it can be found that for the Mozilla dataset, the NBM classifier-based instance reduction method GA(IS) works best for identifying high-impact bug reports. It achieves the largest AUC value, which is 0.8545, and the accuracy, F-measure, precision, and recall values are 0.8026, 0.803, 0.8238, and 0.8026, respectively. Compared with the original data classification without data reduction, the F-measure value results have the greatest improvement. For the Mozilla dataset after attribute reduction with CHI, the SVM classifier has the best classification performance for identifying high-impact bug reports, and the accuracy, F-measure, precision, recall, and AUC values are 0.8219, 0.8227, 0.8326, 0.8219, 0.8288, respectively, and achieve the maximum accuracy, F-measure, precision and recall values.

From Table 6, it can be found that for the GCC dataset, the GA(FS) method based on the NBM classifier has the greatest improvement in the accuracy rate, recall rate and the AUC value. Among them, the accuracy of the GA(FS) method based on the NBM classifier for identifying high-impact bug reports is 0.7814, the recall rate is 0.7814, and the AUC value is 0.8. From Table 7, for the Eclipse dataset, using the OneR reduction method based on the NBR classifier to identify high-impact bug reports has the greatest improvement, especially for the accuracy, F-measure, recall and AUC values; the classification results are 0.6399, 0.6467, 0.6399, 0.7707, respective.

Therefore, according to the experimental results, the reduced dataset has better classification performance than the unreduced dataset.

5.2. RQ2: How Does the Order of the Feature Selection and Instance Selection on the Datasets Impact the Experimental Performance?

In this research question, we want to investigate whether the order of the feature selection and instance selection on the Mozilla, GCC and Eclipse datasets has an impact on the experimental results. We consider four feature selection methods, i.e., IG, CHI, OneR, and Relief, and four instance selection methods, i.e., CNN, ENN, MCS, and ICF. We use the five evaluation metrics mentioned above (accuracy, precision, recall, F-measure and AUC) to analyze the experimental results.

To answer this question, first, we use the genetic algorithm to simultaneously perform the approximate reduction of features and instances on the original dataset, i.e., GA(FS+IS), and record the experimental results. Thus, we use genetic algorithms to apply the feature selection for the datasets. Based on the obtained optimal individuals, we use the genetic algorithm to apply the instance selection, i.e., GA(FS_IS), and the experimental results are recorded. Next, we use genetic algorithms to apply the instance selection for the datasets. Based on the obtained optimal individuals, we use the genetic algorithm to apply the feature selection, i.e., GA(IS_FS), and record the experimental results. Finally, according to the first experiment, among the NB, NBM, KNN, and SVM classifiers, NBM has the best classification effect, so we only retain the experimental results of using NBM as a classifier in Table 8.

Table 8 shows that the best classification performance is obtained after instance reduction using NBM as a classifier to identify high-impact bug reports for Mozilla dataset. The accuracy, F-measure, and recall values show the most obvious improvements, with values of 0.8026, 0.803, and 0.8026, respectively. For the GCC dataset, the GA(FS_IS) method (this is, feature selection is first performed on the dataset, and then the instance selection of the dataset is performed) achieves the best classification performance in identifying high-impact bug reports. In the accuracy value and the recall value, there is the most obvious improvement: the accuracy and the recall values are 0.796 and 0.796, respectively. For the Eclipse dataset, the GA(IS + FS) method (that is, feature selection and instance selection are performed simultaneously) performs better to identify high-impact bug reports. The most obvious improvement is obtained for the accuracy, F-measure and recall values, which are 0.6907, 0.6983 and 0.6907, respectively.

Therefore, we found that the order of the feature selection and instance selection has an impact on identifying high-impact bug reports through the experiments on the Mozilla, GCC and Eclipse datasets.

Table 8. The effect of the feature selection and instance selection order on the experimental results for the Mozilla and GCC datasets when using NBM as a classifier.

Datasets	Evaluation Metrics	Origin	GA(FS+IS)	GA(FS_IS)	GA(IS_FS)	FS	IS
Mozilla	Accuracy	0.7565	0.7107	0.7505	0.7357	0.7681	**0.8026**
	Fmeasure	0.755	0.7106	0.7506	0.7355	0.7689	**0.803**
	Precision	0.7969	0.7135	0.7526	0.7627	0.7847	**0.8238**
	Recall	0.7565	0.7107	0.7505	0.7357	0.7681	**0.8026**
	AUC	0.8249	0.7659	0.8095	0.8125	0.851	**0.8545**
GCC	Accuracy	0.7582	0.785	**0.796**	0.7887	0.7755	0.7782
	Fmeasure	0.7719	0.7885	**0.796**	0.7905	0.7796	0.79
	Precision	0.7994	0.7926	**0.796**	0.7926	0.7846	0.8136
	Recall	0.7582	0.785	**0.796**	0.7887	0.7755	0.7782
	AUC	0.7769	0.7848	**0.8149**	0.8019	0.7928	0.7834
Eclipse	Accuracy	0.5549	**0.6907**	0.5876	0.5942	0.6399	0.5427
	Fmeasure	0.5486	**0.6983**	0.5911	0.6011	0.6467	0.5323
	Precision	0.7331	**0.738**	0.7282	0.7088	0.7348	0.7342
	Recall	0.5549	**0.6907**	0.5876	0.5942	0.6399	0.5427
	AUC	0.723	**0.7778**	0.7357	0.7242	0.7707	0.7202

5.3. RQ3: What Is the Effect of the Balance Processing and Reduction Denoising Order on the Experimental Results?

In this research question, we want to investigate the effect of the balance processing and reduction denoising order on the experimental results for the Mozilla, GCC and Eclipse datasets.

To answer this question, we first use several different imbalanced processing strategies, i.e., RUS, ROS, SMOTE and CMA, to balance the preprocessed datasets after performing feature selection and instance selection and record the experimental results as the GA(FS+IS)_imbalance. After we balance the original dataset, we simultaneously perform feature selection and instance selection and record the results as imbalance_GA$(\mathrm{FS}+\mathrm{IS})$. Second, we perform feature selection on the original datasets and then use the different imbalanced processing strategies, RUS, ROS, SMOTE and CMA, to balance the datasets. The experimental results are recorded as the GA(FS)_imbalance. We perform the feature selection after the raw datasets are balanced by the different imbalanced processing strategies, RUS, ROS, SMOTE and CMA, and the experimental results are recorded as imbalance _GA(FS). Third, we perform instance selection on the original datasets and then use the different imbalanced processing strategies, RUS, ROS, SMOTE and CMA, to balance the datasets. The experimental results are recorded as the GA(IS)_imbalance. We perform the feature selection after the raw datasets are balanced by the different imbalanced processing strategies, RUS, ROS, SMOTE and CMA, and the experimental results are recorded as imbalance_GA(IS). Fourth, we perform feature selection on the original datasets first, then perform instance selection, and then use the different imbalanced processing strategies, RUS, ROS, SMOTE and CMA, for balance processing; the experimental results are recorded as the GA(FS_IS)_imbalance. We first balance the original dataset with the RUS, ROS, SMOTE and CMA method, then perform feature selection and instance selection; we record the experimental results as imbalance_GA(FS_IS). Finally, we perform instance selection on the original datasets first and then perform feature selection; subsequently, we use the different imbalanced processing strategies, RUS, ROS, SMOTE and CMA, for balance processing, and the experimental results are recorded as the GA(IS_FS)_imbalance. We first balance the original dataset with the RUS, ROS, SMOTE and CMA methods then perform instance selection and feature selection; the experimental results are subsequently recorded as imbalance_GA(IS_FS). We use the five evaluation metrics mentioned above (accuracy, precision, recall, F-measure and AUC) to make comparisons and only retain the experimental results of NBM as a classifier. Tables 9–13 shows the performance of the processing sequence for balance processing and reduction denoising to identify high-impact bug reports for Mozilla, GCC and Eclipse.

From Table 9, we find that the imbalance_GA(FS + IS) method (that is, first using the SMOTE method for unbalanced data sets and then using feature selection and instance selection at the same time) can achieve the maximum performance improvement for identifying high-impact bug reports for the Mozilla dataset. The accuracy, F-measure, precision, recall and AUC values are 0.7435, 0.7444, 0.7615, 0.7435, and 0.8322, respectively. For the GCC and Eclipse datasets, it can be found that the GA(FS+IS_imbalance method (that is, first using the feature selection and instance selection for the dataset, then using the ROS method for imbalance processing) performs better in identifying high-impact bug reports. For the GCC dataset, the accuracy, the F-measure, the precision, the recall, and AUC values are 0.7562, 0.7725 0.805, 0.7562, and 0.7853, respectively. For the Eclipse dataset, the accuracy, F-measure, and recall values are 0.6251, 0.6303, 0.7207, 0.6251, and 0.7761, respectively.

From Table 10, we can observe that the imbalance_GA(FS) method (that is, using the SMOTE method to imbalance the dataset first, and then only using feature selection) has the greatest improvement for identifying high-impact bug reports for the Mozilla dataset. The accuracy, F-measure, precision, recall and AUC values are 0.762, 0.7616, 0.7926, 0.762, and 0846, respectively. For the GCC dataset, the GA(FS)_imbalance method (that is, the dataset is processed by feature selection first, and then the dataset is processed using the CMA method) performs better in identifying high-impact bug reports. The accuracy, F-measure, precision, recall and AUC values are 0.7778, 0.7878, 0.8054, 0.7778, and 0.799, respectively. For the Eclipse dataset, the GA(FS)_imbalance method (that is, feature selection is used first, and then the ROS method is used to imbalance the dataset) performs better in identifying high-impact bug reports. The accuracy, F-measure, precision, recall and AUC values are 0.5697, 0.5686, 0.7319, 0.5697, and 0.734, respectively.

From Table 11, For the Mozilla and GCC datasets, the GA(IS)_imbalance method (that is, dataset is processed by feature selection first and then the CMA method is used to imbalance the data set) performs better in identifying high-impact bug reports. For the Mozilla dataset, the accuracy, the F-measure, the precision, the recall, and AUC values are 0.8044, 0.8048, 0.8251, 0.8044, and 0.8547, respectively. For the GCC dataset, the accuracy, F-measure, the precision, recall and AUC values are 0.7855, 0.7981, 0.827, 0.7855, and 0.8052, respectively. For the Eclipse dataset, the GA(IS)_imbalance method (that is, the instance selection of the dataset is first performed, and then the SMOTE method is used to imbalance the dataset) performs better for identifying high-impact bug reports. The accuracy, F-measure, precision, recall and AUC values are 0.6061, 0.6092, 0.7465, 0.6061 and 0.7475, respectively.

From Table 12, we can find that the imbalance_GA(FS_IS) method (that is, first the SMOTE method is used to imbalance the dataset, then feature selection is used to reduce it, and then the instance selection is also used to reduce it) achieves the maximum performance improvement for identifying high-impact bug reports for the Mozilla dataset. The accuracy, F-measure, precision, recall and AUC values are 0.7952, 0.796, 0.8078, 0.7952, and 0.8633, respectively. For the GCC dataset, we find that the imbalance_GA(FS_IS) method (that is, first the CMA method is used for the imbalanced dataset, then feature selection is used for reduction, and then use instance selection is used for reduction) can achieve the maximum performance improvement for identifying high-impact bug reports and the accuracy, F-measure, precision, recall, and AUC values are 0.7887, 0.797, 0.8109, 0.7887, and 0.8108, respectively. For the Eclipse dataset, we find that the imbalance_GA(FS_IS) method (that is, first imbalance the dataset using the ROS method, then use the feature selection to reduce, and then use the instance selection to reduce) could achieve the maximum performance improvement for identifying high-impact bug reports, and the accuracy, F-measure, precision, recall, and AUC values are 0.5779, 0.5779, 0.7368, 0.5779, and 0.7417, respectively.

From Table 13, we find that the GA(IS_FS)_imbalance method (that is, first instance selection is used, then feature selection is performed, and then the imbalanced learning strategy is used to imbalance the dataset) can achieve the best classification effect. For the Mozilla and Eclipse datasets, the accuracy, F-measure, precision, recall, and AUC values after the imbalance processing using the ROS method are 0.7724, 0.7734, 0.7889, 0.7724, and 0.8303 and 0.5928, 0.5961, 0.7307, 0.5928, and 0.7342, respectively. For the GCC datasets, the accuracy, F-measure, precision, recall and AUC

values after the imbalanced processing using the SMOTE method are 0.7978, 0.8048, 0.8158, 0.7978, and 0.7973, respectively.

Therefore, we found that the imbalance_GA(FS_IS) method (that is, first use the ROS method to imbalance the dataset, then use the feature selection to reduce, and then use the instance selection to reduce) achieved the highest AUC value of 0.86333 for identifying high-impact bug reports for the Mozilla dataset, and using the CMA method for processing datasets after instance reduction achieved the highest precision value of 0.827 for identifying high-impact bug reports for the GCC dataset. For the Eclipse dataset, using the ROS method for processing datasets after FS and IS reduction simultaneously achieved the highest AUC value of 0.7761 to identify high-impact bug reports.

Table 9. The results of GA(FS+IS)_imbalance and imbalance_GA(FS+IS) on the Mozilla, GCC and Eclipse datasets.

Datasets	Evaluation Metrics	GA(FS+IS)_Imbalance				Imbalance_GA(FS+IS)		
		RUS	ROS	CMA	SMOTE	RUS	ROS	SMOTE
Mozilla	Accuracy	0.6983	0.6963	0.7025	0.6921	0.7223	0.7327	**0.7435**
	Fmeasure	0.6965	0.6952	0.7017	0.6909	0.723	0.7326	**0.7444**
	Precision	0.7086	0.7033	0.7087	0.6997	0.7462	0.7618	**0.7615**
	Recall	0.6983	0.6963	0.7025	0.6921	0.7223	0.7327	**0.7435**
	AUC	0.7701	0.771	0.7665	0.7679	0.8056	0.8164	**0.8322**
GCC	Accuracy	0.7102	**0.7562**	0.7543	0.7466	0.7286	0.7091	0.7164
	Fmeasure	0.7348	**0.7725**	0.7705	0.7648	0.7476	0.7313	0.739
	Precision	0.7941	**0.805**	0.8022	0.8031	0.7909	0.7871	0.8041
	Recall	0.7102	**0.7562**	0.7543	0.7466	0.7286	0.7091	0.7164
	AUC	0.7734	**0.7853**	0.786	0.7855	0.7805	0.7651	0.7691
Eclipse	Accuracy	0.6136	**0.6251**	0.6218	0.621	0.5114	0.4859	0.4894
	Fmeasure	0.6162	**0.6303**	0.6263	0.6259	0.4884	0.4541	0.4602
	Precision	0.7272	**0.7207**	0.7229	0.7186	0.7375	0.7237	0.7204
	Recall	0.6136	**0.6251**	0.6218	0.621	0.5114	0.4859	0.4894
	AUC	0.771	**0.7761**	0.7764	0.7746	0.6762	0.6842	0.6728

Table 10. The results of GA(FS)_imbalance and imbalance_GA(FS) on the Mozilla, GCC and Eclipse datasets.

Datasets	Evaluation Metrics	GA(FS)_Imbalance				Imbalance_GA(FS)		
		RUS	ROS	CMA	SMOTE	RUS	ROS	SMOTE
Mozilla	Accuracy	0.7444	0.7444	0.7404	0.7404	0.7505	0.7597	**0.762**
	Fmeasure	0.7428	0.7438	0.7392	0.7397	0.7503	0.7593	**0.7616**
	Precision	0.7594	0.7531	0.7523	0.7489	0.7781	0.7912	**0.7926**
	Recall	0.7444	0.7444	0.7404	0.7404	0.7505	0.7597	**0.762**
	AUC	0.8139	0.8113	0.809	0.8098	0.834	0.831	**0.846**
GCC	Accuracy	0.7541	0.7741	**0.7778**	0.776	0.74	0.7455	0.7273
	Fmeasure	0.7686	0.7843	**0.7878**	0.7878	0.7583	0.7608	0.7466
	Precision	0.7978	0.802	**0.8054**	0.811	0.8027	0.7921	0.7921
	Recall	0.7541	0.7741	**0.7778**	0.776	0.74	0.7455	0.7273
	AUC	0.7929	0.798	**0.799**	0.7959	0.7651	0.7843	0.7645
Eclipse	Accuracy	0.5462	**0.5697**	0.5628	0.5503	0.5503	0.5684	0.5512
	Fmeasure	0.5394	**0.5686**	0.5601	0.5459	0.5423	0.5625	0.542
	Precision	0.7285	**0.7319**	0.7311	0.7229	0.7407	0.7485	0.7399
	Recall	0.5462	**0.5697**	0.5628	0.5503	0.5503	0.5684	0.5512
	AUC	0.7359	**0.734**	0.733	0.7239	0.7111	0.7365	0.7191

Table 11. The results of GA(IS)_imbalance and imbalance_GA(IS) on the Mozilla, GCC and Eclipse datasets.

Datasets	Evaluation Metrics	GA(IS)_Imbalance				Imbalance_GA(IS)		
		RUS	ROS	CMA	SMOTE	RUS	ROS	SMOTE
Mozilla	Accuracy	0.797	0.797	**0.8044**	0.8044	0.7841	0.7915	0.7952
	Fmeasure	0.7973	0.7974	**0.8048**	0.8049	0.7845	0.7917	0.7955
	Precision	0.8201	0.8188	**0.8251**	0.8239	0.8062	0.815	0.8175
	Recall	0.797	0.797	**0.8044**	0.8044	0.7841	0.7915	0.7952
	AUC	0.8556	0.8543	**0.8547**	0.8538	0.8524	0.8527	0.8533
GCC	Accuracy	0.7164	0.7818	**0.7855**	0.78	0.7309	0.7564	0.7436
	Fmeasure	0.7394	0.7944	**0.7981**	0.7919	0.752	0.7743	0.7629
	Precision	0.8105	0.822	**0.827**	0.8161	0.8133	0.8245	0.8158
	Recall	0.7164	0.7818	**0.7855**	0.78	0.7309	0.7564	0.7436
	AUC	0.7541	0.8064	**0.8052**	0.808	0.7904	0.8032	0.8069
Eclipse	Accuracy	0.5474	0.5966	0.5891	**0.6061**	0.5201	0.5468	0.5276
	Fmeasure	0.537	0.598	0.5896	**0.6092**	0.502	0.5348	0.5113
	Precision	0.7418	0.7455	0.7423	**0.7465**	0.7326	0.7488	0.7376
	Recall	0.5474	0.5966	0.5891	**0.6061**	0.5201	0.5468	0.5276
	AUC	0.7244	0.7457	0.742	**0.7475**	0.6931	0.7238	0.7085

Table 12. The results of GA(FS_IS)_imbalance and imbalance_GA(FS_IS) on the the Mozilla, GCC and Eclipse datasets.

Datasets	Evaluation Metrics	GA(FS_IS)_Imbalance				Imbalance_GA(FS_IS)		
		RUS	ROS	CMA	SMOTE	RUS	ROS	SMOTE
Mozilla	Accuracy	0.7383	0.7404	0.7505	0.7363	0.756	0.7579	**0.7952**
	Fmeasure	0.738	0.7401	0.7501	0.736	0.7559	0.7583	**0.796**
	Precision	0.7451	0.7462	0.7574	0.7427	0.7832	0.7795	**0.8078**
	Recall	0.7383	0.7404	0.7505	0.7363	0.756	0.7579	**0.7952**
	AUC	0.8042	0.811	0.8094	0.8107	0.8377	0.8402	**0.8633**
GCC	Accuracy	0.7668	0.7887	**0.7887**	0.7832	0.6927	0.7382	0.7218
	Fmeasure	0.7808	0.7948	**0.797**	0.7907	0.7185	0.7565	0.7433
	Precision	0.8103	0.8036	**0.8109**	0.802	0.8028	0.8001	0.8019
	Recall	0.7668	0.7887	**0.7887**	0.7832	0.6927	0.7382	0.7218
	AUC	0.8019	0.81	**0.8108**	0.8071	0.7621	0.7774	0.7861
Eclipse	Accuracy	0.5483	**0.5779**	0.5766	0.5717	0.5055	0.526	0.5196
	Fmeasure	0.5414	**0.5779**	0.5766	0.5708	0.4836	0.5104	0.4959
	Precision	0.7321	**0.7368**	0.735	0.734	0.7237	0.7304	0.7553
	Recall	0.5483	**0.5779**	0.5766	0.5717	0.5055	0.526	0.5196
	AUC	0.7256	**0.7417**	0.7404	0.7327	0.6645	0.7063	0.717

Table 13. The results of GA(IS_FS)_imbalance and imbalance_GA(IS_FS) on the Mozilla, GCC and Eclipse datasets.

Datasets	Evaluation Metrics	GA(IS_FS)_imbalance				imbalance_GA(IS_FS)		
		RUS	ROS	CMA	SMOTE	RUS	ROS	SMOTE
Mozilla	Accuracy	0.7431	0.7357	0.732	0.7264	0.7188	0.7724	**0.7532**
	Fmeasure	0.7431	0.7354	0.7317	0.7258	0.7201	0.7734	**0.7544**
	Precision	0.7691	0.764	0.7602	0.7577	0.725	0.7889	**0.7642**
	Recall	0.7431	0.7357	0.732	0.7264	0.7188	0.7724	**0.7532**
	AUC	0.8128	0.8108	0.8123	0.8106	0.7765	0.8303	**0.8403**
GCC	Accuracy	0.7723	0.7923	0.7942	**0.7978**	0.725	0.7327	0.6945
	Fmeasure	0.7867	0.8001	0.8021	**0.8048**	0.7476	0.7541	0.7199
	Precision	0.8197	0.8128	0.8153	**0.8158**	0.8202	0.8203	0.7989
	Recall	0.7723	0.7923	0.7942	**0.7978**	0.725	0.7327	0.6945
	AUC	0.785	0.7964	0.7969	**0.7973**	0.7805	0.7932	0.7722
Eclipse	Accuracy	0.5514	**0.5928**	0.5887	0.5845	0.4966	0.5075	0.4863
	Fmeasure	0.5482	**0.5961**	0.5915	0.5877	0.4757	0.4853	0.46
	Precision	0.7141	**0.7307**	0.7287	0.7217	0.7076	0.7282	0.7038
	Recall	0.5514	**0.5928**	0.5887	0.5845	0.4966	0.5075	0.4863
	AUC	0.7097	**0.7342**	0.7308	0.7287	0.6097	0.6477	0.6112

6. Conclusions

In this paper, we propose a high-impact bug report identification approach generated by combining the data reduction, e.g., the Data Reduction based on Genetic algorithm (DRG), and imbalanced learning strategies. We use four feature selection algorithms (i.e., One Rule (OneR), information gain (IG), chi squared (CHI), and filtered selection (Relief)) to extract the important attributes, which aims to reduce the data from the original bug reports by removing noisy or noninformative words. Then, we use imbalanced processing technologies (i.e., random under-sampling (RUS), random over-sampling (ROS), synthetic minority over-sampling technique (SMOTE), and cost-matrix adjuster (CMA)) to reduce the imbalance of small-scale and high-quality training sets obtained after feature selection and instance selection. In our work, we not only reduced the word dimension of the original training set that improved the quality of training set, but we also improved the classification ability for identifying the high-impact bug reports with an imbalanced distribution. Comprehensive experiments have been conducted on public datasets obtained from real-world bug repositories, and the experimental results indicate that our approach can efficiently improve the ability to identify high-impact bug reports.

Author Contributions: Data curation, M.W., S.W. and T.L.; Formal analysis, H.L.; Methodology, S.G. and C.G.; Writing—review and editing, R.C.

Funding: This research was supported by the National Natural Science Foundation of China (grant number. 61902050, 61672122,61602077,61771087,51879027,51579024, and 71831002), Program for Innovative Research Team in University of Ministry of Education of China (No. IRT 17R13), the Fundamental Research Funds for the Central Universities (Nos. 3132019501 and 3132019502,JLU), and CERNET Innovation Project (Nos. NGII20181203 and NGII20181205).

Conflicts of Interest: The authors declare no conflict of interest.

Appendix A

Appendix A.1. Classifiers

Many classifiers are used in text categorization. In this section, we mainly introduce the four classifiers used in our manuscript, which are as follows: naive Bayes(NB), naive Bayes multinomial (NBM), support vector machine (SVM) and K-nearest neighbors (KNN).

Appendix A.1.1. Naive Bayes

The theory of the naive Bayes classifier: For the given unclassified items, calculate the probability of each category occurring under the condition of the item occurring and which probability value is the largest; then, determine which category the classification item belongs to.

Suppose $x = \{a_1, a_2, \ldots, a_m\}$ is an unclassified item, and each a is a feature attribute of x; the category set is $C = \{y_1, y_2, \ldots, y_n\}$, and the probability that x belongs to each category is $P(y_1|x), P(y_2|x), \ldots, P(y_n|x)$.

If $P(y_k|x) = max\{P(y_1|x), P(y_2|x), \ldots, P(y_n|x)\}$, then $x \in y_k$.

Appendix A.1.2. Naive Bayes Multinomial

The polynomial naive Bayes implements the Bayesian algorithm when the discrete features obey the polynomial distribution. The polynomial distribution is parameterized into a vector $\theta_y = \{\theta_{y1}, y_{y2}, \ldots, y_{yn}\}$. For each class y, the parameter n represents the number of features, that is, the size of the word vector. θ_{yi} indicates that the probability of having $P(\mathrm{xi}|y)$ appear for a feature i in a sample classified as y . The parameter θ_y is calculated by the smoothed maximum likelihood estimation method, and the calculation formula is as follows: $\hat{\theta}_{yi} = \frac{N_{yi}+\alpha}{N_y+\alpha n}$, where N_{yi} is the number of occurrences of the feature i in the sample belonging to the y class on the training set T ,

and N_y is the sum of all the features in class y . The smoothing coefficient $\alpha \geq 0$ represents a feature that has not appeared in the training set. If $\alpha = 1$, it is called Laplace smoothing.

Appendix A.1.3. K-Nearest Neighbors

The K-nearest neighbor algorithm is used to input test data in the case where the data and tags in the training set are known to compare the features of the test data with the features corresponding to the training set and to find the top K data whose training set is most similar to the test data. Then, the category corresponding to the test data is the one with the most occurrences among the K data, and the description of the algorithm is as follows:

- Calculate the distance between the test data and each training data;
- Sort according to the increasing relationship of distances;
- Select K data points with the smallest distance;
- Determine the frequency of occurrence of the category of the top K data points;
- Return the category with the highest frequency among the top K data points as the prediction classification of the test data.

Appendix A.1.4. Support Vector Machine

SVM is a supervised learning method. It searches for a classification hyperplane in high-dimensional space and separates the sample points of different categories to maximize the interval between different types of points. The classification hyperplane is the classifier corresponding to the maximum interval hyperplane, called maximum interval classifiers; it can minimize empirical errors and maximize geometric edges.

Appendix A.2. Feature Selection Algorithm

In this section, we mainly introduce the four feature selection algorithms used in our manuscript, which are as follows: One Rule(OneR), Information Gain (IG), Chi-square (CHI) and Filtered Selection (Relief).

Appendix A.2.1. OneR

The basic idea of the OneR algorithm is to select a single attribute for a particular class to maximize the accuracy of that class. First, assign each value of each attribute to the class with the most occurrences of that value. Then, calculate the accuracy of the attribute value corresponding to the other classes Add all the accuracy values to obtain the total accuracy. The OneR algorithm selects those attributes that have the highest accuracy, as shown in Algorithm A1. C represents the category of the original training set, A represents all the attributes of the original training set (a is one of the attributes of A), and s represents the feature extraction range.

Appendix A.2.2. IG

The IG algorithm is often used to assess the quality of features in machine learning. It calculates the accuracy of the classification algorithm including an attribute and not including an attribute. The IG assigns a weight to the attribute based on the difference between the two values. In the IG algorithm, the measure of word importance is mainly to consider how much information the word can provide for the classification algorithm. The more information it provides, the more important the word is, as shown in Algorithm A2.

Algorithm A1 OneR

Input:

C, A, a, s

Output:

P, *the new feature selection set*

$P \leftarrow \emptyset$.
For each a in A
  For each v in a
    For each c in C
      $X \leftarrow numbers(a, v, e)$//calculates the number of v that appear in c.
    end for
  $C(a) \leftarrow findmax(X)$//find the most frequent category for v.
  end for
end for
$P \leftarrow Sort(C(a), s)$.//sort C(a) and obtain the attribute of the former percent s.
Return P

Algorithm A2 IG

Input:

C, A, a, s

Output:

P, *the new feature selection set*

$P \leftarrow \emptyset$.
$m \leftarrow categories(C)$// categories(C) calculates the category number of C.
For each a in A
$p(a) \leftarrow appear(C, a)$// appear(C, a) means the probability of a appearing in C.
$p(-a) \leftarrow noappear(C, a)$//noappear(C, a) means the probability that a does not appear in C.
  For each c in C
    $p(c_a) \leftarrow categories(C, c_a)$// c_a means the categories of C that contain a.
    $P(c_a|a) \leftarrow probability(c_a, a)$ // probability(c_a, a) means the probability that when a appears, c_a appears.
    $P(c_a|high-dimensional) \leftarrow probability(c_a, -a)$ // probability(c_a,-a) means the probability that when a does not appear, c_a appears.
  end for
$G(a) = -\sum_a P(c_a) * \log P(c_a) + P(a) \times \sum_a P(c_t|a) \log P(c_t|a) + P(|-a) \times \sum_a P(c_t|-a) \log P(c_t||-a)$ //calculate the information gain of each a.
end for
$P \leftarrow Sort(G(a), s)$ Sort(G(a), s) means to sort G(a) and obtain the attribute of the former percent s.
Return P

Appendix A.2.3. CHI

The purpose of the CHI algorithm is to assess the degree of independence between words and categories. It is actually a common method to test the independence of two words in mathematical statistics. The basic idea is to verify that the theory is correct by calculating the deviation between the observed and theoretical values. The larger the calculated CHI value is, the more relevant the attribute and category, as shown in Algorithm A3. M represents the number of times the words a and c occur simultaneously; N represents the number of times a appears and c does not occur; J represents the number of times c appears and a does not appear; and Q represents the number of times a and c are not present.

Algorithm A3 CHI

Input:

C, A, a, s

Output:

$P, the\ new feature selection set$

$P \leftarrow \emptyset$.
$R \leftarrow \emptyset$ // R is the chi-squared value set of A.
$D \leftarrow numbers(C)$// numbers(C) means the number of C.
$M, N, J, Q \leftarrow \emptyset$
For each a in A
    For each c in C
$M, N, J, Q \leftarrow values(A, C)$ //values(A, C) calculates the values of M, N, J, Q, respectively.
$\chi^2(a,c) = \frac{D\times(M\times Q-J\times N)}{(N+Q)\times(M+N)\times(J+Q)\times(M+J)}$ // calculates the χ^2 of a for c.
    end for
$R \leftarrow findmax(\chi^2(a,c))$ //find the maximum chi-square value of a.
end for
$P \leftarrow Sort(R, s)$ //sort R and obtain the attribute of the former percent s.
Return P

Appendix A.2.4. Relief

The Relief algorithm is a feature weighting algorithm that assigns different weights according to the correlation of each feature and category. The features with weights less than a certain threshold will be removed. The correlation of features and categories in the Relief algorithm is based on the ability of the features to distinguish between close-range samples. The algorithm randomly selects a sample R from the training set D and then searches for the nearest neighbor sample H from the samples of the same type R, called Near Hit. Then, the algorithm finds the nearest neighbor sample M from the samples of different R types, called near miss, and updates the weight of each feature according to the following rules. If the distance between R, near hit and a feature is less than the distance between R and near miss, it indicates that the feature is beneficial for distinguishing the nearest neighbors of the same type and different classes, and the weight of the feature is increased. Conversely, if the distance between R, near hit and a feature is greater than the distance between R and near miss, indicating that the feature has a negative effect on distinguishing the nearest neighbors of the same class and different class, the weight of the feature is decreased. The above process is repeated m times, and finally, the average weight of each feature is obtained. The greater the weight of the feature is, the stronger the classification ability of the feature and the weaker the ability to classify the feature. The running time of the Relief algorithm increases linearly with the sampling number of samples m and the number of original features N, so the operating efficiency is very high. Algorithm A4 introduces the specific steps.

Algorithm A4 Relief

Input:

training data set D, samples sampling number m, threshold S of feature weight

Output:

Weight T of each feature

Reset all feature weights to 0, *T is an empty set..*
    for $i = 1$ *to m do*
        Randomly select a sample R;
        Find the nearest neighbor sample H of R from the same sample set, and find the nearest neighbor sample M from different sample sets;
for $A = 1$ *to N do*
$W(A) = W(A) - diff(A, R, H)/m + diff(A, R, M)/m$
    for $A = 1$ *to N do*
if $W(A) \geq S$
    Add the $A - th$ *feature to T*
end

References

1. Kumaresh, S.; Baskaran, R. Mining software repositories for defect categorization. *J. Commun. Softw. Syst.* **2015**, *11*, 31–36. [CrossRef]
2. Bertram, D.; Voida, A.; Greenberg, S.; Walker, R. Communication, collaboration, and bugs: The social nature of issue tracking in small, collocated teams. In Proceedings of the 2010 ACM Conference on Computer Supported Cooperative Work, Savannah, GA, USA, 6–10 February 2010; pp. 291–300.
3. Xia, X.; Lo D.; Wang, X.; Zhou, B. Accurate developer recommendation for bug resolution. In Proceedings of the 2013 20th Working Conference on Reverse Engineering (WCRE), Koblenz, Germany, 14–17 October 2013; pp. 72–81.
4. Liu, C.; Yang, J.; Tan, L.; Hafiz, M. R2Fix: Automatically generating bug fixes from bug reports. In Proceedings of the 2013 IEEE Sixth International Conference on Software Testing, Verification and Validation, Luxembourg, 18–22 March 2013; pp. 282–291.
5. Lang, G.; Li, Q.; Guo, L. Discernibility matrix simplification with new attribute dependency functions for incomplete information systems. *Knowl. Inf. Syst.* **2013**, *37*, 611–638. [CrossRef]
6. Guo, S.; Chen, R.; Wei, M.; Li, H.; Liu, Y. Ensemble Data Reduction Techniques and Multi-RSMOTE via Fuzzy Integral for Bug Report Classification. *IEEE Access* **2018**, *6*, 45934–45950. [CrossRef]
7. Zhu, X.; Wu, X. Cost-constrained data acquisition for intelligent data preparation. *IEEE Trans. Knowl. Data Eng.* **2005**, *17*, 1542–1556. [CrossRef]
8. Zhao, H.; Yao, R.; Xu, L.; Yuan, Y.; Li, G.; Deng, W. Study on a novel fault damage degree identification method using high-order differential mathematical morphology gradient spectrum entropy. *Entropy* **2018**, *20*, 682. [CrossRef]
9. Jeong, G.; Kim, S.; Zimmermann, T. Improving bug triage with bug tossing graphs. In Proceedings of the 7th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering, Amsterdam, The Netherlands, 24–28 August 2009; pp. 111–120.
10. AAnvik, J.; Hiew, L.; Murphy, G.C. Who should fix this bug? In Proceedings of the 28th International Conference on Software Engineering, Shanghai, China, 20–28 May 2006; ACM: New York, NY, USA, 2006; pp. 361–370.
11. Deng, W.; Zhao, H.; Zou, L.; Li, G.; Yang, X.; Wu, D. A novel collaborative optimization algorithm in solving complex optimization problems. *Soft Comput.* **2017**, *21*, 4387–4398. [CrossRef]
12. Yang, X.L.; Lo D.; Xia, X.; Huang, Q.; Sun, J.L. High-Impact Bug Report Identification with Imbalanced Learning Strategies. *J. Comput. Sci. Technol.* **2017**, *32*, 181–198. [CrossRef]
13. Naganjaneyulu, S.; Kuppa, M.R.; Mirza, A. An efficient wrapper approach for class imbalance learning using intelligent under-sampling. *Int. J. Artif. Intell. Appl. Smart Dev.* **2014**, *2*, 23–40.

14. Cieslak, D.A.; Chawla, N.V. Learning decision trees for unbalanced data. In Proceedings of the Joint European Conference on Machine Learning and Knowledge Discovery in Databases, Antwerp, Belgium, 14–18 September 2008; pp. 241–256.
15. Chen, R.; Guo, S.; Wang, X.; Zhang, T. Fusion of Multi-RSMOTE with Fuzzy Integral to Classify Bug Reports with an Imbalanced Severity Distribution. *IEEE Trans. Fuzzy Syst.* **2019**. [CrossRef]
16. He, H.; Garcia, E.A. Learning from Imbalanced Data. *IEEE Trans. Knowl. Data Eng.* **2009**, *21*, 1263–1284.
17. Mani, I.; Zhang, I. kNN approach to unbalanced data distributions: a case study involving information extraction. In Proceedings of the Workshop on Learning From Imbalanced Datasets, Washington DC, USA, 21 August 2003; p. 126.
18. Li, H.; Gao, G.; Chen, R.; Ge, X.; Guo, S.; Hao, L. The Influence Ranking for Testers in Bug Tracking Systems. *Int. J. Softw. Eng. Knowl. Eng.* **2019**, *29*, 93–113. [CrossRef]
19. Chawla, N.V.; Bowyer, K.W.; Hall, L.O.; Kegelmeyer, W.P. SMOTE: Synthetic minority over-sampling technique. *J. Artif. Intell. Res.* **2002**, *16*, 321–357. [CrossRef]
20. Mozilla. Available online: http://Mozilla.apache.org/ (accessed on 3 September 2019).
21. GCC. Available online: http://GCC.apache.org/ (accessed on 3 September 2019).
22. Eclipse. Available online: http://Eclipse.apache.org/ (accessed on 3 September 2019).
23. Jiang, H.; Li, X.; Ren, Z.; Xuan, J.; Jin, Z. Toward Better Summarizing Bug Reports with Crowdsourcing Elicited Attributes. *IEEE Trans. Reliab.* **2019**, *68*, 2–22. [CrossRef]
24. Anvik, J. Evaluating an assistant for creating bug report assignment recommenders. *Workshop Eng. Comput. Hum. Interact. Recomm. Syst.* **2016**, *1705*, 26–39.
25. Ai, J.; Su, Z.; Li, Y.; Wu, C. Link prediction based on a spatial distribution model with fuzzy link importance. *Phys. A Stat. Mech. Appl.* **2019**, *527*, 121155. [CrossRef]
26. Deng, W.; Xu, J.; Zhao, H. An improved ant colony optimization algorithm based on hybrid strategies for scheduling problem. *IEEE Access* **2019**, *7*, 20281–20292. [CrossRef]
27. Antoniol, G.; Ayari, K.; Di Penta, M.; Khomh, F.; Guéhéneuc, Y.G. Is it a bug or an enhancement?: A text-based approach to classify change requests. In Proceedings of the CASCON 2008, 18th Annual International Conference on Computer Science and Software Engineering, Conference of the Centre for Advanced Studies on Collaborative Research, Richmond Hill, ON, Canada, 27–30 October 2008.
28. MMenzies, T.; Marcus, A. Automated severity assessment of software defect reports. In Proceedings of the 2008 IEEE International Conference on Software Maintenance, Beijing, China, 28 September–4 October 2008; pp. 346–355.
29. Tian, Y.; Lo D.; Sun, C. DRONE: Predicting Priority of Reported Bugs by Multi-factor Analysis. In Proceedings of the 2013 IEEE International Conference on Software Maintenance, Eindhoven, The Netherlands, 22–28 September 2013; pp. 200–209.
30. Hooimeijer, P.; Weimer, W. Modeling bug report quality. In Proceedings of the Twenty-Second IEEE/ACM International Conference on Automated Software Engineering, Atlanta, GA, USA, 5–9 November 2007; pp. 34–43.
31. Runeson, P.; Alexandersson, M.; Nyholm, O. Detection of Duplicate Defect Reports Using Natural Language Processing. In Proceedings of the 29th International Conference on Software Engineering, Washington, DC, USA, 20–26 May 2007; pp. 499–510.
32. Sun, C.; Lo D.; Wang, X.; Jiang, J.; Khoo, S.C. A discriminative model approach for accurate duplicate bug report retrieval. In Proceedings of the 32nd ACM/IEEE International Conference on Software, Cape Town, South Africa, 1–8 May 2010; pp. 45–54.
33. Sun, C.; Lo D.; Khoo, S.C.; Jiang, J. Towards more accurate retrieval of duplicate bug reports. In Proceedings of the 2011 26th IEEE/ACM International Conference on Automated Software Engineering, Washington, DC, USA, 6–10 November 2011; pp. 253–262.
34. Xia, X.; Lo D.; Wen, M.; Shihab, E.; Zhou, B. An empirical study of bug report field reassignment. In Proceedings of the 2014 Software Evolution Week—IEEE Conference on Software Maintenance, Reengineering, and Reverse Engineering (CSMR-WCRE), Antwerp, Belgium, 3–6 February 2014; pp. 174–183.
35. Zhang, T.; Chen, J.; Yang, G.; Lee, B.; Luo, X. Towards more accurate severity prediction and fixer recommendation of software bugs. *J. Syst. Softw.* **2016**, *117*, 166–184. [CrossRef]
36. Feng, Y.; Chen, Z.; Jones, J.A.; Fang, C.; Xu, B. Test report prioritization to assist crowdsourced testing. In Proceedings of the 10th Joint Meeting of the European Software Engineering Conference

and the ACM SIGSOFT Symposium on the Foundations of Software Engineering, Bergamo, Italy, 30 August–4 September 2015; pp. 225–236.
37. Feng, Y.; Jones, J.A.; Chen, Z.; Fang, C. Multi-objective test report prioritization using image understanding. In Proceedings of the 2016 31st IEEE/ACM International Conference on Automated Software Engineering (ASE), Singapore, 3–7 September 2016; pp. 202–213.
38. Wang, J.; Cui, Q.; Wang, Q.; Wang, S. Towards Effectively Test Report Classification to Assist Crowdsourced Testing. In Proceedings of the 10th ACM/IEEE International Symposium on Empirical Software Engineering and Measurement, Ciudad Real, Spain, 8–9 September 2016.
39. Wang, J.; Wang, S.; Cui, Q.; Wang, Q. Local-based active classification of test report to assist crowdsourced testing. In Proceedings of the 2016 31st IEEE/ACM International Conference on Automated Software Engineering (ASE), Singapore, 3–7 September 2016; pp. 190–201.
40. Cubranic, D.; Murphy, G.C. Automatic bug triage using text categorization. In Proceedings of the SEKE 2004: Sixteenth International Conference on Software Engineering & Knowledge Engineering 2004, Banff, AB, Canada, 20–24 June 2004; pp. 92–97.
41. Xuan, J.; Jiang, H.; Ren, Z.; Yan, J.; Luo, Z. Automatic bug triage using semi-supervised text classification. In Proceedings of 22nd International Conference on Software Engineering and Knowledge Engineering (SEKE 2010), Redwood City, San Francisco Bay, CA, USA, 1–3 July 2010; pp. 209–214.
42. Zhao, H.; Zheng, J.; Xu, J.; Deng, W. Fault diagnosis method based on principal component analysis and broad learning system. *IEEE Access* **2019**. [CrossRef]
43. Bettenburg, N.; Just, S.; Schröter, A.; Weiss, C.; Premraj, R.; Zimmermann, T. What makes a good bug report. In Proceedings of the SIGSOFT 2008/FSE-16, 16th ACM SIGSOFT International Symposium on Foundations of Software Engineering, Atlanta, GA, USA, 9–15 November 2008; pp. 308–318.
44. Gao, K.; Khoshgoftaar, T.M.; Seliya, N. Predicting high-risk program modules by selecting the right software measurements. *Softw. Qual. J.* **2012**, *20*, 3–42. [CrossRef]
45. Deng, W.; Zhao, H.; Yang, X.; Xiong, J.; Sun, M.; Li, B. Study on an improved adaptive PSO algorithm for solving multi-objective gate assignment. *Appl. Soft Comput.* **2017**, *59*, 288–302. [CrossRef]
46. Xuan, J.; Jiang, H.; Hu, Y.; Ren, Z.; Zou, W.; Luo, Z.; Wu, X. Towards Effective Bug Triage with Software Data Reduction Techniques. *IEEE Trans. Knowl. Data Eng.* **2015**, *27*, 264–280. [CrossRef]
47. Xuan, J.; Jiang, H.; Zhang, H.; Ren, Z. Developer recommendation on bug commenting: a ranking approach for the developer crowd. *Sci. China Ser. Inf. Sci.* **2017**, *60*, 072105. [CrossRef]
48. Liu, S.; Hou, H.; Li, X. Feature Selection Method Based on Genetic and Simulated Annealing Algorithm. *Comput. Eng.* **2005**, *31*, 157–159.
49. Jiang, H.; Nie, L.; Sun, Z.; Ren, Z.; Kong, W.; Zhang, T.; Luo, X. Rosf: Leveraging information retrieval and supervised learning for recommending code snippets. *IEEE Trans. Serv. Comput.* **2016**, *12*, 34–46. [CrossRef]
50. Huang, G. An Insight into Extreme Learning Machines: Random Neurons, Random Features and Kernels. *Cogn. Comput.* **2014**, *6*, 376–390. [CrossRef]
51. Guo, S.; Chen, R.; Li, H.; Zhang, T.; Liu, Y. Identify Severity Bug Report with Distribution Imbalance by CR-SMOTE and ELM. *Int. J. Softw. Eng. Knowl. Eng.* **2019**, *29*, 139–175. [CrossRef]