2016 IEEE 40th Annual Computer Software and Applications Conference

Automated Identification of High Impact Bug Reports Leveraging Imbalanced Learning Strategies

Xinli Yang*, David Lo†, Qiao Huang*, Xin Xia*‡, and Jianling Sun*
*College of Computer Science and Technology, Zhejiang University, Hangzhou, China
†School of Information Systems, Singapore Management University, Singapore
zdyxl@zju.edu.cn, davidlo@smu.edu.sg, {tkdsheep, xxia}@zju.edu.cn
, sunjl@zju.edu.cn

***Abstract*—In practice, some bugs have more impact than others and thus deserve more immediate attention. Due to tight schedule and limited human resource, developers may not have enough time to inspect all bugs. Thus, they often concentrate on bugs that are highly impactful. In the literature, high impact bugs are used to refer to the bugs which appear in unexpected time or locations and bring more unexpected effects, or break pre-existing functionalities and destroy the user experience. Unfortunately, identifying high impact bugs from the thousands of bug reports in a bug tracking system is not an easy feat. Thus, an automated technique that can identify high-impact bug reports can help developers to be aware of them early, rectify them quickly, and minimize the damages they cause.**

Considering that only a small proportion of bugs are high impact bugs, the identification of high impact bug reports is a difficult task. In this paper, we propose an approach to identify high impact bug reports by leveraging imbalanced learning strategies. We investigate the effectiveness of various imbalanced learning strategies built upon a number of well-known classification algorithms. In particular, we choose four widely used strategies for dealing with imbalanced data and use naive Bayes multinominal as the classification algorithm to conduct experiments on four datasets from four different open source projects. We perform an empirical study on a specific type of high impact bugs, i.e., surprise bugs, which were first studied by Shihab et al.. The results show that under-sampling is the best imbalanced learning strategy with naive Bayes multinominal for high impact bug identification.

Keywords*—*High Impact Bug, Imbalanced Data, Text Classification

I. Introduction

Bug fixing is a time-consuming and costly task. In the whole life cycle of software development and maintenance, a large number of bugs will be reported which often overwhelm developers [1]. Due to tight schedules and limited human resources, developers often do not have enough time to take care of all bugs. Thus, they will do their best to concentrate on bugs which have high impact.

In recent years, more and more research studies pay close attention to high impact bugs. Ohira et al. create four datasets of high impact bugs by manually reviewing four thousand bug reports in four open source projects (Ambari, Camel, Derby and Wicket) [2]. They introduce six kinds of high impact bugs, i.e., surprise bugs, dormant bugs, blocker bugs, security bugs, performance bugs and breakage bugs. Shihab et al. develop a model to predict if a file contains a high impact bug [3].

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

Identifying high impact bugs early on can help to largely reduce or mitigate the damage caused by those bugs. Unfortunately, considering the large number of bug reports that are received daily by developers, it is often hard for developers to identify those that have high impact. Bug reporters can set the value of the severity field of a bug report to indicate how serious the bug is. Unfortunately, only a minority of bug reporters use this field, and most bug reports have their severity field set to the default value [4]. Moreover, the initial severity field of many bug reports are wrong and it get corrected later on [5]. Thus, there is a need for an automated technique to help developers identify high impact bug reports, which is the goal of this work.

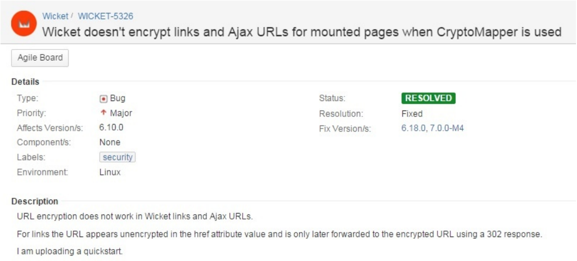
Identifying high impact bug reports is not an easy task. Only a small percentage of bug reports are high impact ones. A bug report dataset is often imbalanced due to the small amount of high impact bugs in a project. Thus, to identify high impact bug reports, we leverage a number of imbalanced learning algorithms for high-impact bug prediction. In particular, we investigate four widely used imbalanced learning strategies.

We focus on one specific type of high impact bugs, i.e., surprise bugs, which are first studied by Shihab et al. [3]. Surprise bugs are bugs which have high impact on developers. These bugs appear in unexpected timing (e.g., in post-release) or locations (e.g., in files that are rarely changed before) and may bring more unexpected effects, catching developers off-guard, disrupting their already-tight quality assurance schedule and workflow.

To evaluate the performance of the different imbalanced learning algorithms, we use four datasets provided by Ohira et al. [2], which contains a total of 2,845 bug reports. We use precision, recall and F1-score as evaluation metrics.

‡Corresponding author.

0730-3157/16 $31.00 © 2016 IEEE
DOI 10.1109/COMPSAC.2016.67



Wicket / WICKET-5326

Wicket doesn't encrypt links and Ajax URLs for mounted pages when CryptoMapper is used

Agile Board

Details

Type:	Bug	Status:	RESOLVED
Priority:	Major	Resolution:	Fixed
Affects Version/s:	6.10.0	Fix Version/s:	6.18.0, 7.0.0-M4
Component/s:	None		
Labels:	security		
Environment:	Linux		

Description

URL encryption does not work in Wicket links and Ajax URLs.

For links the URL appears unencrypted in the href attribute value and is only later forwarded to the encrypted URL using a 302 response.

I am uploading a quickstart.

Fig. 1. An example of a high impact bug report in Wicket.

These metrics are widely used in many software engineering studies [6], [7], [8]. F1-score is a summary measure that combines both precision and recall. A higher F1-score means a better performance. The results show that under-sampling is the best imbalanced learning strategies among the four with naive Bayes multinominal for high impact bug identification.

The main contributions of this paper are:

1) We propose a new problem of identifying high impact bugs. This creates a related but different line of work than the prior work by Shihab et al. which predicts files that contain high impact bugs [3].
2) We propose to use imbalanced learning strategies to deal with the problem of identifying surprise bugs.
3) We conduct an empirical study to investigate the performance of four well-known imbalanced learning strategies built on top of naive Bayes multinominal for high impact bug prediction.
4) We perform experiments on four software projects. The experiment results show that under-sampling is the best imbalanced learning strategies among the four with naive Bayes multinominal for high impact bug identification.

The rest of our paper is organized as follows. Section II briefly presents high-impact bugs. Section III presents the overall framework of our study and elaborates the techniques that we use in our approach. Section IV describes our experiments and the results. Section V discusses the related work. Conclusion and future work are presented in the last section.

II. High-Impact Bugs

As the name implies, high impact bugs are bugs that have high impact to developers and users. Based on prior studies, Ohira et al. summarize six types of high-impact bugs, i.e., surprise bugs, dormant bugs, blocker bugs, security bugs, performance bugs and breakage bugs [2]. In our study, we specifically focus on one type of high impact bugs, i.e., surprise bugs.

Surprise bugs are the bugs which appear in unexpected timings (e.g., in post-release) and locations (e.g., in files that are rarely changed). Shihab et al. show that surprise bugs exist in only 2% of all files [3]. However, surprise bugs may disturb developers' task scheduling greatly.

Figures 1 present an example of a high impact bug report. The bug report (WICKET-4865) describes a bug appearing in the class $CryptoMapper$. Actually, $CryptoMapper$ is rarely changed and bugs rarely appear in $CryptoMapper$. Therefore, the bug is categorized as a surprise bug since the bug appears in an unexpected location.

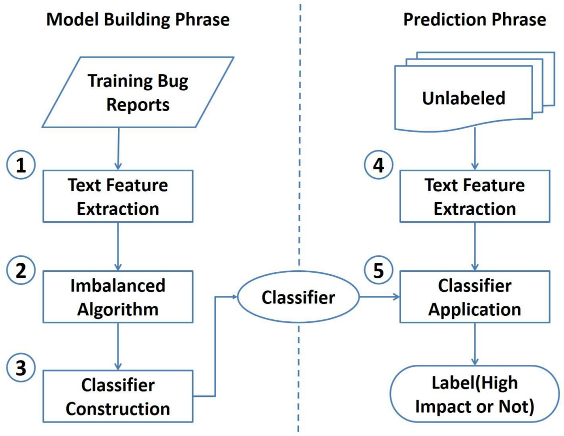


Fig. 2. The Overall Framework of Our Study.

III. Overall Framework

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

A. Overall Framework

Figure 2 presents the overall framework of our proposed approach. The framework mainly contains two phases: the model building phase and the prediction phase. In the model building phase, we build a classifier (i.e., a statistical model) from a training set of bug reports which have been labeled as surprise bugs or not. In the prediction phase, this classifier would be used to identify if an unlabeled bug report would be a surprise or not.

Our framework first extracts a number of features from the training bug reports (Step 1). Features are various quantifiable characteristics of the bugs that could potentially distinguish the surprise (or breakage) bug from the others. In this paper, we use textual features, which are pre-processed words extracted from the summary and description fields of a bug report[1]. Next, we use some imbalanced learning strategies to handle the class imbalance problem (Step 2). We investigate different imbalanced learning strategies[2] for this step. Finally, we build a classifier based on the extracted features (Step 3). We use naive Bayes multinominal as the classification algorithm[3] for this step.

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

[1] Detail information of this process is presented in Section III-B.
[2] Detail information of these techniques is presented in Section III-C.
[3] Detail information of this technique is presented in Section III-D.

B. Feature Extraction

In a bug report, summary and description fields contain most of the useful information for prediction. Therefore, we extract features from these two fields. We first extract all the terms (i.e., words) from the summary and description fields in a bug report. Then, we remove the stop words, numbers and punctuation marks since they provide little information. For the remaining terms, we use Iterated Lovins Stemmer [9] to transform them to their root forms (e.g., 'reading' and 'reads' are reduced to 'read'). We do this stemming step in order to reduce the feature dimensions and to unify similar words into a common representation. Finally, we calculate the term frequency for each stemmed term. After these steps, a bug report b is represented as a term frequency vector, i.e., $b = (w_1, w_2 ... w_n)$, where w_i denotes the number of times the i^{th} term appears in the bug report b. Also, we remove terms which only appear once in one bug report to reduce noise.

C. Imbalanced Learning Strategies

Class imbalance is always a big problem in machine learning. It can lead to a classifier having poor performance. Imbalanced learning strategies can be employed to balance the initial imbalanced dataset and help the trained classifier to not to be biased to the majority class. Thus, in most cases, it can improve the performance of the classifier [10], [11].

There are many different imbalanced learning strategies. In our study, we investigate four well-known strategies [12], [13]. Three of them are sampling methods, namely random under-sampling, random over-sampling and SMOTE. The last one is cost-sensitive methods, which adjusts the cost matrix. We refer to it as cost-matrix adjuster. We introduce each of them briefly. For simplicity, the subsets of data belonging to the minority class (in our case: high-impact bug reports of a particular category) and the majority class (in our case: all other bug reports) are denoted by S_{min} and S_{maj}, respectively.

1) Random Under-Sampling: Under-sampling is one of the effective sampling methods to deal with the class imbalance problem [12], [13]. It deletes data belonging to S_{maj} in order to shrink its scale. Generally, under-sampling first sets a value p, which is the target ratio of data instances belonging to the majority class to the entire data used for training. Then, it repeats the following two steps until the ratio of S_{maj} to the entire data decreases to p:

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

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

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

2) Random Over-Sampling: Over-sampling is another effective approach to deal with the class imbalance problem [12], [13]. It duplicates data belonging to S_{min} in order to expand its scale. Generally, over-sampling first set a value p, which is the target ratio of data instances belonging to the minority class to the entire data used for training. Then it repeats the following two steps until the ratio of S_{min} to the entire data increases to p:

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

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

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

3) SMOTE: SMOTE is a more sophisticated over-sampling method, whose full name is Synthetic Minority Over-sampling TEchnique [14], [13]. Traditional over-sampling methods duplicate data belonging to S_{min}, while SMOTE creates some artificial data which can be assumed to belong to S_{min} based on a specific strategy. Specifically, for each data point x in S_{min}, SMOTE first finds its k-nearest neighbours (data points) belonging to S_{min} and link x with each of these k points to form k line segments (in a multidimensional feature space). Then, SMOTE randomly picks a data point on each line segment. The k new data points can be assumed as belonging to the minority class and be added into S_{min}. Therefore, if there are initially n data points in S_{min}, SMOTE will create $k \times n$ artificial data points and add them to S_{min}. By default, k is set as 5.

4) Cost-Matrix Adjuster: Cost-matrix adjuster is a popular cost-sensitive method to deal with the data imbalance problem [12], [13]. Different from the previous three methods, it does not delete or add any data point to S_{maj} or S_{min}. Instead, it changes the cost of misclassifying different training instances belonging to different classes. It makes the cost of misclassifying instances in S_{min} larger than that of S_{maj} so that the classifier will value S_{min} more than S_{maj}.

By default, the cost matrix of many classifiers is:

$$\begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix} \tag{1}$$

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

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

$$\begin{bmatrix} 0 & y \\ x & 0 \end{bmatrix} \tag{2}$$

D. Naive Bayes Multinomial

To introduce Naive Bayes Multinominal (NBM), we first show the base version of it, i.e., Naive Bayes (NB).

NB is a probabilistic model based on Bayes theorem for conditional probabilities [12]. Naive Bayes assumes that features are independent from one another. Also, all the features are binominal. That is, each feature only has two values of 0 and 1 (in our case, representing whether a word exists in a bug report or not).

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

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

With Bayes theorem, we can have compute the probability of a label c_j given the BR as follows:

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

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

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

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

NBM is very similar with NB [12]. However, in NBM the value of each feature is not restricted to 0 or 1, rather it can be any non-negative number (in our case, representing the frequency of a word in a bug report). Since NBM can capture more information, it often outperforms NB.

IV. Experiments and Results

In our study, the experimental environment is an Intel(R) Core(TM) T6570 2.10 GHz CPU, 4GB RAM desktop running Windows 7 (64-bit). The platform is Eclipse, and the algorithms we use are built in Weka. We first present our experimental setting and evaluation metrics in Sections IV-A to IV-C. We then present the research question and our experiment result that answer the question in Section IV-D.

A. Datasets

We perform experiments on four datasets from four well-known open source projects, which are Ambari, Camel, Derby, Wicket, containing a total of 2845 bug reports. All the bug reports are collected and manually categorized by Ohira et al. [2]. Table I summarizes the statistics of each dataset, containing the total number of bug reports (BRs), the number of surprise BRs and the ratio of surprise BRs to all the BRs. We can see that all the datasets are imbalanced, especially in Derby, the ratio of surprise BRs is only about 15%.

TABLE I. Statistics Of The Datasets Used In Our Study

Project	# Total BRs	# Surprise BRs	Ratio (%)
Ambari	871	266	30.54%
Camel	579	228	39.38%
Derby	731	111	15.18%
Wicket	663	242	36.50%

B. Experimental Settings

In the experiments, we use the default values of these imbalanced learning strategies. We use stratified ten-fold cross validation [12] to evaluate the effectiveness of various imbalanced learning strategies. Following stratified 10-fold cross validation, each of the dataset is divided into 10 folds, where each fold contains more or less equal proportion of instances belonging to each class. Next, we perform 10 evaluation rounds; in each round, 9 folds are used as a training dataset, and the remaining one fold is used as a testing dataset. We aggregate the results of the 10 evaluation rounds and report the overall performance. Stratified cross validation is a standard evaluation setting, which is widely used in software engineering studies [15], [16], [17]. Since stratified ten-fold cross validation involves randomness, to increase the confidence of the results, we repeat it 10 times and report the average results.

C. Evaluation Metrics

We use precision, recall and F1-score as evaluation metrics. These metrics are commonly-used measures to evaluate classification performance [6], [7], [8]. They can be derived from a confusion matrix, as shown in Table II. The confusion matrix lists all four possible classification results. If a bug report is correctly classified as “surprise”, it is a true positive (TP); if a bug report is misclassified as “surprise”, it is a false positive (FP). Similarly, there are false negatives (FN) and true negatives (TN). Based on the four numbers, Precision, recall and F1-score are calculated. Precision is the ratio of correctly predicted “surprise” bug reports to all bug reports predicted as “surprise”. Mathematically, precision is defined as $\frac{TP}{TP+FP}$. Recall is the ratio of the number of correctly predicted “surprise” bug reports to the actual number of “surprise” bug reports. Mathematically, recall is defined as $\frac{TP}{TP+FN}$). Finally, F1-score is a harmonic mean of precision and recall. Mathematically, F1-score is defined as $\frac{2*Recall*Precision}{Recall+Precision}$). F1-score is often used as a summary measure to evaluate if an increase in precision outweighs a reduction in recall (and vice versa).

TABLE II. Confusion Matrix

	Predicted (to be) Important	Predicted (to be) Unimportant
Truly Important	TP	FN
Truly Unimportant	FP	TN

D. Research Questions

Our experiments are designed to answer the following research question: **Which of the four imbalanced learning strategies perform the best?**

Motivation. Since generally different imbalanced learning strategies are suitable to different problem, we want to investi-

TABLE III. PRECISION OF FOUR IMBALANCED LEARNING STRATEGIES FOR SURPRISE BUG IDENTIFICATION

Project	RUS	ROS	SMOTE	CMA
Ambari	0.3326	0.3427	0.3236	0.3450
Camel	0.4164	0.4253	0.4299	0.4220
Derby	0.1632	0.1844	0.2913	0.1931
Wicket	0.3720	0.3749	0.3761	0.3619
Average	0.3211	0.3318	0.3552	0.3305

TABLE IV. RECALL OF FOUR IMBALANCED LEARNING STRATEGIES FOR SURPRISE BUG IDENTIFICATION

Project	RUS	ROS	SMOTE	CMA
Ambari	0.5966	0.4350	0.3872	0.4812
Camel	0.6654	0.5798	0.6680	0.6053
Derby	0.4937	0.2027	0.1577	0.2523
Wicket	0.4694	0.3421	0.3983	0.3843
Average	0.5563	0.3899	0.4028	0.4308

gate which of the imbalanced learning strategies performs the best for identifying surprise bug reports.

Approach. To answer this question, we compare four imbalanced learning strategies, i.e., random under-sampling (RUS), random over-sampling (ROS), SMOTE and cost-matrix adjuster (CMA). We use naive Bayes multinominal as the classification algorithm and use the three evaluation metrics mentioned above to compare the four different strategies.

Results. Tables III to V present the precision, recall, and F1-score values of the four imbalanced learning strategies for surprise bug identification. From these tables, we can conclude several points.

First, from Table V, we can see that random under-sampling achieves an average F1-score of 0.40 for surprise bug identification, which is the best performance compared with the other imbalanced learning strategies.

Second, from Tables III and IV, we can find that in terms of precision, SMOTE performs the best, though the performance gaps of the different strategies are relatively small. However, in terms of recall, random under-sampling is the best performer and it exceeds the other methods much. It achieves an average recall of 56% for surprise bug identification, while all the others only have an average recall of 35% to 45%. Considering the setting of our problem, recall is more important than precision because as many high impact bugs should be identified as possible.

Random under-sampling is more effective than the other imbalanced learning strategies. On average, it improves F1-score by 7% on the surprise bug identification compared to the next best performing imbalanced learning strategies (i.e., cost-matrix adjuster).

TABLE V. F1-SCORE OF FOUR IMBALANCED LEARNING STRATEGIES FOR SURPRISE BUG IDENTIFICATION

Project	RUS	ROS	SMOTE	CMA
Ambari	0.4270	0.3833	0.3525	0.4019
Camel	0.5121	0.4907	0.5231	0.4973
Derby	0.2452	0.1931	0.2045	0.2188
Wicket	0.4150	0.3575	0.3868	0.3727
Average	0.3998	0.3562	0.3667	0.3727

E. Threats to Validity

Threats to construct validity relate to the suitability of our evaluation metrics. We use precision, recall and F1-score which are also used by many past software engineering studies to evaluate the effectiveness of various classification techniques [6], [7], [8]. Thus, we believe there is little threat to construct validity.

Threats to internal validity relate to errors in our experiments or bias due to randomization. We have double checked our implementations and we repeat all the experiments 10 times to report the average performance. Hence, we believe there are minimal threats to internal validity.

Threats to external validity relate to the generalizability of our results. We have evaluated our approach on 2,845 bug reports from four open source projects. In the future, we plan to reduce this threat further by analyzing more datasets from more open source software projects and commercial software projects.

V. RELATED WORK

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

A. High Impact Bugs

The most related works to ours are the recent studies by Ohira et al. [2] and Shihab et al. [3]. Ohira et al. create four datasets of high impact bugs by manually reviewing four thousand bug reports in four open source projects (Ambari, Camel, Derby and Wicket) [2]. They introduce six kinds of high impact bugs, i.e., surprise bugs, dormant bugs, blocker bugs, security bugs, performance bugs and breakage bugs. In addition, they classify them into two types, i.e., process and product. A bug management process in a project will be impacted by the first three kinds of bugs, while user experience and satisfaction with software products will be affected by the last three kinds of bugs. Shihab et al. develop prediction models to identify if a file contains a breakage or surprise bug [3]. In this work, we investigate the usage of text mining and imbalanced learning strategies to identify high impact bug reports in a collection of bug reports. This is a related but *different* problem than the one considered by Shihab et al.. Rather than predicting if a file contains a breakage or surprise bug, we *identify* breakage and surprise bug reports from a collection of bug reports.

Aside from the two works highlighted above, there are also other studies that are about high impact bugs [18], [17]. Zaman et al. conduct a case study on the Firefox project to demonstrate the difference between performance and security bugs [18]. Garcia and Shihab study blocking bugs in six open source projects and propose a model to identify them [17].

In this paper, we propose an approach that can identify bug reports that correspond to surprise and breakage bugs. We evaluate many variants of our approach using four datasets created by Ohira et al. [2]. We have shown that the best variant of our approach outperforms the state-of-the-art high-impact bug report identification approach by Garcia and Shihab [17].

B. Bug Categorization

There are many studies on bug prediction in the literature [19], [20]. Huang et al. propose a novel Orthogonal Defect Classification (ODC) system by integrating experts' experience and domain knowledge [19]. Thung et al. propose a text mining solution that can categorize bugs into various types [20]. They compare six classic classification algorithms and conclude that SVM achieves the best performance for automatic bug categorization.

In this paper, we have compared our work against a state-of-the-art work that automatically categorizes bugs, i.e., [20]. Our experiments demonstrate that the best performing variant of our approach which leverages under sampling outperforms that work.

C. Imbalanced Learning Strategies

There are a number of software engineering studies which leverage imbalanced learning strategies [21], [11]. We highlight some of them below. Due to the page limit, the survey here is by no means complete.

Kamei et al. investigate the effectiveness of over and under sampling strategies on fault-prone module detection [21]. They evaluate the performance of four sampling methods applied to four fault-prone detection models. They conclude that all the four sampling methods can improve the prediction performance. Wang et al. use class imbalance learning for software defect prediction [11]. They investigate different types of imbalanced learning strategies and propose a dynamic version of AdaBoost.NC, which is an ensemble learning method that automatically adjusts its parameters during training.

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

VI. Conclusion and Future Work

In this paper, we leverage imbalanced learning strategies to identify high impact bug reports. We investigate four widely used imbalanced learning strategies (i.e., random under-sampling, random over-sampling, SMOTE and cost-matrix adjuster) and use naive Bayes multinominal as the classification algorithm to perform experiments on datasets from four different open source projects. We focus on a specific type of high impact bugs, i.e., surprise bugs, which are first studied by Shihab et al. [3]. The results show that under-sampling is the best imbalanced learning strategy with naive Bayes multinominal for high impact bug identification.

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

References

[1] J. Anvik, L. Hiew, and G. C. Murphy, "Coping with an open bug repository," in *Proceedings of the 2005 OOPSLA workshop on Eclipse Technology eXchange, ETX 2005, San Diego, California, USA, October 16-17, 2005*, 2005, pp. 35–39.

[2] M. Ohira, Y. Kashiwa, Y. Yamatani, H. Yoshiyuki, Y. Maeda, N. Limsettho, K. Fujino, H. Hata, A. Ihara, and K. Matsumoto, "A dataset of high impact bugs: Manually-classified issue reports," in *Mining Software Repositories (MSR), 2010 7th IEEE Working Conference on*. IEEE, 2015, pp. 518–521.

[3] E. Shihab, A. Mockus, Y. Kamei, B. Adams, and A. E. Hassan, "High-impact defects: a study of breakage and surprise defects," in *Proceedings of the 19th ACM SIGSOFT symposium and the 13th European conference on Foundations of software engineering*. ACM, 2011, pp. 300–310.

[4] A. Lamkanfi, S. Demeyer, E. Giger, and B. Goethals, "Predicting the severity of a reported bug," in *Mining Software Repositories (MSR), 2010 7th IEEE Working Conference on*. IEEE, 2010, pp. 1–10.

[5] X. Xia, D. Lo, M. Wen, E. Shihab, and B. Zhou, "An empirical study of bug report field reassignment," in *2014 Software Evolution Week - IEEE Conference on Software Maintenance, Reengineering, and Reverse Engineering, CSMR-WCRE 2014, Antwerp, Belgium, February 3-6, 2014*, 2014, pp. 174–183.

[6] J. Nam, S. J. Pan, and S. Kim, "Transfer defect learning," in *Proceedings of the 2013 International Conference on Software Engineering*. IEEE Press, 2013, pp. 382–391.

[7] X. Xia, D. Lo, E. Shihab, X. Wang, and X. Yang, "Elblocker: Predicting blocking bugs with ensemble imbalance learning," *Information and Software Technology*, vol. 61, pp. 93–106, 2015.

[8] X. Xia, D. Lo, E. Shihab, and X. Wang, "Automated bug report field reassignment and refinement prediction."

[9] J. B. Lovins, "Development of a stemming algorithm," *Mechanical Translation and Computational Linguistics*, vol. 11, pp. 22–31, 1968.

[10] Y. Kamei, A. Monden, S. Matsumoto, T. Kakimoto, and K.-i. Matsumoto, "The effects of over and under sampling on fault-prone module detection," in *Empirical Software Engineering and Measurement, 2007. ESEM 2007. First International Symposium on*. IEEE, 2007, pp. 196–204.

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

[12] J. Han and M. Kamber, *Data Mining: Concepts and Techniques*. Morgan kaufmann, 2006.

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

[14] N. V. Chawla, K. W. Bowyer, L. O. Hall, and W. P. Kegelmeyer, "S-mote: synthetic minority over-sampling technique," *Journal of artificial intelligence research*, pp. 321–357, 2002.

[15] X. Xia, Y. Feng, D. Lo, Z. Chen, and X. Wang, "Towards more accurate multi-label software behavior learning," in *Software Maintenance, Reengineering and Reverse Engineering (CSMR-WCRE), 2014 Software Evolution Week-IEEE Conference on*. IEEE, 2014, pp. 134–143.

[16] X. Xia, D. Lo, X. Wang, and B. Zhou, "Tag recommendation in software information sites," in *Proceedings of the 10th Working Conference on Mining Software Repositories*. IEEE Press, 2013, pp. 287–296.

[17] H. V. Garcia and E. Shihab, "Characterizing and predicting blocking bugs in open source projects," in *Proceedings of the 11th Working Conference on Mining Software Repositories*. ACM, 2014, pp. 72–81.

[18] S. Zaman, B. Adams, and A. E. Hassan, "Security versus performance bugs: a case study on firefox," in *Proceedings of the 8th working conference on mining software repositories*. ACM, 2011, pp. 93–102.

[19] L. Huang, V. Ng, I. Persing, R. Geng, X. Bai, and J. Tian, "Autoodc: Automated generation of orthogonal defect classifications," in *Automated Software Engineering (ASE), 2011 26th IEEE/ACM International Conference on*. IEEE, 2011, pp. 412–415.

[20] F. Thung, D. Lo, and L. Jiang, "Automatic defect categorization," in *Reverse Engineering (WCRE), 2012 19th Working Conference on*. IEEE, 2012, pp. 205–214.

[21] Y. Kamei, A. Monden, S. Matsumoto, T. Kakimoto, and K.-i. Matsumoto, "The effects of over and under sampling on fault-prone module detection," in *Empirical Software Engineering and Measurement, 2007. ESEM 2007. First International Symposium on*. IEEE, 2007, pp. 196–204.